Expert Oracle9*i* Database Administration

SAM R. ALAPATI

Apress™

ISBN (pbk): 1-59059-022-8
Printed and bound in the United States of America 12345678910

Technical Reviewer: Kee Heng Tiow

Editorial Directors: Dan Appleman, Gary Cornell, Simon Hayes, Martin Streicher, Karen Watterson, John Zukowski

Assistant Publisher: Grace Wong

Project Manager: Alexa Stuart

Development Editor: Tracy Brown Collins

Copy Editor: Nicole LeClerc

Production Editor: Laura Cheu

Compositor: Diana Van Winkle, Van Winkle Design Group

Indexer: Julie Kawabata

Artist and Cover Designer: Kurt Krames

Production Manager: Kari Brooks

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, email `orders@springer-ny.com`, or visit `http://www.springer-ny.com`. Outside the United States: fax +49 6221 345229, email `orders@springer.de`, or visit `http://www.springer.de`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, email `info@apress.com`, or visit `http://www.apress.com`.

CHAPTER 19

Performance Tuning: Tuning the Instance

In the previous chapter, you learned how to tune an application by writing efficient SQL in order to maximize its performance. The use of optimal SQL, efficient design of the layout of the database objects, and so on are all part of a planned or proactive tuning effort. This chapter focuses on the efficient use of the resources Oracle works with: memory, CPU, and storage disks.

The chapter discusses how to monitor and optimize memory allocation for the Oracle instance. In this context, you'll learn about the traditional database hit ratios, such as the buffer cache hit ratios. However, focusing on the hit ratios isn't the smartest way to maintain efficient Oracle databases, as you need to focus on the user's response time. Investigating factors that are causing processes to spend excessive time waiting for resources is a better approach to performance tuning. This chapter provides you with a solid introduction to Oracle wait events and tells you how to interpret them and reduce the incidence of these wait events in your system.

A fairly common problem in many production systems is that of a database *hang,* when things seem to come to a standstill for some reason. The chapter explores several potential showstoppers and shows you ways to avoid the occurrence of database hangs.

To tune the instance, you need to collect the relevant instance statistics. In this chapter you'll learn how to use the useful Oracle Statspack to collect performance data. The chapter explains the key dynamic performance tables that you need to be familiar with to understand instance performance issues. You'll also see how the OEM Diagnostics Pack can help you monitor and tune instance performance.

Although it's nice to be able to proactively design a system for high performance, more often than not, the DBA has to deal with proactive tuning when performance is unsatisfactory and a fix needs to be found right away. The final part of this chapter deals with a simple methodology to follow when your system performance deteriorates and you need to fine-tune the Oracle instance.

I begin this chapter with a short introduction to instance tuning and then turn to cover in detail the tuning of crucial resources such as memory, disk, and CPU usage. Later on in the chapter, I review the important Oracle wait events, which will help you get a handle on several kinds of database performance issues.

An Introduction to Instance Tuning

Oracle doesn't give anything but minimal and casual advice regarding the appropriate settings of key resources such as total memory allocation or the sizes of the components of memory. Oracle has some general guidelines about the correct settings for several key initialization parameters that have a bearing on performance. However, beyond specifying wide ranges for the parameters, the company's guidelines aren't very helpful to DBAs deciding on the "optimal" levels for these parameters.

Oracle says this is because all these parameters are heavily application dependent. All of this means that you as a DBA have to find out what the optimal sizes of resource allocations and the ideal settings of key initialization parameters are, through trial and error. You have to be resourceful when it comes to memory sizing: If you need more shared pool memory, first see if you can drop the buffer cache some, and vice versa. In this chapter, I provide some guidelines for figuring out the optimal sizes of memory and other configuration parameters.

As a DBA, you're often called in to tune the instance when users perceive a slow response, which is in turn caused by a bottleneck somewhere in the system. This bottleneck is the result of either an excessive use of or insufficient provision of one of the resources such as memory, CPU, or disks. In addition, database locks and latches may cause a slowdown. You have to remember, though, that in most cases, the solution isn't to simply increase the resource that seems to be getting hit hard—that may be the symptom, not the cause of a problem. If you address the performance slowdown by "fixing" the symptoms, the root causes will remain potential troublemakers.

Performance tuning an Oracle database instance involves tuning memory and I/O, as well as operating system resources such as CPU, the operating system kernel, and the operating system memory allocation. In the previous chapter, you saw how tuning the application leads to faster response times and more throughput. Application code changes, however, are incremental, and once the application is in production, instance tuning takes center stage in the tuning efforts.

When you receive calls from the help desk or other users of the system complaining that the system is running slowly, you can only change what is under your direct control—mainly, the allocation of memory and its components, and some dynamic initialization parameters that have a bearing on instance performance. Depending on what the various indicators tell you, you may adjust the shared pool and other components of memory to improve performance. You can also change the operating system priority of some processes, or quickly add some disk drives to your system.

One of the main reasons for a slow response time in a production system is the waiting time incurred by user processes. Oracle provides several ways of monitoring these waits, but you need to understand their significance in your system. Long wait times are not the problem themselves; they're symptoms of deep-seated problems. The DBA should be able to connect different types of waits with possible causes in the application or in the instance.

Although some manuals will tell you that you should do performance tuning before application tuning—before you proceed to tuning areas such as memory, I/O, and contention—real life isn't so orderly. Most of the time, you don't have the opportunity to have the code revised, even if there are indications that it isn't optimal. Instead of being an orderly process, tuning databases is an iterative process, where you may have to go back and forth between stages.

More often than not, DBAs are forced to do what they can to fix the performance problem that is besetting them at that moment. In this sense, most performance tuning is a reactive kind of tuning. Nevertheless, DBAs should endeavor to understand the innards of wait issues and seek to be proactive in their outlooks.

There are two big advantages to being in a proactive mode of tuning. First, you have fewer sudden performance problems that force hurried reactions. Second, as your understanding of your system increases, so does your familiarity with the various indicators of poor performance and the likely causes for them, so you can resolve problems that do occur much faster.

If you're fortunate enough to be with an application during its design stages, you can improve performance by performing several steps, including choosing automatic space management and setting correct storage options such as PCTFREE for your tables and indexes. Sizing the table and indexes correctly doesn't hurt, either. If you're stuck with a database that has a poor design, however, all is not lost. You can still tune the instance using techniques that I show later in this chapter to improve performance.

When response time is slower than usual, or when throughput falls, you'll notice that the Oracle instance isn't performing at its usual level. If response times are higher, obviously there's a problem somewhere in one of the critical resources Oracle uses. If you can rule out any network slowdowns, that leaves you with memory (Oracle's memory and the system's memory), the I/O system, and CPUs. One of these resources is usually the bottleneck that's slowing down your system.

In the next few sections, you'll learn how to tune key system resources such as memory and CPU to improve performance. You'll also see how to measure performance, detect inefficient waits in the system, and resolve various types of contention in an Oracle database. The next section presents a discussion of how tuning Oracle's memory can help improve database performance.

Patches and New Versions of Software

Oracle Corporation, like the other software vendors, releases periodic *patches* or *patch sets,* which are a set of fixes for bugs discovered by either Oracle or its customers. When you get in touch with Oracle technical support, one of the things the technical support representative will commonly ask you to do is make sure you have applied the latest patch set to your Oracle software. Similarly, UNIX operating systems may have their own patch sets that you may have to apply to fix certain bugs.

Each of Oracle's patch sets could cover fixes for literally hundreds of bugs. My recommendation is to apply a patch set as soon as it is available. One of the primary reasons for this is to see if your bug is unique to your database, or if a general solution has already been found for the problem. When you ask Oracle technical support to resolve a major problem caused by a bug, Oracle will usually provide you with a workaround. Oracle recommends that you upgrade your database to the latest versions and patch sets, because some Oracle bugs may not have any workarounds or fixes. Oracle will continue to support older versions of its server software throughout their support life cycle, which is usually about 2 to 3 years after the release of the next major release. Many organizations see no urgency to move to newer versions, as Oracle continues to support the older versions after the release of the new versions.

The question regarding how fast you should convert to a new version is somewhat tricky to answer. Traditionally, people have shied away from being early adopters of new Oracle software versions. Oracle, like most other software companies, has a reputation for very buggy initial releases of their major software versions (such as the 9.0 version of the 9*i* server software). Therefore, DBAs and managers in general prefer to wait a while until a "stable version" comes out. Although the logic behind this approach is understandable, you must also figure in the cost of not being able to use the many powerful features Oracle introduces in each of its major releases.

Because nobody likes to jeopardize the performance of a production system, the ideal solution is to maintain a test server where the new server is tested thoroughly and then move into production as early as possible. Oracle does provide support for a version for several years after its release, so companies have no urgent reason to migrate to uncertain new versions. However, don't wait forever to move to a new version—by the time some companies move to the new version, an even newer Oracle version is already out!

One of the best ways to improve performance is to move aggressively to the newer versions and take advantage of the powerful features they offer. For example, the new cursor-sharing features of Oracle8*i* and 9*i* have improved performance tremendously when compared to the older versions. I haven't met anybody yet who complains that his or her system is slower with the newer version. Of course, there's a caveat here: Some of your good SQL statements may not be so good after you migrate to a new version, due to the way a hint might behave in the new version, for example. That's why it's extremely important to test the whole system on the new version before cutting over production systems.

A smart strategy is to collect a set of performance statistics that can serve as a baseline, before you make any major changes in the system. These system changes may include the following:

- Migrating or upgrading a database
- Applying a new database or operating system patch set
- Adding a new application to your database
- Substantially increasing the user population

Tuning Oracle Memory

A well-known fact of system performance is that fetching data that's stored in memory is a lot faster than retrieving data from disk storage. Given this, Oracle tries to keep as much of the recently accessed data as possible in its SGA. In addition to data, shared parsed SQL code and necessary data dictionary information are cached in memory for quick access. Memory allocated to Oracle is highly configurable and is one of the easiest things to adjust, especially with the new Oracle9*i* version.

There is a two-way relationship between memory configuration and the application's use of that memory. The correct memory allocation size depends on the nature of your application, the number of users, and the size of transactions. If there is not enough memory, the application will have to perform time-consuming disk I/Os, but the application itself might be using memory unnecessarily, and throwing more memory at it may not be the right strategy.

As a DBA, you must not view memory and its sizing in isolation—this can lead to some poor choices, as you address the symptoms instead of the causes for what seems like insufficient memory. The tendency on a DBA's part is to allocate as much memory as possible to the shared pool, hoping that doing so will resolve the problem, but sometimes this only exacerbates the problem. It is wise to manage the database with as little memory as necessary, and no more. The system can always use the free memory to ensure there is no swapping or paging, which can slow down your application. Performance slowdowns caused by paging outweigh the benefits of a larger SGA under most operating systems.

The SGA has several components. Of these components, the redo log buffer and the Java pool are relatively small components. You need to allocate the Java pool only for applications that use Java. The main components of the SGA that you have to worry about are the following:

- Shared pool
- Buffer cache
- Large pool
- Redo log buffer
- Process-private memory

The total size of the SGA is the sum of all its components, as shown in the following query:

```
SQL> select * from v$sga;
NAME                      VALUE
-------------------- ----------
Fixed Size               736304
Variable Size         117440512
Database Buffers       33554432
Redo Buffers            2371584
SQL>
```

In this listing, the database buffers column shows the database buffer cache memory, and the variable size column consists mostly of the shared pool component. As shown in Chapter 16, you can dynamically change the settings of all the SGA components except the redo log buffer and the Java pool. That is, you don't have to restart the instance if you need to adjust the important components of the SGA.

In the following sections you'll look at each of these memory components in detail.

TIP *The* max_sga_size *initialization parameter is purely optional. However, if you don't set it, the maximum size will be the sum of all its components that you explicitly set. If you want to dynamically increase one component, you're forced to decrease some other component to make room for it in the SGA. To avoid this dilemma, always set the* max_sga_size *parameter to a value larger than the sum of its components.*

Tuning the Shared Pool

In a production database, the shared pool is going to command most of your attention because of its direct bearing on application performance. The shared pool is a part of the SGA that holds almost all the necessary elements for execution of the SQL statements and PL/SQL programs. In addition to caching program code, the shared pool caches the data dictionary information that Oracle needs to refer to often during the course of program execution.

Proper shared pool configuration leads to dramatic improvements in performance. An improperly tuned shared pool leads to problems such as the following:

- Fragmentation of the pool
- Increased latch contention with the resulting demand for more CPU resources
- Greater I/O because executable forms of SQL are not present in the shared pool
- Higher CPU usage because of unnecessary parsing of SQL code

The general increase in shared pool waits and other waits observed during a severe slowdown of the production database is the result of SQL code that fails to use bind variables. As the number of users increases, so does the demand on shared pool memory and latches, which are internal locks for memory areas. The result is a higher wait time and a slower response time. Sometimes the entire database seems to "hang."

The shared pool consists of two major areas: the library cache and the data dictionary cache. You can't allocate or decrease memory specifically for one of these components. If you increase the total shared pool memory size, both components will increase in some ratio that is determined by Oracle. Similarly, when you decrease the total shared pool memory, both components will decrease in size. Let's look at these two important components of the shared pool in detail.

The Library Cache

The *library cache* holds the parsed and executable versions of SQL and PL/SQL code. As you may recall from Chapter 18, all SQL statements undergo the following steps during their processing:

- *Parsing,* which includes syntactic and semantic verification of the statements and checking necessary object privileges to perform the actions.
- *Optimization,* where the Oracle optimizer evaluates how to process the statement with the least cost, after it evaluates several alternatives.
- *Execution,* where Oracle uses the optimized physical execution plan to perform the action stated in the SQL statement.
- *Fetching,* which only applies to *select* statements where Oracle has to return rows to you. This step isn't necessary, of course, in any nonquery-type statements.

Parsing is a very resource-intensive operation, and if your application needs to execute the same SQL statement repeatedly, having a parsed version of the same in memory will reduce contention for latches, CPU, I/O, and memory usage. The first time Oracle parses a statement, it creates a *parse tree.* The optimization step is necessary only for the first execution of a SQL statement. Once the statement is optimized, the best access path is encapsulated in the *access plan.* Both the parse tree and the access plan are stored in the library cache before the statement is executed for the first time. Future invocation of the same statement will need to go through only the last stage, execution, which avoids the overhead of parsing and optimizing as long as Oracle can find the parse tree and access plan in the library cache. Of course, if the statement is a SQL query, the last statement will be the fetch operation.

The library cache, being limited in size, will discard aged SQL when there is no more room for new SQL statements. The only way you can use a parsed statement repeatedly for multiple executions is if all the SQL statements are identical. Two statements are considered identical if they have exactly the same code, *including case and spaces.* The reason for this is that when Oracle compares a new statement to existing statements in the library cache, it uses simple string comparisons. In addition, any bind variables used must be similar in *data type and size.* Here are a couple of examples that show you how picky Oracle is when it comes to considering whether two SQL statements are identical.

In the following example, the statements are not considered identical because of an extra space in the second statement:

```
select * from employees;
select *  from employees;
```

In this example, the statements are not considered identical because of the different case used for the table Employees in the second statement. The two versions of "employees" are termed *literals* because they are literally different from each other.

```
Select * from employees;
Select * from Employees;
```

Hard Parsing and Soft Parsing

You may recall from the last chapter that all SQL code goes through the parse, optimize, and execute phases. When an application issues a statement, Oracle will first see if a parsed version of the statement already exists. If it does, the result is a so-called soft parse and is considered a library cache hit. If, during a parse phase or the execution phase, Oracle is not able to find the parsed version or the executable version of the code in the shared pool, it will perform a *hard parse,* which means that the SQL statement has to be reloaded into the shared pool and parsed completely.

During a hard parse, Oracle performs syntactic and semantic checking, checks the object and system privileges, builds the optimal execution plan, and finally loads it into the library cache. A hard parse involves a lot more CPU usage and is inefficient compared to a soft parse, which depends on reusing previously parsed statements. Hard parsing involves building all parse information from scratch, and therefore it is more resource intensive. Besides involving a higher CPU usage, hard parsing involves a large number of latch gets, which may increase the response time of the query. The ideal situation is where you parse once and execute many times.

CAUTION *High hard parse rates lead to severe performance problems, so it's critical that you reduce hard parse counts in your database. Later sections in this chapter show you how to do this.*

A soft parse simply involves checking the library cache for an identical statement and reusing it. The major step of optimizing the SQL statement is completely omitted during a soft parse. There's really no parsing (as done during a hard parse) during a soft parse, because the new statement is hashed and its hash value is compared with the hash values of similar statements in the library cache. During a soft parse, Oracle only checks for the necessary privileges. For example, even if there's an identical statement in the library cache, your statement may not be executed if Oracle determines during the (soft) parsing stage that you don't have the necessary privileges.

Here are some very rough figures listed by Oracle as general guidelines to determine if there's excessive parsing in your database:

- If the hard parse rate is more than 100/second, it's excessive.
- If the soft parse rate is more than 300/second, you need to examine the reasons for it.

Using SQL Trace and TKPROF to Examine Parse Information

One of the most useful pieces of information the SQL Trace utility provides concerns the hard and soft parsing information for a query. The following simple example demonstrates how you can derive the parse information for any query:

1. Enable tracing in the session by using the following command:

   ```
   SQL> alter session set sql_trace=true;
   Session altered.
   SQL>
   ```

   To make sure none of your queries were parsed before, flush the shared pool, which removes all SQL statements from the library cache:

   ```
   SQL> alter system flush shared_pool;
   System altered.
   SQL>
   ```

2. Use the following query to create a trace in the user dump directory:

   ```
   SQL> select * from bsa_orgs where created_date >sysdate-5;
   ```

   The sql_trace output shows the following in the output file:

   ```
   PARSING IN CURSOR #1 len=63 dep=0 uid=21 oct=3
   lid=21 tim=1326831345 hv=71548308
   select * from bsa_orgs where created_date > sysdate-:"SYS_B_0"
   END OF STMT
   PARSE #1:c=4,e=4,p=0,cr=57,cu=3,mis=1,r=0,dep=0,og=0,tim=1326831345
   ```

   Note that *mis=1* indicates a hard parse because this SQL isn't present in the library cache.

3. Use a slightly different version of the previous query next. The output will be the same, but Oracle won't use the previously parsed version, because the statements in steps 2 and 3 aren't identical.

   ```
   SQL> select * from bsa_orgs where created_date > (sysdate -5);
   ```

   Here's the associated sql_trace output:

   ```
   PARSING IN CURSOR #1 len=77 dep=0 uid=21 oct=3 lid=21 tim=1326833972
   select /* A  Hint */ * from bsa_orgs where
    created_date > sysdate-:"SYS_B_0"
   END OF STMT
   PARSE #1:c=1,e=1,p=0,cr=0,cu=0,mis=1,r=0,dep=0,og=0,tim=1326833972
   ```

   Again, there is a hard parse, indicated by *mis=1*, showing a library cache miss. This isn't a surprise, as this statement isn't identical to the one before, so it has to be parsed from scratch.

4. Use the original query again. Now, Oracle will perform only a soft parse, because the statements here and in the first step are the same. Here's the sql_trace output:

   ```
   PARSING IN CURSOR #1 len=63 dep=0 uid=21 oct=3 lid=21 tim=1326834357
   select * from bsa_orgs where created_date > sysdate-:"SYS_B_0"
   END OF STMT
   PARSE #1:c=0,e=0,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=1326834357
   ```

The statement is step 4 is identical in all respects to the statement in step 1, so Oracle reuses the parsed version, hence *mis=0* indicating there wasn't a hard parse but merely a soft parse, which is a lot cheaper in terms of resource usage.

If you now look at the TKPROF output (see Chapter 18 for details on how to use the TKPROF utility to format sql_trace output in a meaningful format), you'll see the following section for the SQL statements in step 2 and step 4 (identical statements):

```
******************************************************
select *from national_orgs where created_date > sysdate- 5
call    count    cpu   elapsed     disk    query   current   rows
------- ------  -------- ---------- ---------- ---------- ----------
Parse     2      0.03    0.01       0          1       3       0
Execute   2      0.00    0.00       0          0       0       0
Fetch     4      0.07    0.10     156        166      24      10
total     8      0.10    0.11     156        167      27      10
Misses in library cache during parse: 1
******************************************************
```

As you can see, there was one miss in the library cache when you first executed the statement. The second time around, there was no hard parse and hence no library cache miss.

Measuring Library Cache Efficiency

You can use simple ratios to see if your library cache is sized correctly. The V$LIBRARYCACHE data dictionary view provides you all the information you need to see if the library cache is efficiently sized. Listing 19-1 shows the structure of the V$LIBRARYCACHE view.

Listing 19-1. The V$LIBRARYCACHE View

```
SQL> desc v$librarycache
 Name                                      Null?    Type
 ----------------------------------------- -------- -------------
 NAMESPACE                                          VARCHAR2(15)
 GETS                                               NUMBER
 GETHITS                                            NUMBER
 GETHITRATIO                                        NUMBER
 PINS                                               NUMBER
 PINHITS                                            NUMBER
 PINHITRATIO                                        NUMBER
 RELOADS                                            NUMBER
 INVALIDATIONS                                      NUMBER
 DLM_LOCK_REQUESTS                                  NUMBER
 DLM_PIN_REQUESTS                                   NUMBER
 DLM_PIN_RELEASES                                   NUMBER
 DLM_INVALIDATION_REQUESTS                          NUMBER
 DLM_INVALIDATIONS                                  NUMBER
SQL>
```

The following is the formula that provides you with an indicator of the library cache hit ratio:

```
SQL> Select sum(pinhits)/sum(pins)  Library_cache_hit_ratio
  2  From v$librarycache;
LIBRARY_CACHE_HIT_RATIO
-----------------------
       .993928013
SQL>
```

The formula indicates that the library cache currently has a higher than 99 percent hit ratio, which is considered good. However, be cautious about relying exclusively on high hit ratios for the library cache and the buffer caches like the one shown here. You may have a hit ratio such as 99.99 percent, but if there are significant waits caused by events such as excessive parsing (both hard and soft), you are going to have a slow database. Always keep an eye on the wait events in your system, and do not rely blindly on high hit ratios like these.

Listing 19-2 shows how to determine the number of reloads and pinhits of various statements in your library cache.

Listing 19-2. Determining the Efficiency of the Library Cache

```
SQL>  SELECT namespace,pins,pinhits,reloads
   2  FROM V$LIBRARYCACHE
   3  ORDER BY namespace;

NAMESPACE             PINS      PINHITS    RELOADS
---------            -----    ----------  ---------
BODY                    25          12          0
CLUSTER                248         239          0
INDEX                   31           0          0
JAVA DATA                6           4          0
JAVA RESOURCE            2           1          0
JAVA SOURCE              0           0          0
OBJECT                   0           0          0
PIPE                     0           0          0
SQL AREA            390039      389465         14
TABLE/PROCEDURE       3532        1992          0
TRIGGER                  5           3          0
11 rows selected.
SQL>
```

If the reloads column of the V$LIBRARYCACHE view shows large values, it means that many SQL statements are being reloaded into the library pool after they've been aged out. You might want to increase your shared pool, but this still may not do the trick if the application is large and the number of executions is large or the application doesn't use bind variables. If the SQL statements aren't exactly identical and/or they use constants instead of bind variables, more hard parses will be performed, and hard parses are inherently expensive in terms of resource usage. You can force the executable SQL statements to remain in the library cache component of the shared pool by using the Oracle DBMS_SHARED_POOL package. The package has *keep* and *unkeep* procedures, using which you can retain and release objects in the shared pool.

NOTE *The section "Pinning Objects in the Shared Pool" later in this chapter shows you how to pin objects in the shared pool. Chapter 21 explains the DBMS_SHARED_POOL package in more detail.*

You can use the V$LIBRARY_CACHE_MEMORY view to determine the number of library cache memory objects currently in use in the shared pool and to determine the number of freeable library cache memory objects in the shared pool. The V$SHARED_POOL_ADVICE view provides you information about the parse time savings you can expect for various sizes of the shared pool.

Optimizing the Library Cache

Other than mechanically increasing the size of the library cache, is there anything you can do to improve the performance of this critical component of Oracle's memory? Fortunately, the answer is yes. You can configure some important initialization parameters so the library cache areas are used efficiently. You'll look at some of these initialization parameters in the following sections.

Using the Cursor_Sharing (Literal Replacement) Parameter

The key idea behind optimizing the use of the library cache is to reuse previously parsed or executed code. One of the easiest ways to do this is to use bind variables rather than literal statements in the SQL code. *Bind variables* are like placeholders—they allow binding of application data to the SQL statement. Each time a SQL statement is presented to Oracle, it will have to be parsed by Oracle. If a parsed version of the SQL statement already exists in the shared pool, Oracle can skip the parsing phase.

Because parsing is expensive, you should make every attempt to reduce the amount of parsing your application goes through. Using bind variables enables Oracle to reuse statements when the only thing changing in the statements are the values of the input variables. Bind variables enable you to reuse the cached parsed versions of queries and thus speed up your application. Here's an example of the use of bind variables. The following code sets up a bind variable as a number type:

```
SQL> variable bindvariable number;
SQL> begin
  2  :bindvariable := 1200;
  3  end;
  4  /
PL/SQL procedure successfully completed.
SQL> select last_name from persons where person_id = :bindvariable;
```

Once you set up your bind variable in the preceding fashion, all identical SQL statements that follow will use the parsed version of the query, thus cutting down on the total time taken to retrieve data. For example, all the following statements can use the parsed version of the query. Each time, once Oracle sees the parsed statement in the shared pool, it will replace person_id with the new bind variable value.

```
select last_name from persons where person_id =  98765;
select last_name from persons where person_id = 1010101;
select last_name from persons where person_id = 9999999;
```

A better way is to use bind variables, which transform each of the previous literal statements into the following statement:

```
select last_name from persons where person_id =  :b;
```

You can execute this statement multiple times with different values for the bind variable *b*. The statement will be parsed only once and execute many times. Unfortunately, in too many applications, literal values rather than bind values are used, which leads to a heavy amount of hard parsing and consequent latch contention for shared pool areas. The application is already in implementation and there is no way to rewrite code, replacing literal values with bind variables. What are you to do? You can take the easy way out of the problem by setting up the following initialization parameter:

```
  Cursor_sharing=force
```

or

```
  Cursor_sharing=similar
```

By default, the *cursor_sharing* initialization parameter is set to *exact*, meaning that only statements that are identical in all respects will be shared among different executions of the statement. Either of the alternative values for the *cursor_sharing* parameter, *force* or *similar*, will ensure Oracle will reuse statements even if they are not identical in all respects.

For example, if two statements are identical in all respects and differ only in the value of their literal values for some variables, using *cursor sharing=force* will enable Oracle to reuse the parsed SQL statements in cache. Literal values will be replaced by bind values to make the statements identical. The *cursor_sharing=force* option will literally force the use of bind variables under all circumstances, whereas the *cursor sharing=similar* option will do so only when Oracle thinks doing so won't adversely affect optimization. Oracle recommends the use of *cursor_sharing=similar* rather than *cursor_sharing=force* because of possible deterioration in the execution plans. However, in reality, the benefits provided by the *cursor_sharing=force* parameter far outweigh any possible damage to the execution plans. You can improve the performance of your database dramatically when you notice a high degree of hard parsing due to the failure to use bind variables by moving from the default *cursor_sharing=exact* option to the *cursor_sharing=force* option. You can change the value of this parameter in the init.ora file or SPFILE, or you can do so dynamically by using the *alter system* (instancewide) statement or the *alter session* (session-level) statement.

By allowing users to share statements that differ only in the value of the constants, the *cursor_sharing* parameter enables the Oracle database to scale easily to a large number of users who are using similar, but not identical, SQL statements. This major innovation started in the Oracle8*i* version.

Sessions with a High Number of Hard Parses

The query in Listing 19-3 enables you to find out how the hard parses compare with the number of executions since the instance was started. It also tells you the session ID for the user using the SQL statements.

Listing 19-3. Determining Sessions with a High Number of Parses

```
SQL> SELECT pa.sid, pa.value "Hard Parses",
  3  ex.value "Execute   Count"
  3  FROM v$sesstat s, v$sesstat t
  4  WHERE s.sid=t.sid
  5  AND s.statistic#=(select statistic#
  6  FROM v$statname where name='parse count (hard)')
  7  AND t.statistic#=(select statistic#
  8  FROM v$statname where name='execute count')
  9* AND s.value>0;
        SID Hard Parses Execute Count
       ---- ----------- -------------
          5          12          1122
          6           1            39
         11           6           189
         12         136          5475
         15           4            69
         17           1            37
6 rows selected.
SQL>
```

Using the Cursor_Space_For_Time Parameter

By default, cursors can be deallocated even when the application cursors are not closed. This forces an increase in Oracle's overhead because of the need to check if the cursor is flushed from the library cache. The parameter that controls whether this deallocation of cursors takes place is the *cursor_space_for_time* initialization parameter, whose default value is *false*. If you set this parameter to *true*, you ensure that the cursors for the application cannot be deallocated while the application cursors are still open. The initialization parameter in the init.ora file should be as follows:

```
Cursor_space_for_time=true
```

TIP *Of course, if you want to set this parameter, make sure that you have plenty of free shared pool memory available, because this parameter will use more shared pool memory for saving the cursors in the library cache.*

Using the Session_Cached_Cursors Parameter

Ideally, an application should have all the parsed statements available in separate cursors, so that if it has to execute a new statement, all it has to do is to pick the parsed statement and change the value of the variables (if you're using bind variables). If the application reuses a single cursor with different SQL statements, it still has to pay the cost of a soft parse. After opening a cursor for the first time, Oracle will parse the statement, and then it can reuse this parsed version in the future. This is a much better strategy than re-creating the cursor each time the database executes the same SQL statement. If you can cache all the cursors, you'll retain the server-side context, even when clients close the cursors or reuse them for new SQL statements.

You'll appreciate the usefulness of the *session_cached_cursors* parameter in a situation where users repeatedly parse the same statements, as happens in an Oracle Forms-based application when users switch among various forms. Using the *session_cached_cursors* parameter will prevent the closed cursors from being cached at the session level, so any new calls to parse the same statements will avoid the parsing overhead. Using the initialization parameter *session_cached_cursors* and setting it to a high number will make the query processing more efficient. Although soft parses are cheaper than hard parses, you can reduce even soft parsing by using the *session_cached_cursors* parameter and setting it to a high number.

The perfect situation is where a SQL statement is soft parsed once in a session and executed multiple times. For a good explanation of bind variables, cursor sharing, and related issues, please read the Oracle white paper "Efficient use of bind variables, cursor_sharing, and related cursor parameters" (`http://otn.oracle.com/deploy/performance/pdf/cursor.pdf`).

Parsing and Scaling Applications

When the number of users keeps increasing, some systems have trouble coping. Performance slows down dramatically in many systems as a result of trying to scale to increased user populations. When your user counts are increasing, focus on unnecessary parsing in your system. A high amount of parsing leads to latch contention, which will slow down the system. Here are some guidelines that help summarize the previous discussion about the library cache, parsing, and the use of special initialization parameters:

- A standard rule is to put as much of the code as possible in the form of stored code—packages, procedures, and functions—so you don't have the problems caused by ad hoc SQL. Use of ad hoc SQL could wreak havoc with your library cache, and it's an inefficient way to run a large application with many users. Using stored code guarantees that code is identical and thus reused, thereby enhancing scalability.

- Lower the number of hard parses, as they could be very expensive. One way to convert a hard parse to a soft parse is to use bind variables, as you saw earlier in this chapter. Reducing hard parsing will reduce shared pool latch contention.
- If bind variables aren't being used in your system currently, you can use the *cursor_sharing=force* parameter to force the sharing of SQL statements that differ only in the value of literals.
- Pay attention to the *amount* of soft parsing, not the *per unit* cost, which is much lower than that of a hard parse. A high amount of soft parsing will increase contention for the library cache latch and could lead to a slow-performing database.
- Use the *session_cached_cursors* initialization parameter to reuse the open cursors in a session. This, as you've seen, will reduce the amount of soft parsing. Set the value of this parameter to somewhere between the value of the *open_cursors* initialization parameter and the number of cursors that are being used in the instance.
- Use the *cursor_space_for_time* initialization parameter (set it to *true*) to prevent the early deallocation of cursors. If you don't mind the extra cost of using more memory, this feature will enhance your application's scalability level.
- Reduce the amount of session logging on/off activity by the users, as this will reduce scalability due to the increased amount of parsing that results. Each time a session logs off, the SQL it uses has to be reparsed, leading to a waste of time and resources. Furthermore, the users may be spending more time trying to log into the system than actually executing their SQL statements. Frequent logging off and logging back on might also cause contention for the Web server and other resources and increase the time it takes to log into your system.

The Dictionary Cache

The dictionary cache, as mentioned earlier, caches data dictionary information. This cache is much smaller than the library cache, and to increase or decrease it you modify the shared pool accordingly. If your library cache is satisfactorily configured, chances are that the dictionary cache is going to be fine too. You can get an idea about the efficiency of the dictionary cache by using the following query:

```
SQL> select (sum(gets - getmisses - fixed)) / sum(gets)
  2* "data dictionary hit ratio" from v$rowcache;
data dictionary hit ratio
-------------------------
     .936781093
SQL>
```

Usually, it's a good idea to shoot for a dictionary hit ratio as high as 95 to 99 percent. To do so in the preceding example, you need to increase the shared pool size for the instance.

Sizing the Shared Pool

You can estimate the size of the total shared pool by trial and error. Start with a moderate amount and watch the library cache and dictionary cache hit ratios. If the hit ratios are high (99 percent and above for the library cache, and 85 to 90 percent for the dictionary cache), you *do* have enough shared pool memory in the SGA. How do you know if you have too much memory allocated to the shared pool? You can easily find out if you have too much shared pool memory by querying the V$SGASTAT view in the following manner:

```
SQL>  select bytes from v$sgastat
   2  where pool='shared pool'
   3* and name = 'free memory';
        BYTES
     ----------
     123647628
SQL>
```

If you consistently see a lot of free memory, obviously your shared pool allocation is excessive and you can reduce it. On the other hand, if the free memory is consistently hovering around 1MB to 5MB, it is an indication that the shared pool memory is too low and you need to raise it. A very low amount of free memory in the shared pool is usually seen along with heavy fragmentation of the shared pool.

Another way you can decide the minimum size of your shared pool memory is by looking at a pair of key dynamic performance views, V$DB_OBJECT_CACHE and V$SQLAREA. Chapter 21 discusses both of these views in detail. Here, let me show you how to use these views to help you size the shared pool.

The V$DB_OBJECT_CACHE view has information on all objects that are presently cached in the library cache, including tables, indexes, views, procedures, functions, packages, and triggers. An important column is sharable_mem, which is the amount of memory used by a particular object.

The V$SQLAREA view contains information about parsed SQL statements that are ready for execution. In this view, the column sharable_mem shows the total amount of shared memory used by a cursor, including all its child cursors.

Another key column is users_opening, which shows the number of users who have any child cursors open. Obviously, any amount of memory in your library cache should be enough to enable the execution of all SQL and PL/SQL statements that are currently executing in your database. Using these two dynamic performance views, you can use the query shown in Listing 19-4, which indicates what the minimum (not the ideal) amount of your shared pool ought to be.

Listing 19-4. Determining the Correct Shared Pool Size

```
SQL> select to_number (value)
  2  shared_pool_size,
  3  sum_obj_size,
  4  sum_sql_size,
  5  sum_user_size,
  6  (sum_obj_size + sum_sql_size + sum_user_size) *
  7  1.2  min_shared_pool
  8  from
  9  (select sum(sharable_mem) sum_obj_size
  10 from v$db_object_cache),
 11  (select sum(sharable_mem) sum_sql_size
 12  from v$sqlarea),
 13  (select sum(250 * users_opening) sum_user_size
 14  from v$sqlarea), v$parameter
 15* where name='shared_pool_size';
SHAR_POOL_SIZ  SUM_OBJ_SIZ  SUM_SQL_SIZ   SUM_USR_SIZ  MIN_SHAR_POOL
---------------- ------------ ------------ ---------- --------------
   900000000     328233570    306795453      26035125    793276978
1 row selected.
SQL>
```

In Listing 19-4, the minimum shared pool is calculated as the sum of the following three major areas that use the library cache:

- *Sum_obj_size* indicates the total amount of memory used by all objects cached in the library cache.
- *Sum_sql_size* is the total memory used by all the SQL cursors in the library cache.
- *Sum_user_size* is the total amount of memory in the library cache allocated to all the users who have cursors open in the library cache.

In this case, the query indicates that the database needs a minimum of about 800MB. Because the database already has 900MB, you have a little more memory in the shared pool than you need.

Pinning Objects in the Shared Pool

If code objects have to be repeatedly hard-parsed and executed, database performance will deteriorate eventually. Your goal should be to see that as much of the executed code remains in memory as possible, so compiled code can be re-executed. You can avoid repeated reloading of objects in your library cache by pinning objects using the DBMS_SHARED_POOL package. Listing 19-5 shows how you can determine the objects that should be pinned in your library cache (shared pool).

Listing 19-5. Determining the Objects to Be Pinned in the Shared Pool

```
SQL>  select
  2   type,
  3   count(*)  objects,
  4   sum(decode(kept, 'YES', 1, 0))  kept,
  5   sum(loads) - count(*)  reloads
  6   from
  7   v$db_object_cache
  8   group by
  9   type
 10   order by
 11*  objects desc;
TYPE                           OBJECTS      KEPT    RELOADS
---------------------------- ---------- ---------- ----------
NOT LOADED                       43245          0        498
CURSOR                           22482          0       3548
TABLE                             1373          0        908
PUB_SUB                            534          0        280
SYNONYM                            331          0         85
PACKAGE                            165          0        106
PACKAGE BODY                       158          0         63
TRIGGER                            125          0        128
SEQUENCE                            82          0          0
VIEW                                66          0        140
FUNCTION                            17          0         28
INDEX                               15          0          0
PROCEDURE                            8          0          1
PIPE                                 7          0          0
NON-EXISTENT                         6          0          5
CLUSTER                              5          5          0
16 rows selected.
SQL>
```

If the number of reloads in the output shown in Listing 19-5 is high, you need to make sure that the objects are pinned using the following command:

```
SQL> execute sys.dbms_shared_pool.keep(object_name, object_type);
```

You can use the following statements to first pin a package in the shared pool and then remove it:

```
SQL> execute sys.dbms_shared_pool.keep(NEW_EMP.PKG, PACKAGE);
SQL> execute sys.dbms_shared_pool.unkeep(NEW_EMP.PKG,PACKAGE);
```

Of course, if you shut down and restart your database, the shared pool won't retain the pinned objects. That's why most DBAs use scripts with all the objects they want to pin in the shared pool, which they schedule to run right after every database start. Exactly what and how many objects should you pin in your shared pool? Well, if you realize the point that most of the objects usually are very small, there's no reason to be too conservative in this regard. For example, I pin all my packages, including Oracle-supplied PL/SQL packages.

Look at the following example, which gives you an idea about the total memory taken up by a large number of packages. This query shows the total number of packages in my database:

```
SQL>  select COUNT(*)
   2  from v$db_object_cache
   3* where type='PACKAGE';
    COUNT(*)
---------------
      167
1 row selected.
```

The following query shows the total amount of memory needed to pin all my packages in the shared pool:

```
SQL>  select sum(sharable_mem)
   2  from v$db_object_cache
   3* where type='PACKAGE';
      SUM(SHARABLE_MEM)
      -----------------
          4771127
SQL>
```

As you can see, pinning every single package in my database will take up less than 5MB of a total of several hundred megabytes of memory allocated to the shared pool.

Using the Shared Pool Advisor

"Eyeballing" the free memory through the V$SGASTAT view and the collection of cache hit ratios for the shared pool is helpful, but it doesn't really tell you whether the configuration of your shared pool is optimal. Oracle provides an excellent Shared Pool Advisor as part of the OEM toolset to help you figure out the optimal amount of shared pool memory allocation. Chapters 5 and 16 explore the Shared Pool Advisor in detail.

Tuning the Buffer Cache

When users request data, Oracle reads the data from the disks (in terms of Oracle blocks) and stores it in the buffer cache so it may access the data easily if necessary. As the need for the data diminishes, eventually Oracle removes the data from the buffer cache to make room for newer data. Note that some operations do not use the buffer cache (SGA); rather, they read directly into the PGA area. Direct sort operations and parallel reads are examples of such operations.

How to Size the Buffer Cache

You use a process of trial and error to set the buffer cache size. You assign an initial amount of memory to the pool and watch the buffer cache hit ratios to see how often the application can retrieve the data from memory, as opposed to going to disk. The terminology used for calculating the buffer hit ratio can be somewhat confusing on occasion. Here are the key terms you need to understand:

- *Physical reads:* These are the data blocks that Oracle reads from disk. Reading data from disk is much more expensive than reading data that's already in Oracle's memory. When you issue a query, Oracle will always first try to retrieve the data from memory—the database buffer cache—and not disk.
- *DB block gets:* When Oracle finds the required data in the database buffer cache, it checks whether the data in the blocks is up-to-date. If a user changes the data in the buffer cache but hasn't committed those changes yet, new requests for the same data can't show these interim changes. If the data in the buffer blocks is up-to-date, each such data block retrieved is counted as a DB block get.
- *Consistent gets:* Sometimes Oracle finds the necessary data blocks in the buffer cache, but some of the data has been changed by other users. Oracle then applies rollback information from the undo segments to maintain the read consistency principle (see Chapter 8 for more information about read consistency). These data blocks are part of the consistent gets category.
- *Logical reads:* Every time Oracle is able to satisfy a request for data by reading it from the database buffer cache, you get a logical read. The logical reads include both DB block gets and consistent gets.
- *Buffer gets:* This term refers to the number of database cache buffers retrieved. This value is the same as logical reads described earlier.

Here's a query on the V$BUFFER_POOL_STATISTICS view that indicates how the physical and logical reads stack up:

```
SQL> select name,value from v$sysstat
  2  where name in ('physical reads','db block gets', 'consistent gets');
NAME                                VALUE
-------------------------------- ----------
db block gets                    31641879
consistent gets                  827618230
physical reads                   52805579
3 rows selected.
SQL>.
```

The following formula gives you the buffer cache hit ratio. Note that you need to subtract the number of *physical_reads_direct* from *physical_reads* to get a more accurate measure of total disk reads, because direct physical reads bypass the buffer cache and use the PGA instead. I am assuming there are no large objects (LOBs) in this application. If you have LOBs, you need to modify the formula slightly.

```
Buffer cache hit ratio=1–(physical reads – physical_reads_direct) /
(db block gets + consistent gets _ physical_reads_direct)
```

For example, the following calculation shows that the buffer cache hit ratio for my database is a little over 95 percent:

```
1 - (66755055-2003269) / (43026759+1338792226-2003269) = .953072149
```

As you can see from the formula for the buffer cache hit ratio, the lower the ratio of physical reads to the total logical reads, the higher the buffer cache hit ratio.

In addition, you can use the Buffer Cache Advisory, which you learned about in Chapter 5, to help you get an idea about what an optimal buffer cache size may be. To use this advisory, first set the *db_cache_advisor* parameter on. You can then examine the V$DB_CACHE_ADVICE view to see how much you need to increase the buffer cache to lower the physical I/O by a certain amount. Essentially, the output of the V$DB_CACHE_ADVICE view shows you how much you can increase your buffer cache memory before the gains in terms of a reduction in the amount of physical reads (estimated) will be insignificant. The Buffer Cache Advisory simulates the miss rates in the buffer cache for caches of different sizes, both higher and lower than the current buffer cache size. In this sense, the Buffer Cache Advisory can keep you from throwing excess memory in a vain attempt at lowering the amount of physical reads in your system.

TIP *Oracle may decide to keep only part of the large table in the buffer cache to avoid having to flush out its entire buffer cache. If your application involves many full table scans for some reason, increasing the buffer cache size isn't going to improve performance. Some DBAs are obsessed about achieving a very high cache hit ratio, such as 99 percent or so. Well, a high buffer cache hit ratio is no guarantee that your application response time and throughput will be high also. If you have a large number of full table scans or if your database is more of data warehouse rather than an OLTP system, your buffer cache may be well below 100 percent, and that's not a bad thing. If your database consists of inefficient SQL, there will be an inordinately high number of logical reads, making the buffer cache hit ratio look very good (say 99.99 percent), but this may not mean your database is performing efficiently. Please read the interesting article by Cary Millsap titled "Why a 99% Database Buffer Cache Hit Ratio Is* Not *OK" (*`http://www.hotsos.com`*).*

Using Multiple Pools for the Buffer Cache

You don't have to allocate all the buffer cache memory to a single pool. As Chapter 5 showed you, you can use three separate pools: the *keep* buffer pool, the *recycle* buffer pool, and the *default* buffer pool. Although you don't have to use the keep and default buffer pools, it's a good idea to configure all three pools so you can assign objects to them based on their access patterns. In general, you follow these rules of thumb when you use the multiple buffer pools:

- Use the recycle cache for large objects that are infrequently accessed. You don't want these objects to unnecessarily occupy a large amount of space in the default pool.

- Use the keep cache for small objects that you want in memory at all times.
- Oracle automatically uses the default pool for all objects not assigned to either the recycle or keep cache.

Since version 8.1, Oracle has used a concept called *touch count* to measure how many times an object is accessed in the buffer cache. This algorithm of using touch counts for managing the buffer cache is somewhat different from the traditional modified LRU algorithm that Oracle used to employ for managing the cache. Each time a buffer is accessed, the touch count is incremented. If you have large objects that have a low touch count but occupy a significant proportion of the buffer cache, you can consider them ideal candidates for the recycle cache. Listing 19-6 contains a query that shows you how to find out which objects have a low touch count. The touch count column is in the x$bh table owned by the user SYS.

Listing 19-6. Determining Candidates for the Recycle Buffer Pool

```
SQL>  select
   2  obj object,
   3  count(1)  buffers,
   4  (count(1)/totsize) * 100 percent_cache
   5      from x$bh,
   6           (select value totsize
   7            from v$parameter
   8   where name ='db_block_buffers')
   9   where tch=1
  10   or (tch = 0 and lru_flag <10)
  11               group by obj, totsize
  12*              having (count(1)/totsize)  *  100 > 5
     OBJECT    BUFFERS  PERCENT_CACHE
---------- ---------- ---------------
      1386      14288     5.95333333
      1412      12616     5.25666667
    613114      22459     9.35791667
```

The preceding query shows you that three objects, each with a low touch count, are taking up about 20 percent of the total buffer cache. Obviously, they are good candidates for the recycle buffer pool. In effect, what you are doing is limiting the number of buffers these three infrequently accessed tables can use up in the buffer cache.

The following query on DBA_OBJECTS gives you the names of the objects:

```
SQL> select object_name from dba_objects
  2  where object_id in (1386,1412,613114);
OBJECT_NAME
-----------------------------------------
EMPLOYEES
EMPLOYEE_HISTORY
FINANCE_RECS
SQL>
```

You can then assign these three objects to the reserved buffer cache pool. You can use a similar criterion to decide which objects should be part of your keep buffer pool. Say you want pin all objects in the keep pool that occupy at least 25 buffers and have an average touch count of more than 5. Listing 19-7 shows the query that you should run as the user SYS.

Listing 19-7. Determining Candidates for the Keep Buffer Cache

```
  SQL>  select obj object,
     2  count(1) buffers,
     3  avg(tch) average_touch_count
     4  from x$bh
     5  where lru_flag = 8
     6  group by obj
     7  having avg(tch) > 5
     8*    and count(1) > 25;
    OBJECT    BUFFERS AVERAGE_TOUCH_COUNT
---------- ---------- -------------------
   1349785         36                  67
4294967295         87           57.137931
SQL>
```

Again, querying the DBA_OBJECTS view provides you with the names of the objects that are candidates for the keep buffer cache pool.

Here's a simple example to show how you can assign objects to specific buffer caches (keep and recycle). First, make sure you configure the keep and recycle pools in your database by using the following set of initialization parameters:

```
db_cache_size=33554432
db_keep_cache_size=10000000
db_recycle_cache_size=10000000
```

In this example, the keep and recycle caches are 10MB each. The rest of the buffer cache, about 13MB, will remain the default buffer cache. Once you create the keep and recycle pools, it's easy to assign objects to these pools. All tables are originally in the default buffer cache, where all tables are cached automatically unless specified otherwise in the object creation statement.

You can use the *alter table* statement to assign any table or index to a particular type of buffer cache. For example, you can assign the following two tables to the keep and recycle buffer caches:

```
SQL> alter table test1 storage (buffer_pool keep);
Table altered.
SQL> alter table test2 storage (buffer_pool recycle);
Table altered.
SQL>
```

NOTE *For details about Oracle's touch count buffer management, please read Craig A. Shallahamer's interesting paper "All About Oracle's Touch Count Data Block Buffer Management" at* `http://www.orapub.com`. *You need to get a free membership to OraPub.com before you can download this and other papers.*

Tuning the Large Pool

You mainly use the large pool, an optional component of the SGA, for providing memory for backup and restores, and shared server processes. Oracle recommends the use of the large pool if you are using shared server processes so you can keep the shared pool fragmentation low. If you are using shared server configurations or the Recovery Manager, you should configure the large pool.

NOTE *You size the large pool based on the number of active simultaneous session in a shared server environment. Remember that if you're using the shared server configuration and you don't specify a large pool, Oracle will allocate memory to the shared sessions out of your shared pool.*

Tuning PGA Memory

Each server process is allocated a private memory area, the PGA, most of which is dedicated to memory-intensive tasks such as group by, order by, rollup, and hash joins. Operations such as in-memory sorting and building hash tables need specialized work areas. The memory you allocate to the PGA determines the size of these work areas for specialized tasks, such as sorting, and determines how fast the system can finish them. In the following sections you'll examine how you can decide on the optimal amount of PGA for your system.

Automatic PGA Memory Management

The management of the PGA memory allocation is easy from a DBA's point of view. You can set a couple of basic parameters and let Oracle automatically manage the allocation of memory to the individual work areas. You need to do a couple of things before Oracle can automatically manage the PGA: You need to use the *pga_aggregate_target* parameter to set the memory limit and you need to use the V$PGA_TARGET_ADVICE view to tune the target's value. In the next sections I discuss those tasks.

Using the Pga_Aggregate_Target Parameter

The *pga_aggregate_target* parameter in the init.ora file sets the maximum limit on the total memory allocated to the PGA. Oracle offers the following guidelines on sizing the *pga_aggregate_target* parameter:

- For an OLTP database, the target should be 16 to 20 percent of the total memory allocated to Oracle.
- For a DSS database, the target should be 40 to 70 percent of the total memory allocated to Oracle.

Using the V$PGA_TARGET_ADVICE View

Once you've set the initial allocation for the PGA memory area, you can use the V$PGA_TARGET_ADVICE view to tune the target's value. Oracle will populate this view with the results of its simulations of different workloads for various PGA target levels. You can then query the view as follows:

```
SQL>  SELECT round(PGA_TARGET_FOR_ESTIMATE/1024/1024) target_mb,
  2   ESTD_PGA_CACHE_HIT_PERCENTAGE cache_hit_perc,
  3   ESTD_OVERALLOC_COUNT
  4*  FROM v$pga_target_advice;
```

Using the estimates from the V$PGA_TARGET_ADVICE view, you can then set the optimal level for PGA memory. Chapters 5 and 16 provide you with more detailed explanations of PGA memory and how to use the V$PGA_TARGET_ADVICE view.

Evaluating System Performance

The instance tuning efforts that you undertake from within Oracle will have only a limited impact (they may even have a negative impact) if you don't pay attention to the system performance as a whole. System performance includes the CPU performance, disk I/O, and memory usage at the operating system level. In the following sections you'll look at each of these important resources in more detail.

CPU Performance

You can use operating system utilities such as sar (system activity reporter) or vmstat to find out how the CPU is performing. Don't panic if your processors seem busy during peak periods—that's what they're there for, so you can use them when necessary. If the processors are showing a heavy load during low usage times, you do need to investigate further. Listing 19-8 shows a *sar* command output indicating how hard your system is using the CPU resources right now.

Listing 19-8. Sar Command Output Showing CPU Utilization

```
[finance1] $ sar -u 10 5
HP-UX finance1  B.11.00 A 9000/800    01/28/03
13:39:17     %usr      %sys       %wio     %idle
13:39:27      34        23          7        36
13:39:37      37        17          8        38
13:39:47      34        18          6        41
13:39:57      31        16          9        44
13:40:07      38        19         11        32
Average       35        19          8        38
oracle@finance1.netbsa.org   [/u01/app/oracle/dba]
[finance1] $
```

In the preceding listing, the four columns report on the following CPU usage patterns:

- *%usr* shows the proportion of total CPU time taken up by the various users of the system.
- *%sys* shows the proportion of time the system itself was using the CPU.
- *%wio* indicates the percent of time the system was waiting for I/O.
- *%idle* is the proportion of time the CPU was idle.

If the %wio or %idle percentages are near zero during nonpeak times, it's an indication of a CPU-bound system.

Remember that a very intensive CPU usage level may mean that an operating system process is hogging CPU, or an Oracle process may be doing the damage. If it is Oracle, a background process such as PMON may be the culprit, or an Oracle user process may be running some extraordinarily bad ad hoc SQL query on the production box. You may sometimes track down such a user and inform the person that you are killing the process in the interest of the welfare of the entire system. Imagine your surprise when you find that the user's Oracle process is hale and hearty, while merrily continuing to devastate your system in the middle of a busy day. This could happen because a child process or a bequeath process continued to run even after you "killed" this user. It pays to double-check that the user is gone—lock, stock, and barrel—instead of assuming that the job has been done.

That said, let's look at some of the common events that could cause CPU-related slowdowns on your system.

The Run Queue Length

One of the main indicators of a heavily loaded CPU system is the length of the run queue. A longer run queue means that more processes are lined up, waiting for CPU processing time. Occasional blips in the run queue length are not bad, but prolonged high run queue lengths indicate that the system is CPU bound.

CPU Units Used by Processes

You can determine the number of CPU units a UNIX process is currently using by using the simple process (*ps*) command, as shown here:

```
oracle@finance1   [/u01/app/oracle/dba]
[finance1] $  ps -ef | grep f60
    UID   PID   PPID  C    STIME  TTY  TIME   CMD
  oracle 20108  4768  0  09:11:49 ?   0:28  f60webm
  oracle   883  4768  5 17:12:21   ?  0:06  f60webm
  oracle  7090  4768 16 09:18:46   ?  1:08  f60webm
  oracle 15292  4768 101 15:49:21  ?  1:53  f60webm
  oracle 18654  4768  0 14:44:23   ?  1:18  f60webm
  oracle 24316  4768  0 15:53:33   ?  0:52  f60webm
```

The key column to watch is the fourth one from the left, which indicates the CPU units of processing each process is using. If each CPU on a server has 100 units, the Oracle process with PID 15292 (the fourth in the preceding list) is occupying more than an entire CPU's processing power. If you have only two processors altogether, you should worry about this process and why it is so CPU intensive.

Finding High CPU Users

If the CPU usage levels are high, you need to find out which of your users are among the top CPU consumers. Listing 19-9 shows how you can easily identify those users.

Listing 19-9. Identifying High CPU Users

```
SQL>  select n.username,
  2   s.sid,
  3   s.value
  4   from v$sesstat s,v$statname t, v$session n
  5   where s.statistic# = t.statistic#
  6   and n.sid = s.sid
  7   and t.name='CPU used by this session'
  8   ORDER BY s.value desc;
USERNAME           SID      VALUE
--------------------------- ----------
JOHLMAN            152      20745
NROBERTS           103       4944
JOHLMAN            167       4330
LROLLINS            87       3699
JENGMAN            130       3694
JPATEL              66       3344
NALAPATI            73       3286
SQL>
```

Listing 19-9 shows that CPU usage is not uniformly spread across the users. You need to investigate why one user is using such a significant amount of resources. If you need to, you can control CPU usage by a single user or a group of users by using the Database Resource Manager, as explained in Chapter 11. You can also find out session-level CPU usage information by using the V$SESSION_STAT view, as shown in Listing 19-10.

Listing 19-10. Determining Session-Level CPU Usage

```
SQL> select sid, s.value "Total CPU Used by this Session"
  2  from v$sesstat s
  3  where s.statistic# = 12
  4* order by s.value desc;
      SID Total CPU Used by this Session
     ----- ------------------------------
      496                          27623
      542                          21325
      111                          20814
      731                          17089
      424                          15228
SQL>
```

Using OEM to Track CPU Usage

Instead of running SQL scripts, you can simply use OEM to analyze CPU usage in your database. Select Tools ➤ Diagnostic Pack ➤ Performance Overview in the OEM console. You can see all the active sessions ordered by physical reads, logical reads, and total CPU time used. You can also review CPU usage through OEM's TopSessions, as shown in Chapter 17.

What Is the CPU Time Used For?

It would be a mistake to treat all CPU time as equal. CPU time is generally understood as the processor time taken to perform various tasks, such as the following:

- Loading SQL statements into the library cache
- Searching the shared pool for parsed versions of SQL statements
- Parsing the SQL statements
- Querying the data dictionary
- Reading data from the buffer cache
- Traversing index trees to fetch index keys

The total CPU time used by an instance (ora session) can be viewed as the sum of the following components:

```
Total CPU Time = Parsing CPU usage + Recursive CPU usage + Other CPU usage
```

Ideally, your total CPU usage numbers should show a very small proportion of the first two categories of CPU usage—parsing and recursive CPU usage. For example, for a sessionwide estimate of CPU usage, you can run the query shown in Listing 19-11.

Listing 19-11. Decomposition of Total CPU Usage

```
  SQL>  select name,value from v$sysstat
     2  where name in ('CPU used by this session',
     3                 'recursive cpu usage',
     4*                'parse time cpu')
  NAME                                         VALUE
---------------------------------------------- ----------
recursive cpu usage                              4713085
CPU used by this session                        98196187
parse time cpu                                    132947
3 rows selected.
SQL>
```

In this example, the sum of recursive CPU usage and parse time CPU usage is a small proportion of total CPU usage. You need to be concerned if the parsing or recursive CPU usage is a significant part of total CPU usage. Let's see how you can go about reducing the CPU usage attributable to these various components.

NOTE *In the following examples, you can examine CPU usage at the instance level by using the V$SYSSTAT view or at an individual session level by using the V$SESSTAT view. Just remember that the column "total CPU used by this session" in the V$SYSSTAT view refers to the* sum *of the CPU used by all the sessions combined.*

Parse CPU Usage

As you learned at the beginning of this chapter, parsing is an expensive operation that you should reduce to a minimum.

In the following example, the parse time CPU usage is quite low as a percentage of total CPU usage. The first query tells you that the total CPU usage in your instance is 49159124:

```
SQL> select name, value from v$sysstat
 2*  where name like '%CPU%';
NAME                                   VALUE
-------------------------------------------------
CPU used when call started           13220745
CPU used by this session             49159124
2 rows selected.
SQL>
```

The next query shows the parse time CPU usage at 96431, which is an insignificant proportion of total CPU usage in your database:

```
SQL> select name, value from v$sysstat
  2  where name like '%parse%';
NAME                                              VALUE
parse time cpu                                     96431
parse time elapsed                                295451
parse count (total)                              3147900
parse count (hard)                                 29139
4 rows selected.
SQL>
```

Listing 19-12 shows an example of a session whose CPU usage is predominantly due to a high amount of parse time CPU usage.

Listing 19-12. Determining Parse Time CPU Usage

```
SQL  select  a.value  " Tot_CPU_Used_This_Session",
  2  b.value "Total_Parse_Count",
  3  c.value "Hard_Parse_Count",
  4  d.value "parse_time_cpu"
  5  from v$sysstat a,
  6  v$sysstat b,
  7  v$sysstat c,
  8  v$sysstat d
  9  where a.name = 'CPU used by this session'
 10  and b.name = 'parse count (total)'
 11  and c.name = 'parse count (hard)'
 12* and d.name = 'parse time cpu';
Tot_CPU_Used  Total_Parse_Count  Hard_Parse_Count  Parse_Time_CPU
This_Session
------------------------------ ----------------- ----------------
2240                    53286              281            1486
SQL>
```

Parse time CPU in the preceding example is fully two-thirds of the total CPU usage. Obviously, you need to be concerned about the high rates of parsing, even though most of the parses are soft parses. The next section shows you what you can do to reduce the amount of parsing in your database.

Reducing Parse CPU Usage

If parse time CPU is the major part of total CPU usage, you need to reduce this by performing the following steps:

1. Use bind variables and remove hard-coded literal values from code, as explained in the “Optimizing the Library Cache” section earlier in this chapter.

2. Make sure you aren't allocating *too much memory* for the shared pool. Remember that even if you have an exact copy of a new SQL statement in your library cache, Oracle has to find it by scanning all the statements in the cache. If you have a zillion relatively useless statements sitting in the cache, all they're doing is slowing down the instance by increasing the parse time.

3. Make sure you don't have latch contention on the library cache, which could result in increased parse time CPU usage.

4. If your TKPROF output or one of the queries shown previously indicates that total parse CPU time is as high as 90 percent or more, check to make sure all the tables in the queries have been analyzed recently. If you don't have statistics on some of the tables, the parsing process generates the statistics, but the parse CPU usage time goes up dramatically.

Recursive CPU Usage

Recursive CPU usage is mostly for data dictionary lookups and for executing PL/SQL programs. Thus, if your application uses a high number of packages and procedures, you'll see a significant amount of recursive CPU usage.

In the following example, there's no need for alarm, because the percentage of recursive CPU usage is only about 5 percent of total CPU usage.

```
SQL>  select name,value from v$sysstat
   2  where name in ('CPU used by this session',
   3*                'recursive cpu usage');
NAME                                               VALUE
--------------------------------------------- ----------
recursive cpu usage                              4286925
CPU used by this session                        84219625
2 rows selected.
SQL>
```

If the recursive CPU usage percentage is a large proportion of total CPU usage, you may want to make sure the shared pool memory allocation is inadequate. However, a PL/SQL-based application will always have a significant amount of recursive CPU usage.

NOTE *A very high number of recursive SQL statements indicates that Oracle is busy with space management activities such as allocating extents. This has a detrimental effect on performance and you can avoid this problem by increasing the extent sizes for your database objects. Of course, this is another good reason to choose locally managed tablespaces, which really cut down on the number of recursive SQL statements.*

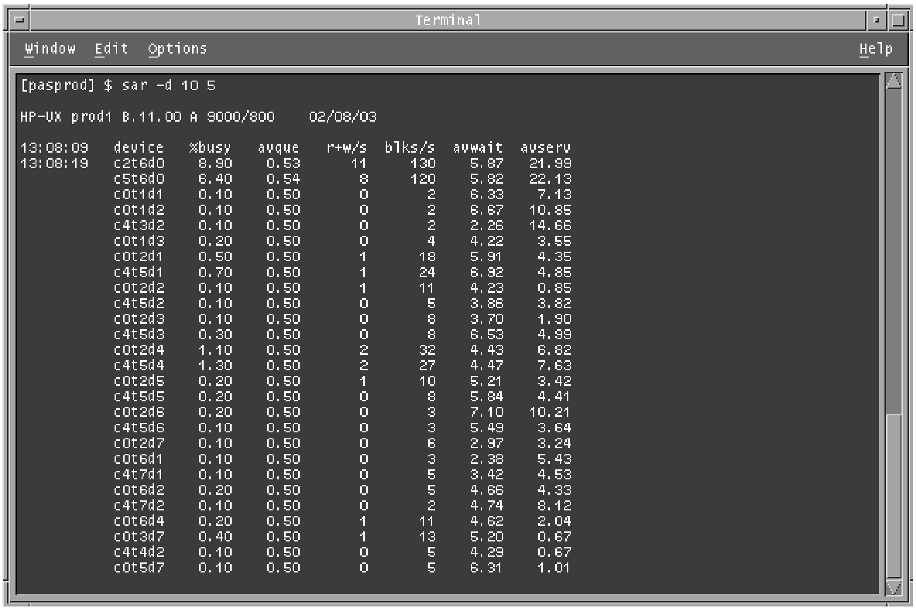
Disk I/O

The way you configure your disk system has a profound impact on your I/O rates. You have to address several issues when you are planning your disk system. Important factors that have a bearing on your I/O are as follows:

- *Choice of RAID configuration:* Chapter 3 covered RAID systems configuration in detail. Just remember that a RAID 5 configuration doesn't give you ideal I/O performance if your application involves a large number of writes. For faster performance, use a RAID 1 configuration, in which you mirror all the disks.
- *Raw devices or operating system file systems:* Under some circumstances, you can benefit by using raw devices, which bypass the operating system buffer cache. Raw devices have their own drawbacks, including limited backup features, and you want to be sure the benefits outweigh the drawbacks. Raw devices in general provide faster I/O capabilities and give better performance for a write-intensive application. You might also want to consider alternative file systems such as Veritas VxFsS, which helps large I/O operations through its direct I/O option.
- *I/O size:* I/O size is in terms of the Oracle block size. The minimum size of I/O depends on your block size, and the maximum size depends on the *multi_block_read_count* initialization parameter. If your application is OLTP based, the I/O size needs to be small, and if your application is oriented toward a DSS, the I/O size needs to be much larger.
- *Logical volume stripe sizes:* Stripe size (or stripe width) is a function of the stripe depth and the number of drives in the striped set. If you stripe across multiple disks, your database's I/O performance will be higher and its load balancing will be enhanced. Make sure that the stripe size is larger than the average I/O request; otherwise, you'll be making multiple I/Os for a single I/O request by Oracle. If you have multiple concurrent I/O requests, your stripe size should be much larger than the I/O size. Most modern LVMs can dynamically reconfigure the stripe size.
- *Number of controllers and disks:* The number of spindles and the number of controllers are both important variables in determining disk performance. Even if you have a large number of spindles, you could conceivably run into contention at the controller level.
- *Distribution of I/O:* Your goal should be to avoid a lopsided distribution of I/O in your disk system. If you are using an LVM or using striping at the hardware level, you don't have a whole lot to worry about in this area. If you aren't using an LVM or using striping at the hardware level, however, you should manually arrange your data files on the disks such that the I/O rate is fairly even across the system. Note that your tables and indexes are usually required to be in different tablespaces, but there is no rule that they can't be placed on different disks. Because the index is read before the table, they can coexist on the same disk.

Measuring I/O Performance

You have a choice of several excellent tools to measure I/O performance. The operating system utility is easy to use and gives you information about how busy your disks are. Iostat and sar are two of the popular operating system utilities that measure disk performance. Figure 19-1 shows the partial output of a typical *sar* command.



```
[pasprod] $ sar -d 10 5

HP-UX prod1 B.11.00 A 9000/800    02/08/03

13:08:09   device   %busy   avque   r+w/s  blks/s  avwait  avserv
13:08:19   c2t6d0    8.90    0.53      11     130    5.87   21.99
           c5t6d0    6.40    0.54       8     120    5.82   22.13
           c0t1d1    0.10    0.50       0       2    6.33    7.13
           c0t1d2    0.10    0.50       0       2    6.67   10.85
           c4t3d2    0.10    0.50       0       2    2.26   14.66
           c0t1d3    0.20    0.50       0       4    4.22    3.55
           c0t2d1    0.50    0.50       1      18    5.91    4.35
           c4t5d1    0.70    0.50       1      24    6.92    4.85
           c0t2d2    0.10    0.50       1      11    4.23    0.85
           c4t5d2    0.10    0.50       0       5    3.86    3.82
           c0t2d3    0.10    0.50       0       8    3.70    1.90
           c4t5d3    0.30    0.50       0       8    6.53    4.99
           c0t2d4    1.10    0.50       2      32    4.43    6.82
           c4t5d4    1.30    0.50       2      27    4.47    7.63
           c0t2d5    0.20    0.50       1      10    5.21    3.42
           c4t5d5    0.20    0.50       0       8    5.84    4.41
           c0t2d6    0.20    0.50       0       3    7.10   10.21
           c4t5d6    0.10    0.50       0       3    5.49    3.64
           c0t2d7    0.10    0.50       0       6    2.97    3.24
           c0t6d1    0.10    0.50       0       3    2.38    5.43
           c4t7d1    0.10    0.50       0       5    3.42    4.53
           c0t6d2    0.20    0.50       0       5    4.66    4.33
           c4t7d2    0.10    0.50       0       2    4.74    8.12
           c0t6d4    0.20    0.50       1      11    4.62    2.04
           c0t3d7    0.40    0.50       1      13    5.20    0.67
           c4t4d2    0.10    0.50       0       5    4.29    0.67
           c0t5d7    0.10    0.50       0       5    6.31    1.01
```

Figure 19-1. Output of the sar command showing disk usage statistics

Figure 19-1 shows the execution of the *sar* command five times at intervals of 10 seconds, and the columns have the following meaning:

- *Device* is the name of the physical device.
- *%busy* is the percentage of time the device was busy.
- *Avque* is the average number of wait requests for a device.
- *R+w/s* is the total number of reads and writes.
- *Blks/s* is the number of blocks.
- *Avwait* is the average wait time for the device.
- *Avserv* is the average service time for this device.

Is the I/O Optimally Distributed?

From the *sar* output, you can figure out if you're using the storage subsystem heavily. If the number of waits is higher than the number of CPUs, or if the service times are high (say, greater than 20ms), then your system is facing contention at the I/O level. One of the most useful pieces of information from using the *sar –d* command is finding out if you are using any of your disks excessively compared to other disks in the system. Once you identify such hot spots, you can move the data files to less busy drives, thereby spreading the load more evenly.

The following is the output of a *sar –d* command that shows extremely high queue values. Even at peak levels, the avque column value should be less than 2. Here, it is 61.4. Obviously, something is happening on the file system c2t6d0 that is showing up as a high queue value.

```
[finance1 $ sar -d 10 5
HP-UX finance1 B.11.00 A 9000/800    02/02/03
11:27:13   device   %busy    avque   r+w/s  blks/s  avwait  avserv
11:27:23   c2t6d0    100     61.40    37     245     4.71   10.43
           c5t6d0    20.38    0.50    28     208     4.92    9.54
           c2t6d0    100     61.40    38     273     4.55    9.49
           c5t6d0    18.28    0.50    27     233     4.46    7.93
           c0t1d0    0.10     0.50     4      33     4.99    0.81
$
```

The Statspack output has a section on I/O distribution. If you want an ad hoc picture of the same, you can obtain it by using the query in Listing 19-13.

Listing 19-13. Determining I/O Distribution in the Database

```
SQL> select d.name,
  2  f.phyrds reads,
  3  f.phywrts wrts,
  4  (f.readtim/ decode(f.phyrds,0,-1,f.phyrds)) readtime,
  5  (f.writetim / decode(f.phywrts,0,-1,phywrts)) writetime
  6  from
  7  v$datafile d,
  8  v$filestat f
  9  where
 10  d.file# = f.file#
 11  order by
 12* d.name;
NAME                          READS     WRTS   READTIME   WRITETIME
-----------------------------------------------------------------
/pa01/oradata/pa/lol_i_17.dbf  23      9    .608695652    .222222222

/pa01/oradata/pa/lol_i_18.dbf  18      7    .277777778             0

/pa02/oradata/pa/pli_i_17.dbf  121816  9    .234066133    .111111111

/pa02/oradata/pa/ptr_d_02.dbf  27      5             0             0
```

```
/pa04/oradata/pa/m_01_01.dbf   355689  5    .108594868            0

/pa10/oradata/pa/pe_d_09.dbf   800759  5     .204540942           0
SQL>
```

CAUTION *Excessive reads and writes on some disks indicate that there might be disk contention in your I/O system.*

Reducing Disk Contention

If there is sever I/O contention in your system, you can undertake some of the following steps, depending on your present database configuration:

- Increase the number of disks in the storage system.
- Separate the database and the redo log files.
- For a large table, use partitions to reduce I/O.
- Stripe the data either manually or by using a RAID disk-striping system.
- Invest in cutting-edge technology, such as file caching, to avoid I/O bottlenecks.

The Oracle SAME Guidelines for Optimal Disk Usage

Oracle recently published the *Stripe and Mirror Everything* (SAME) guidelines for optimal disk usage. This methodology advocates striping all files across all disks and mirroring all data to achieve a simple, efficient, and highly available disk configuration. Striping across all the available disks aims to spread the load evenly and avoid hot spots. The SAME methodology also recommends placing frequently accessed data on the outer half of the disks. The goal of the SAME disk storage strategy is to eliminate I/O hot spots and maximize I/O bandwidth. For more details about the SAME guidelines, please refer to the article "Optimal Storage Configuration Made Easy," by Juan Loaiza, which is available at `http://otn.oracle.com/deploy/availability/pdf/OOW2000_same_ppt.pdf`.

Collecting Instance Performance Statistics with Statspack

You can collect instance performance information in a couple of ways. You can use Oracle's excellent Statspack tool to capture and store performance information. The Statspack tool lets you create quick instance performance reports, which enable you to figure out what events are causing waits in the system, for example.

You can also directly query important dynamic performance views to see how the instance is performing. You can use OEM's Diagnostics Pack to quickly get summary information about current instance performance.

No matter which of the two methods you use, the dynamic performance views underlie the computation of the statistics. In this section, you'll look at the use of the Statspack tool to collect and report performance statistics. In this way, you'll get a chance to understand exactly what factors contribute to database performance at a very detailed level. Statspack also provides a way for you to maintain baseline information and compare current database performance statistics with the baseline numbers to see what's happening within the database.

Statspack is essentially a set of SQL scripts provided by Oracle to help gather and store performance statistics. When you're trying to determine which of your SQL statements is using the most resources, the Statspack data is very useful, with its precalculated cache hit ratios and a number of other statistics. Statspack collects instance performance data between two periods (called *snapshots*), and based on the time interval it computes a number of timed performance statistics.

Installing Statspack

To start using Statspack, you must first create a new user named sqlplus sysqlplus, who will own all the statistics tables. To create the perfstat user and install Statspack, you'll first connect as SYS and run the installation scripts from the $ORACLE/HOME/rdbms/admin directory. To install Statspack, you need to run three scripts: spcuser.sql, spctab.sql, and spcpkg.sql. You can call of them from within one script, spcreate.sql, but I have found that if you use the three scripts, Oracle is less prone to erroring out. The three scripts perform the following tasks:

- Spcuser.sql creates the perfstat user.
- Spctab.sql creates the necessary tables and other objects to store the statistics.
- Spcpkg.sql creates the Statspack package.

NOTE *Although the Oracle manuals don't make this obvious, you need to run the three scripts under different schemas. You must run the first script, spcusr.sql, under the SYS schema, and you must run the next two as the Statspack user perfstat.*

Listing 19-14 shows the sequence of the Statspack creation steps.

Listing 19-14. Installing Statspack

```
SQL> connect sys/mark1@mark1 as sysdba
Connected.
SQL> @%ORACLE_HOME%\rdbms\admin\spcreate
... Installing Required Packages
```

```
Package created.
... Creating PERFSTAT user...
Choose the PERFSTAT user's password.
Not specifying a password will result in the installation FAILING
Specify PERFSTAT password
Enter value for perfstat_password: perfstat1
Specify PERFSTAT user's temporary tablespace.
Enter value for temporary_tablespace: TEMP
Using TEMP for the temporary tablespace
PL/SQL procedure successfully completed.
User altered.
NOTE:
SPCUSR complete. Please check spcusr.lis for any errors.
SQL> @?/rdbms/admin/spctab
SQL> Rem $Header: spctab.sql 16-feb-2003.14:54:46 vbarrier Exp $
SQL> Rem spctab.sql
If this script is automatically called from spcreate (which is
the supported method), all STATSPACK segments will be created
in the PERFSTAT user's default tablespace.
Using EXAMPLE tablespace to store Statspack objects
... Creating STATS$SNAPSHOT_ID Sequence
Sequence created.
Synonym created.
... Creating STATS$... tables
Table created.
Synonym created.
…
Synonym created.
NOTE:
SPCTAB complete. Please check spctab.lis for any errors.
SQL>SQL> @?/rdbms/admin/spcpkg
SQL> Rem $Header: spcpkg.sql 17-feb-2003.16:59:10 vbarrier Exp $
Creating Package STATSPACK...
Package created.
No errors.
Creating Package Body STATSPACK...
Package body created.
No errors.
NOTE:
SPCPKG complete. Please check spcpkg.lis for any errors.
SQL>
```

NOTE *If you run into problems during the installation of Statspack, simply use the spdrop.sql script to drop the perfuser schema and start over.*

Using Statspack

The key to using Statspack is taking a snapshot of the instance at a given time, which involves gathering vital instance statistics and storing them in the Statspack tables. This will serve as your baseline for comparison with later snapshots. This snapshot is compared with the snapshot taken after a specified time elapses, and the changes in the instance are evaluated with reference to the time elapsed. How long should the all-important snapshot interval be? Well, there's no point in collecting statistics all day long if your key "spikes" in system activity last for only a few minutes. Just turn on statistics collection for short intervals—say 15 to 30 minutes.

You need to have the initialization parameter *timed_statistics* turned on so Statspack can capture useful information. You can ensure that the *timed_statistics* parameter is turned on in any of the following ways:

- *Timed_statistics=true* (in the init.ora file)
- *Statistics_level=typical* (in the init.ora file)
- *Statistics_level=all* (in the init.ora file)
- *Alter session set timed_statistics=true* (at the session level)

Note that the *statistics_level=typical* is the default level, and it provides you with a broad array of statistics with the least amount of overhead. The *statistics_level=full* setting gives you more details, but it's much more expensive to run in terms of a performance hit. By querying the V$STATISTICS_LEVEL view, you can get detailed information about all the statistics that are being currently captured in your instance. Before you actually take the snapshots, which will collect performance data, you need to decide on two things: the level at which you want to collect the statistics and the threshold for the SQL statements. You'll look at each of these issues in the following sections.

Snapshot Levels and SQL Thresholds

Oracle gives you a lot of flexibility in choosing the level at which you want Statspack to gather statistics. You need to specify a higher level for progressively more detailed performance data. You can summarize the snapshot levels as follows.

Level 0 collects the following general performance information:

- Wait statistics
- System statistics
- System events
- Session events
- Lock and latch statistics
- Rollback segment data
- SGA, buffer pool, and row cache statistics

Level 5 collects all the statistics gathered in the lower levels, plus SQL statement statistics that exceed a high resource usage threshold. The SQL statement statistics include parse calls, disk reads, buffer reads, and the number of executions. You can specify thresholds at the time you take the snapshots. The snapshot examples presented later on in this chapter show you how to set thresholds.

Level 6 collects all the statistics gathered in the lower levels, plus the execution plans for SQL statements, which will help you determine if the plans changed over time.

Level 7 statistics help you optimize the physical layout of your disks by providing information on segment-level access and contention. You get all the statistics from the lower levels, plus the following statistics:

- Top five segments by logical reads
- Top five segments by physical reads
- Top five segments by buffer busy waits

Levels 10 and above are for gathering specialized latch information, and you should set them only upon the recommendation of Oracle support personnel.

NOTE *The higher the snapshot level you choose, the more data Statspack collects. Level 5 is the default level. At present, the only valid levels are levels 0, 5, 6, 7, and 10. Please review the Oracle document spdoc.txt, which is in the $ORACLE_HOME/rdbms/admin directory on your server, for detailed information about the various snapshot levels and other information about the Statspack utility.*

Collecting Statspack Data

The first step in collecting data is to capture the initial snapshot of the instance, which you can do in several ways. You can do it manually, you can use the DBMS_JOB package, or you can use the crontab (in Windows, the *at* facility) to automate statistics collection. You need to take at least one other snapshot later so you can make comparisons and derive performance statistics. In the following sections you'll look at examples of the three methods.

TIP *Statspack depends on the V$ dynamic performance tables for its data. This means that if you restart the database in between two snapshots, the results will be meaningless. Each time the database is shut down, the data in the dynamic performance tables is completely lost.*

Method 1: Collecting Statistics Manually

You can start the statistics collection process by executing the *statspack.snap* procedure (as the perfstat user) both to start and stop data collection, as shown here:

```
SQL> show user
USER is "PERFSTAT"
SQL> execute statspack.snap;
PL/SQL procedure successfully completed.
SQL>
```

To get the snap_id of the snapshot you just captured, you need to execute the following procedure:

```
SQL> variable snap_num number;
SQL> begin
  2  :snap_num := statspack.snap;
  3  end;
  4  /
PL/SQL procedure successfully completed.
SQL> print snap_num
     SNAP_NUM
    ----------
        2
SQL>
```

After you let some time elapse (how much time depends on how long you want to capture statistics for), repeat the two steps:

```
SQL> begin
  2  :snap_num := statspack.snap;
  3  end;
  4  /
PL/SQL procedure successfully completed.
SQL> print snap_num
     SNAP_NUM
   ----------
        4
SQL>
```

Method 2: Using the DBMS_JOB Package to Automate Statspack

When you create the perfstat user, the DBMS_JOB package is created for scheduling jobs through Oracle. Chapter 21 presents a detailed explanation of the DBMS_JOB package, but the scheduling is really a simple matter, as you can see in Listing 19-15.

Listing 19-15. Automating Statspack Execution

```
SQL>  declare
  2   job_number integer;
  3   begin
  4   dbms_job.submit(
  5   job_number,
  6   'statspack.snap',
  7   sysdate + (1/48),
  8   'sysdate + (1/48)',
  9   true);
 10*  end;
SQL> /
PL/SQL procedure successfully completed.
SQL>
```

The job to run the *statspack.snap* procedure is scheduled to run every half hour in this example.

Method 3: Using the Crontab or the At Command to Schedule Statistics Collection

You can just use the crontab or the Windows *at* command to schedule the *snapshot.snap* procedure periodically. Chapter 3 shows you how to use the *crontab* (UNIX and Linux) and *at* (Windows) commands to schedule database jobs.

Deleting Statspack Data

Oracle provides an easy-to-use script, sppurge.sql (which is located in the $ORACLE_HOME/rdbms/admin directory), to remove unnecessary Statspack data. Once you execute this script, it asks you for an upper and lower bound of snapshot IDs, and it removes all the data that belongs to the snapshot IDs within. There is an even faster way to get rid of all Statspack data that you have collected: Simply execute the sptrunc.sql script (also located in the $ORACLE_HOME/rdbms/admin directory). Remember to log in as the user perfstat when you want to purge or truncate data using the Oracle-provided scripts.

Obtaining Statspack Reports

All you need to generate the all-important Statspack report are a pair of snapshot IDs, which will serve as the beginning and ending values for the time over which you want the reporting to be done. If you have taken many snapshots, you can have Statspack report over any specific period of time you wish.

NOTE *All wait events are shown in microseconds, rather than 10 milliseconds, as in the previous Oracle versions.*

You can run two types of Statspack reports: a general database health report called *spreport.sql* and a more specific report on single SQL statements. Let's see how you can run obtain a simple database health report from the two Statspack snapshots you gathered in the previous section. In the following output, note that you have to provide the *begin* and *end* snapshot IDs that you have created through using the *snapshot.snap* procedure (see the earlier section "Method 1: Collecting Statistics Manually"). Listing 19-16 shows the report.

Listing 19-16. The Statspack Report

```
SQL> sho user
USER is "PERFSTAT"
SQL> @?/rdbms/admin/spreport
Current Instance
~~~~~~~~~~~~~~~~
DB Id         DB Name   Inst Num   Instance
----------- ------------ -------- ------------
 1672169339    MANAGER      1       manager
Instances in this Statspack schema
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   DB Id    Inst Num    DB Name     Instance      Host
----------- -------- ------------ ------------ ------------
  1672169339     1        MANAGER     manager     ALAPATISAM
Using 1672169339 for database Id
Using          1 for instance number
Completed Snapshots
                           Snap                    Snap
Instance     DB Name         Id   Snap Started    Level Comment
------------ ------------ ----- ----------------- ----- --------
manager      MANAGER          1 19 Jan 2003 13:44     5
                              2 19 Jan 2003 14:55     5
                              3 19 Jan 2003 15:12     5
                              4 19 Jan 2003 15:13     5
Specify the Begin and End Snapshot Ids
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Enter value for begin_snap: 2
Begin Snapshot Id specified: 2
Enter value for end_snap: 4
End   Snapshot Id specified: 4
Specify the Report Name
~~~~~~~~~~~~~~~~~~~~~~~
The default report file name is sp_2_4.  To use this name,
press <return> to continue, otherwise enter an alternative.
Enter value for report_name: statspack.rep1_Jan192003
Using the report name statspack.rep1_Jan192003
STATSPACK report for
DB Name    DB Id    Instance  Inst Num   Release  Cluster   Host
-------- ----------- --------- -------- --------- ------- ----------
MANAGER  1672169339   manager     1     9.2.0.1.0   NO    ALAPATISAM
```

```
            Snap Id     Snap Time      Sessions Curs/Sess Comment
            ------- ------------------ -------- --------- ----------
Begin Snap:       2 19-Jan-03 14:55:40       21       5.4
  End Snap:       4 19-Jan-03 15:13:58       22       5.3
   Elapsed:               18.30 (mins)
Cache Sizes (end)
~~~~~~~~~~~~~~~~~
Buffer Cache:                     32M      Std Block Size:       4K
Shared Pool Size:                136M      Log Buffer:         512K
```

The Load Profile section (see Listing 19-17) gives you an idea about the logical and physical reads and writes per second and per transaction. The per-second load figures give you an indication of the throughput ("Is the database performing more work per second?") of the instance. The per-transaction figures tell you how the application transaction characteristics are changing over time. One of the items you need to watch carefully is the hard parse rate per second. If the hard parse rate is over 100 per second, the system is going to slow down noticeably because of the increased need for shared pool and library cache latches.

Listing 19-17. The Load Profile Section of the Statspack Report

```
                                 Per Second      Per Transaction
                               ------------      ----------------
                Redo size:           860.77              2,213.41
            Logical reads:            18.36                 47.21
            Block changes:             2.94                  7.56
           Physical reads:             0.19                  0.48
          Physical writes:             0.11                  0.29
               User calls:             6.17                 15.88
                   Parses:             1.44                  3.71
              Hard parses:             0.02                  0.04
                    Sorts:             2.18                  5.62
                   Logons:             0.00                  0.00
                 Executes:             1.73                  4.44
             Transactions:             0.39
  % Blocks changed per Read:   16.01    Recursive Call %:    45.65
 Rollback per transaction %:   40.98       Rows per Sort:     7.31
```

The Instance Efficiency section of the Statspack report (see Listing 19-18) indicates how good the shared pool and buffer cache hit ratios are.

Listing 19-18. The Instance Efficiency Section of the Statspack Report

```
            Buffer Nowait %:   99.99       Redo NoWait %:  100.00
            Buffer  Hit   %:   98.98    In-memory Sort %:  100.00
            Library Hit   %:   98.70        Soft Parse %:   98.86
         Execute to Parse %:   16.51         Latch Hit %:  100.00
Parse CPU to Parse Elapsd %:   10.98      % Non-Parse CPU:   99.90
 Shared Pool Statistics        Begin   End
                               ------  ------
```

```
            Memory Usage %:   21.15   21.53
   % SQL with executions>1:   49.16   49.31
 % Memory for SQL w/exec>1:   49.66   51.08
```

The Top 5 Timed Events section gives you the top five (timed) wait events and the CPU usage (if it ranks in the top five events) during the period the statistics collection was going on. These top five events may include the CPU usage time, but the CPU time only indicates CPU usage, *not* a CPU wait event. The idle wait events are omitted from the list. Note that the low hard parse rates (shown in the Load Profile section) mean that there will not be a significant wait event for latches in this section. Listing 19-19 shows the Top 5 Timed Events list.

Listing 19-19. The Top 5 Timed Events

```
Top 5 Timed Events
                                                         % Total
Event                              Waits      Time (s)    Ela Time
-----------------------------  ------------- ------------ ----------
control file sequential read        162            2         26.58
db file sequential read             132            2         21.13
log file sync                       254            2         16.73
control file parallel write         356            1         13.74
db file parallel write               42            1          7.89
```

In the example here, the Top 5 Timed Events section shows only wait events. If you see a Top 5 Timed Events section such as the one shown in Listing 19-20, you'll notice that about two-thirds of the total elapsed time is due to CPU usage, not wait events. Thus, you don't have to query the V$SYSSTAT view to determine if high CPU usage or high wait events are responsible for delays in your system.

Listing 19-20. A Different Set of Top 5 Timed Events

```
Top 5 Timed Events
                                                 % Total
Event                        Waits     Time (s)    Ela Time
---------------------  ------------- ---------    --------
CPU time                                 4         66.78
log file parallel write       83         1         14.87
control file parallel write  246         1         12.93
db file sequential read       65         0          4.06
db file scattered read         4         0           .44
```

The next two sections in the Statspack report show you all the wait events for the instance, including the background wait events. Listing 19-21 shows these wait events.

Listing 19-21. All Wait Events in the Database

```
Wait Events
-> s  - second
-> cs - centisecond -     100th of a second
-> ms - millisecond -    1000th of a second
-> us - microsecond - 1000000th of a second
-> ordered by wait time desc, waits desc (idle events last)
                                                    Avg
                                           Tot Wait wait  Waits
Event                         Waits  Timouts  Time  (ms) /txn
----------------------------  ------------ ---------- ----------
log file parallel write       783      0       1     1   195.8
control file parallel write   246      0       1     3    61.5
db file sequential read        65      0       0     4    16.3
db file scattered read          4      0       0     7     1.0
control file sequential read  228      0       0     0    57.0
…
Background Wait Events
-> ordered by wait time desc, waits desc (idle events last)
                                                      Avg
                                          Total Wait  wait    Waits
Event                       Waits  Timeouts  Time (s) (ms)   /txn
---------------------------- ------------ ---------- ---------- ---
log file parallel write      783      0         1      1    195.8
control file parallel write  246      0         1      3     61.5
db file sequential read       42      0         0      3     10.5
control file sequential read 100      0         0      0     25.0
```

The next section in the report gives you all the SQL statements that were executed in the database during the statistics collection interval. The SQL statements are ordered by the number of buffer gets. Listing 19-22 shows the SQL statements in the report.

Listing 19-22. SQL Statements Ordered by Buffer Gets

```
SQL ordered by Gets
-> End Buffer Gets Threshold:   10000
-> Note that resources reported for PL/SQL includes the resources used by
   all SQL statements called within the PL/SQL code.  As individual SQL
   statements are also reported, it is possible and valid for the summed
   total % to exceed 100
                                                    CPU  Elapsd
Buffer Gets Executions Gets per Exec  %Total Time  Time  Hash Value
--------------- ------------ -------------- ------ -------- --------
  10,038        24        418.3        63.3   1.15  1.11  238087931
select count(*) from sys.job$ where (next_date > sysdate) and (n
ext_date < (sysdate+5/86400))
```

You can also review the instance activity statistics, which tells you how the CPU was being used by the instance. Listing 19-23 shows the instance activity statistics.

Listing 19-23. The Instance Activity Statistics

```
Instance Activity Stats
Statistic                          Total   per Second  per Trans
---------------------------      -------  ----------- ----------
CPU used by this session             420       0.6        105.0
CPU used when call started           417       0.6        104.3
DBWR checkpoint buffers written      199       0.3         49.8
SQL*Net roundtrips to/from client  100       0.1         25.0
consistent changes                 1,128       1.5        282.0
consistent gets                    9,636      12.7      2,409.0
data blocks consistent reads-un  1,128       1.5        282.0
db block changes                   7,872      10.4      1,968.0
db block gets                      6,214       8.2      1,553.5
deferredCURRENT)block cleanout   1,010       1.3        252.5
dirty buffers inspected                5       0.0          1.3
```

In addition to the preceding information, the Statspack report also provides you with the following details (note that the Statspack data was collected at Level 5, which is the default level):

- Tablespace I/O stats
- File I/O stats
- Buffer pool statistics and the Buffer Pool Advisory for the instance
- Instance recovery statistics
- PGA aggregate target statistics
- Undo segment statistics
- Latch activity and library cache activity

NOTE *For a more on the Statspack utility, please refer to the Oracle paper "Diagnosing Performance Using Statspack"* (`http://otn.oracle.com/deploy/performance/pdf/statspack.pdf`).

Once you learn how the Statspack utility works, you may want to invest in a GUI-based tool through which you can launch Statspack statistics collection, create customized charts and reports, and get expert tuning advice. For one such tool, please visit the Statspack Viewer Professional page (`http://www.statsviewer.narod.ru/sppro.html`).

Using the OEM Diagnostics Pack to Monitor Performance

You can use a number of dynamic performance table–based scripts to check database performance. However, there are several drawbacks to the use of scripts. First, it just takes too much time to run all the scripts manually. Second, it takes an even longer time to pore over the reports and come up with meaningful conclusions. For these reasons, the use of a GUI-based tool is mandatory to get a quick visual idea of how well the database is performing. OEM has a specialized Diagnostics Pack that lets you capture operating system, middle-tier, and application and instance performance data. You can thus easily diagnose system problems, and detect and analyze the cause of problems while they are occurring. This is difficult to achieve by exclusively using SQL and operating system scripts.

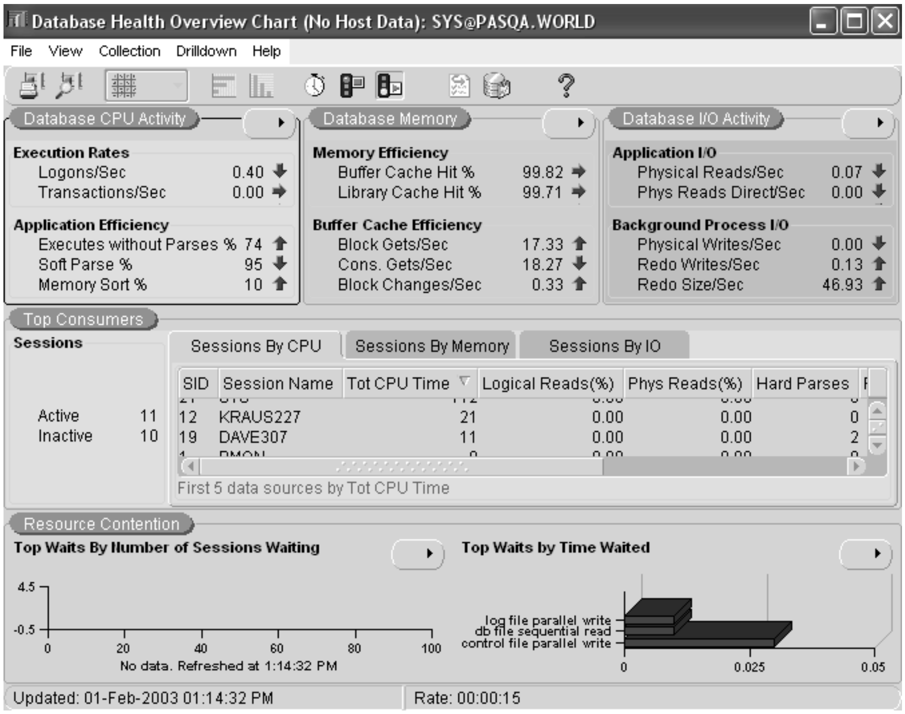
NOTE *The current version of OEM's Diagnostics Pack has very useful and powerful performance-monitoring features. Oracle recommends using this tool for database monitoring and tuning. In fact, Oracle recommends using Statspack only if you don't have the Diagnostics Pack. Remember that OEM also has a built-in capability to collect operating system statistics.*

The Diagnostics Pack uses the same V$ tables as the other performance evaluation methods such as Statspack, but it saves the data and makes efficient use of it. You can launch the Diagnostics Pack from the OEM console using either the menu or an icon. You can also select Programs ➤ Oracle Enterprise Manager ➤ Diagnostics Pack if you wish. For a feature review of the Oracle Diagnostics Pack, please read the white paper "Oracle Diagnostics Pack" (`http://otn.oracle.com/products/oem/pdf/DP_9iR2_FO.pdf`).

The Performance Manager

The Performance Manager component of the Diagnostics Pack provides you with an excellent interface for a real-time study of how key resources such as memory, I/O, and the CPU are performing. You can access the Database Health Overview Chart of the Performance Manager from the OEM console by selecting Tools ➤ Diagnostics Pack ➤ Performance Manager. Figure 19-2 shows the main screen of the Performance Manager, the Database Health Overview Chart.

The Database Health Overview Chart helps you track down the causes for deterioration in the I/O, memory, and CPU resources while the problem is manifesting itself. You can also get summary wait information for the instance from this chart. You can drill down into any one of these resources and find the exact SQL statement responsible for the problem. This is powerful, because you can quickly get to the problem SQL and start analyzing it.



Execution Rates

Logons/Sec	0.40	↓
Transactions/Sec	0.00	→

Application Efficiency

Executes without Parses %	74	↑
Soft Parse %	95	↓
Memory Sort %	10	↑

Memory Efficiency

Buffer Cache Hit %	99.82	→
Library Cache Hit %	99.71	→

Buffer Cache Efficiency

Block Gets/Sec	17.33	↑
Cons. Gets/Sec	18.27	↓
Block Changes/Sec	0.33	↑

Application I/O

Physical Reads/Sec	0.07	↓
Phys Reads Direct/Sec	0.00	↓

Background Process I/O

Physical Writes/Sec	0.00	↓
Redo Writes/Sec	0.13	↑
Redo Size/Sec	46.93	↑

Sessions

Active	11
Inactive	10

SID	Session Name	Tot CPU Time	Logical Reads(%)	Phys Reads(%)	Hard Parses
12	KRAUS227	21	0.00	0.00	0
19	DAVE307	11	0.00	0.00	2

First 5 data sources by Tot CPU Time

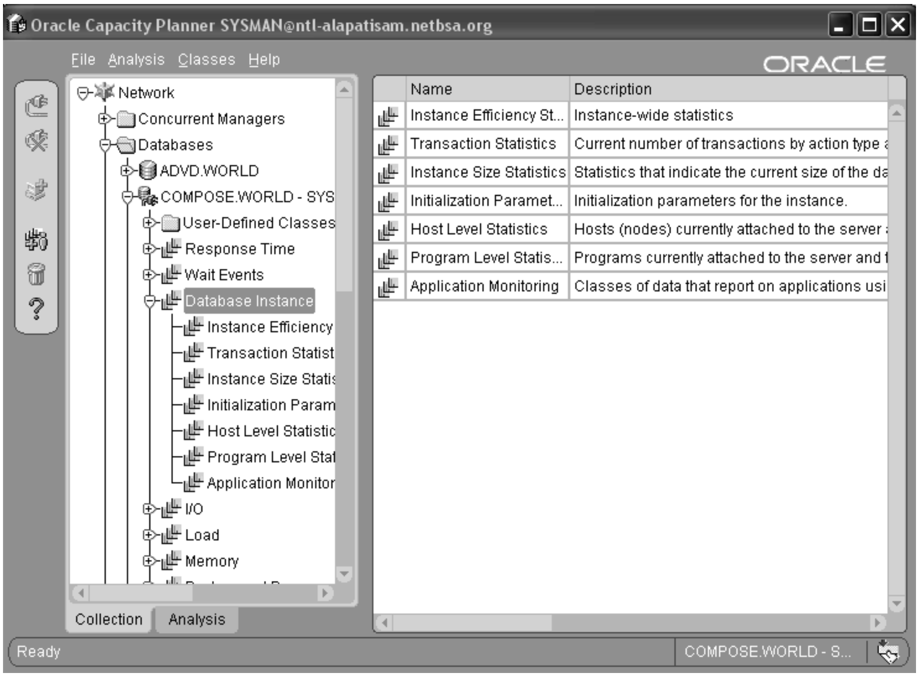
Updated: 01-Feb-2003 01:14:32 PM   Rate: 00:00:15

Figure 19-2. The Database Health Overview Chart of the OEM Performance Manager

The Capacity Planner

The Capacity Planner is another Diagnostics Pack component that you can use to obtain excellent statistics of database performance. You can use this OEM feature to obtain historical performance data that you can store in a history database, so you can plan future requirements of CPU and I/O.

Figure 19-3 shows the Capacity Planner application of the OEM Diagnostics Pack. You can either use Oracle's recommended collection of performance statistics or specify your own metrics. You can select the scope, frequency, and the length of time you want the data to be saved. Historical data collection should be a priority for any Oracle DBA, because it's hard to figure out *what* the instance statistics should look like if you haven't collected any data over time.



Name	Description
Instance Efficiency St...	Instance-wide statistics
Transaction Statistics	Current number of transactions by action type a
Instance Size Statistics	Statistics that indicate the current size of the da
Initialization Paramet...	Initialization parameters for the instance.
Host Level Statistics	Hosts (nodes) currently attached to the server a
Program Level Statis...	Programs currently attached to the server and t
Application Monitoring	Classes of data that report on applications usi

Figure 19-3. The Diagnostics Pack Capacity Planner

Measuring Instance Performance

One of the trickiest parts of the DBA's job is to accurately judge the performance of the Oracle instance. Trainers and the manuals advise you to perform diligent proactive tuning, but in reality most tuning efforts are reactive—they're intensive attempts to fix problems such as a perceptibly slow database that's causing user complaints to increase. You look at the same things whether you're doing proactive or reactive tuning, but proactive tuning gives you the luxury of making decisions in an unhurried and low-stress environment. Ideally, you should spend over two-thirds of your total tuning time on proactive planning. As you do so, you'll find that you're reacting less and less over time to sudden emergencies.

There are statistics that you can look at to see how well the database is performing. These statistics fall into two groups: database hit ratios and database wait statistics. The hit ratios tell you how well the different parts of the SGA memory are configured. If you are consistently seeing numbers in the high 90s for the various hit ratios you saw earlier in this chapter, there is no need to add any more memory to these components.

However, the big question is this: Do high ratios automatically imply a perfectly tuned and efficient database? The surprising answer is no. To understand this confusing fact, you need to look at what hit ratios really indicate. The following sections examine the two main groups of performance statistics.

Database Hit Ratios

Database hit ratios are the most commonly used measures of performance. These include the buffer cache hit ratio, the library cache and dictionary cache hit ratios, the latch hit ratio, and the disk sort ratios. These hit ratios don't really indicate how well your system is performing. They're very broad indicators of proper SGA allocation, and they may be very high even when the system as a whole is performing poorly. The thing to remember is that the hit ratios just measure ratios such as how physical reads compare with logical reads and how much of the time a parsed version of a statement is found in memory. As to whether the statements themselves are efficient or not, the hit ratios can't tell you anything. When your system is slow due to bottlenecks, the hit ratios are of little help and you should turn to a careful study of wait statistics instead.

CAUTION *Even if you have a 99.99 percent buffer cache hit ratio, you may still have major inefficiencies in your application. What if you have an extremely high number of "unnecessary" logical reads? This will make your buffer cache hit ratio look really good, as that hit ratio is defined as physical reads over the sum of logical reads. Although you may think your application should run faster because you're doing most of your reads from memory instead of disk, this may very well not happen. The reason is that even if you're doing logical reads, you're still burning up the CPU units to do the unnecessary logical reads. In essence, by focusing zealously on the buffer cache hit ratio to relieve the I/O subsystem, you could be an unwitting party to a CPU usage problem. Please read Cary Millsap's interesting article, "Why You Should Focus on LIOs instead of PIOs"* (`http://www.hotsos.com`)*, which explains why a high logical I/O level could be a major problem.*

As you're aware by now, high buffer cache hit ratios don't necessarily mean that the application is well tuned. All a high buffer cache hit ratio is telling you is that your physical reads are very small when compared to the total logical reads from the buffer cache. It's entirely possible for you to have a 99.0 buffer cache hit ratio and still have a database slowdown.

When faced with a slow-performing database or a demand for shorter response times, Oracle DBAs have traditionally looked to increase their database hit ratios and tune the database by adjusting a host of initialization parameters (such as spin count and so on). More recently, there's awareness that the key area to focus on is clearing up database bottlenecks that contribute to a lower response time.

The total response time for a query is the time Oracle takes to execute it plus the time the process spends waiting for resources such as latches, data buffers, and so on. For a database instance to perform well, ideally your application should spend very little time waiting for access to critical resources.

Let's now turn to examine the critical wait events in your database, which can be real showstoppers on a busy day in a production instance.

Database Wait Statistics

When your users complain that the database is crawling and they can't get their queries returned fast enough, there is no use in your protesting that your database is showing high hit ratios for the shared pool and the buffer cache (and the large pool and redo log buffer as well). If the users are waiting for long periods of time to complete their tasks, then the response time will be slow, and you can't say that the database is performing well, the high hit ratios notwithstanding.

NOTE *For an interesting review of the Oracle wait analysis (the wait interface), please read the article "Yet Another Performance Profiling Method (or YAPP-Method),"by Anjo Kolk, Shari Yamaguchi, and Jim Viscusi, which is available at* `http://www.dbatoolbox.com/WP2001/tuning/O8I_tuning_method.htm`.

An Oracle process, once it starts executing a SQL statement, doesn't always get to "work" on the execution of the statement without any interruptions. Often, the process has to pause or wait for some resource to be released before it can continue its execution. Thus, an active Oracle process is doing one of the following at any given time:

- The process is executing the SQL statement.
- The process is waiting for something (e.g., a resource such as a database buffer or a latch). It could be waiting for an action such as a write to the buffer cache to complete.

That's why the response time—the total time taken by Oracle to finish work—is correctly defined as follows:

```
Response time =  service time + wait time.
```

When you track the total time taken by a transaction to complete, you may find that only part of that time was taken up by the Oracle server to actually "do" something. The rest of the time, the server may have waiting for some resource to be freed up or waiting for a request to do something. This busy resource may be a slow log writer or database writer process. The wait event may also be due to unavailable buffers or latches. The wait events in the V$SYSTEM_EVENT view (instance-level waits) and the V$SESSION_EVENT view (session-level waits) will tell you what the wait time is due to (full table scans, high number of library cache latches, and so on). Not only will the wait events tell you what the wait time in the database instance is due to, but they will also tell you a lot about bottlenecks in the network and the application.

NOTE *It is very important to understand that the wait events are only the* symptoms *of problems, most likely within the application code. The wait events show you what is slowing down performance, but not why a certain wait event is showing up in large numbers. It is up to you to investigate the SQL code to find out the real cause of the performance problems.*

Three dynamic performance views contain wait information: V$SYSTEM_EVENT, V$SESSION_EVENT, and V$SESSION_WAIT. These three views list just about all the events the instance was waiting for and the duration of these waits. Understanding these wait events is essential for resolving performance issues.

Let's look at the common wait events in detail in the following sections. Remember that the three views I mentioned previously show similar information but focus on different aspects of the database, as you can see from the following summary. The wait events are most useful when you have timed statistics turned on. Otherwise, the wait events will only have the number of times they occurred, not the length of time they consumed. Without timing the events, you really can't tell if a wait event was indeed a contributing factor in a system slowdown.

Using V$ Tables for Wait Information

The three key dynamic performance tables for finding wait information are the V$SYSTEM_EVENT, V$SESSION_EVENT, and V$SESSION_WAIT views. The first two views show the waiting time for different events.

The V$SYSTEM_EVENT view shows the total time waited for all the events for the entire system since the instance started up. The view does not focus on the individual sessions experiencing waits, and therefore it gives you a high-level view of waits in the system. You can use this view to find out what the top instancewide wait events are. You can calculate the top *n* waits in the system by dividing the event's wait time by the total wait time for all events.

The three key columns of the V$SYSTEM_EVENT view are total_waits, which gives the total number of waits; time_waited, which is the total wait time per session since the instance started; and average_wait, which is the average wait time by all sessions per event.

The V$SESSION_EVENT view is similar to the V$SYSTEM_EVENT view, and it shows the total time waited per session. By querying this view, you can find out the specific bottlenecks encountered by each session.

The third dynamic view is the V$SESSION_WAIT view, which shows the current waits or just completed waits for sessions. The information on waits in this view changes continuously based on the type of waits that are occurring in the system. The real-time information in this view provides you tremendous insight into what is holding up things in the database *right now*. The V$SESSION_WAIT view provides detailed information on the wait event, including details such as file number, latch numbers, and block number. This detailed level of information provided by the V$SESSION_WAIT view enables you to probe into the exact bottleneck that is currently slowing down the database. The low-level information helps you zoom in on the root cause of performance problems.

Listing 19-24 shows you the structure of the important V$SESSION_WAIT view.

Listing 19-24. The V$SESSION_WAIT View

```
SQL> desc v$session_wait
 Name            Null?    Type
 ------------- -------- --------------
 SID                     NUMBER
 SEQ#                    NUMBER
 EVENT                   VARCHAR2(64)
 P1TEXT                  VARCHAR2(64)
 P1                      NUMBER
 P1RAW                   RAW(4)
 P2TEXT                  VARCHAR2(64)
 P2                      NUMBER
 P2RAW                   RAW(4)
 P3TEXT                  VARCHAR2(64)
 P3                      NUMBER
 P3RAW                   RAW(4)
 WAIT_TIME               NUMBER
 SECONDS_IN_WAIT         NUMBER
 STATE                  VARCHAR2(19)
SQL>
```

The following columns from the V$SESSION_WAIT view are important for troubleshooting performance issues:

- *Event:* These are the different wait events described in the next section (e.g., latch free and buffer busy waits).
- *P1, P2, P3:* These are the additional parameters that represent different items, depending on the particular wait event. For example, if the wait event is *db_file_sequential_read*, P1 stands for the file number, P2 stands for the block number, and P3 stands for the number of blocks. If the wait is due to a latch-free event, P1 stands for the latch address, P2 stands for the latch number, and P3 stands for the number of attempts for the event.
- *Wait_time:* This is the wait time in seconds if the state is *waited known time.*
- *Seconds_in_wait:* This is the wait time in seconds if the state is *waiting.*
- *State:* The state could be *waited short time, waited known time,* or *waiting,* if the session is currently waiting for an event.

You can start analyzing the wait events in your system by first querying the V$SYSTEM_EVENT view to see if there are any significant wait events currently occurring in the database. You can do this by running the query shown in Listing 19-25.

Listing 19-25. Using the V$SYSTEM_EVENT View to View Wait Events

```
 select event, time_waited, average_wait
  2  from v$system_event
  3  group by event, time_waited, average_wait
  4* order by time_waited desc;
EVENT                              TIME_WAITED        AVERAGE_WAIT
------------------------------------------------------------------
rdbms ipc message                    24483121          216.71465
SQL*Net message from client          18622096          106.19049
PX Idle Wait                         12485418          205.01844
pmon timer                            3120909          306.93440
smon timer                            3093214        29459.18100
PL/SQL lock timer                     3024203         1536.68852
db file sequential read                831831             .25480
db file scattered read                 107253             .90554
free buffer waits                       52955           43.08787
log file parallel write                 19958            2.02639
latch free                               5884            1.47505
…
58 rows selected.
SQL>
```

This example shows a very simple system with hardly any waits other than the idle type of events and the SQL*Net wait events. There aren't any significant I/O-related or latch contention–related wait events in this database. However, if your query on a real-life production system shows significant numbers for any "nonidle" wait event, it's probably a good idea to find out the SQL statements that are causing the waits. After all, the SQL statements are causing the high waits, so that's where you have to focus your efforts to reduce the waits. You have different ways to obtain the associated SQL for the waits, as explained in the following section.

Obtaining Wait Information

Obtaining wait information is as easy as querying the related dynamic performance tables. For example, if you wish to quickly find out the types of waits different user sessions (session-level wait information) are facing and the SQL text of the statements that they are executing, you can use the following query:

```
SQL> select s.username,
     t.sql_text, w.event
     from v$session s, v$sqltext t, v$session_wait w
     where s.sql_hash_value = t.hash_value
     and s.sql_address      = t.address
     and s.type            <> 'BACKGROUND'
     and s.sid              = w.sid
     order by s.sid,t.hash_value,t.piece;
```

NOTE *You need to turn on statistics collection by either setting the initialization parameter* timed_ statistics *to* true *or setting the initialization parameter* statistics_level *to* typical *or* all.

If you want quick instancewide wait event status, showing which events are the biggest contributors to total wait time, you can use the query shown in Listing 19-26 (there will be several "idle" events listed in the output, but don't show them here).

Listing 19-26. Instance-wide Waits Sorted by Total Wait Time

```
SQL> select event, total_waits,time_waited from V$system_event
  2  where event NOT IN
  3  ('pmon timer', 'smon timer', 'rdbms ipc reply', 'parallel deque wait',
  4  'virtual circuit', '%SQL*Net%', 'client message', 'NULL event')
  5* order by time_waited desc;
EVENT                              TOTAL_WAITS         TIME_WAITED
------------------------          -------------       -------------
db file sequential read              35051309            15965640
latch free                            1373973             1913357
db file scattered read                2958367             1840810
enqueue                                  2837              370871
buffer busy waits                      444743              252664
log file parallel write                146221              123435
SQL>
```

It's somewhat confusing in the beginning when you're trying to use all the wait-related V$ views, which all look very similar. Here's a quick summary of how you go about using the key wait-related Oracle9*i* dynamic performance views.

First, look at the V$SYSTEM_EVENT view and rank the top wait events by the total amount of time waited, as well as the average wait time for that event. Start investigating the top waits in terms of the percentage of total wait time. You can also look at any Statspack reports you may have, because Statspack also lists the top five wait events in the instance.

Next, find out more details about the specific wait event that's at the top of the list. For example, if the top event is buffer busy waits, look in the V$WAITSTAT view to see which buffer block is causing the busy buffer waits (a simple *select ** from V$WAITSTAT will get you all the necessary information). For example, if the undo block buffer waits make up most of your buffer busy waits, then it's the undo segments that are at fault, not the data blocks.

Finally, use the V$SESSION_WAIT view to find out the exact objects that may be the source of a problem. For example, if you have a high amount "db file scattered reads"–type waits, the V$SESSION_WAIT view will give you the file number and block number involved in the wait events. In the following example, the V$ views V$SESSION and V$SESSION_WAIT are used to find out who is doing the full table scans showing up as the most important wait events right now:

```
  SQL> select S.sid, s.sql_address, s.sql_hash_value
       from V$SESSION s, V$SESSION_WAIT w
       where w.event like '%scattered read'
       and w.sid = s.sid;
```

You can also use the V$SQLAREA view to find out which SQL statements are responsible for high disk reads. If latch waits predominate, you should be looking at the V$LATCH view to gain more information about the type of latch that's responsible for the high latch wait time.

Looking at Segment-Level Statistics

Whether you use the Statspack tool or the wait-related V$ views, you're going to find no information about where a certain wait event is occurring. For example, you can see from the V$SYSTEM_EVENT view that buffer busy waits are your problem, and you know that you reduce these waits either by increasing the I/O bandwidth or by modifying the table storage parameters and adjusting parameters such as FREELISTS and INITTRANS. However, neither Statspack nor the V$ view indicates which tables or indexes you should be looking at to fix the high wait events. Oracle 9.2 provides three new V$ views to help you drill down to the *segment level.*

The new segment level dynamic performance views are V$SEGSTAT_NAME, V$SEGSTAT, and V$SEGMENT_STATISTICS. Now you can find out which of your tables and indexes are being subjected to high resource utilization or high waits. Once you are aware of a performance problem due to high waits, you can use these segment-level views to find out exactly which table or index is the culprit and fix that object to reduce the waits and increase database performance.

The V$SEGMENT_NAME view provides you with a list of all the segment levels that are being collected, and tells you whether the statistics are sampled or not. Listing 19-27 shows you a query on the V$SEGSTAT_NAME view that gives you all the events for which segment-level information is being collected since the instance started.

Listing 19-27. Events for Which Segment-Level Data Is Available

```
SQL> select * from v$segstat_name;
     STATISTIC         # NAME                            SAMPLED
---------- ---------------------------------------------------
         0         logical reads                          YES
         1         buffer busy waits                      NO
         2         db block changes                       YES
         3         physical reads                         NO
         4         physical writes                        NO
         5         physical reads direct                  NO
         6         physical writes direct                 NO
         8         global cache cr blocks served          NO
         9         global cache current blocks served     NO
        10         ITL waits                              NO
        11         row lock waits                         NO
11 rows selected.
SQL>
```

Let's see how you can use these segment-level views to your advantage when you're confronted by a high number of wait events in your system. Say that you look at the V$SYSTEM_EVENT view and realize that there are a large number of buffer busy waits. You should now examine the V$SEGMENT_STATISTICS view with a query such as the following to find out which object is the source of the high buffer busy waits. You can then decide on the appropriate corrective measures for this wait event, as discussed in the section "Important Oracle Wait Events" later in this chapter.

```
SQL> select owner, object_name, object_type, tablespace_name
     2  from v$segment_statistics
     3  where statistic_name='buffer busy waits'
     4  order by value desc ;
```

Collecting Detailed Wait Event Information

Selecting data from V$ dynamic performance views and interpreting them meaningfully isn't always so easy to do. Because the views are dynamic, the information that they contain is constantly changing, with Oracle updating the underlying tables for each wait event. Also, the wait-related dynamic performance views you just examined don't provide crucial data such as bind variable information. For a more detailed level of wait information, you can use one of the three methods described in the following sections. The first method is simple to use and should suffice for most cases.

Method 1: Using the Oracle Event 10046 to Trace SQL Code

You can get all kinds of bind variable information by using a special trace called the 10046 Trace, which is much more advanced than the SQL Trace you saw in Chapter 18. The use of this trace will cause an output file to be written to the trace directory. You can set the 10046 Trace in many ways by specifying various levels, with each higher level providing you with more detailed information. (Level 12 is used in the following case as an example only—it may give you much more information than necessary. Level 4 will give you detailed bind value information, and Level 8 will give you wait information.)

You can use the *alter session* statement as follows:

```
SQL> alter session set events '10046 trace name context forever, level 12';
Session altered.
SQL>
```

You can also incorporate the following line in your init.ora file:

```
Event=  10046 trace name context forever, level 12
```

Method 2: Using the Oradebug Utility to Perform the Trace

You can use the oradebug utility as shown in the following example:

```
SQL> oradebug setmypid
Statement processed.
SQL> oradebug event 10046 trace name context forever, level 8;
Statement processed.
SQL>
```

In this example, *setmypid* indicates that you want to trace the current session. If you want a different session to be traced, you replace this with *setospid <Process Id>*.

Method 3: Using the DBMS_SYSTEM Package to Set the Trace

The syntax for this package is shown in Chapter 18. Also, Chapter 21 explains the DBMS_SYSTEM package in more detail. Using the *set_ev* procedure of the DBMS_SYSTEM package, so you can set tracing on in any session, as shown in the following example:

```
SQL> execute sys.dbms_system.set_ev (9,271,10046,12,'');
PL/SQL procedure successfully completed.
SQL>
```

Important Oracle Wait Events

The wait events listed in the sections that follow have a significant impact on system performance by increasing response times. Each of these events (and several other events) indicates an unproductive use of time because of an excessive demand for a resource or contention for Oracle structures such as tables or the online redo log files.

NOTE *The V$EVENT_NAME view will give you the complete list of all Oracle wait events and their description.*

Buffer Busy Waits

The *buffer busy waits* event occurs in the buffer cache area when several processes are trying to access the same buffer in the buffer cache area. One session is waiting for another session's read of a buffer into the buffer cache. This wait could also occur when the buffer is in the buffer cache, but another session is changing the buffer.

You should observe the V$SESSION_WAIT view while this wait is occurring to find out exactly what type of block is causing the wait. If the waits are primarily on data blocks, try increasing the PCTFREE parameter to lower the number of rows in each data block. You may also want to increase the INITRANS parameter to reduce contention from competing transactions. If the waits are mainly in segment headers, increase the number of FREELISTS or FREELIST GROUPS for the segment in question, or consider increasing the extent size for the table or index.

Buffer busy waits can occur for several types of buffers, including data block and header blocks buffers for tables and undo segments. However, if you are using Automatic Undo Management (AUM), the undo segments won't have this problem, leaving table and index data block buffers as the main problem areas. The following query clearly shows that in this database, the buffer busy waits are really in the data block:

```
SQL>  select class,count from v$waitstat
   2  where count > 0
   3* order by count desc;
CLASS                   COUNT
------------------ ----------
data block             519731
undo block               5829
undo header              2026
segment header             25
SQL>
```

If data block buffer waits are the main problem, this could be a problem caused by poorly chosen indexes that are leading to large index range scans. Tuning the SQL statements is necessary to fix these waits. Oracle maintains that if you use AUM instead of traditional rollback segments, the two types of buffer busy waits, undo block and undo header, will go away. However, that's not the case in practice, as the following example from a database with AUM shows:

```
CLASS                   COUNT
------------------ ----------
undo header             29891
data block                 52
segment header              1
```

Once in a while, you may have a situation where the buffer busy waits spike suddenly, seemingly for no reason. The sar utility (*sar –d*) might indicate high request queues and service times. This often happens when the disk controllers get saturated by a high amount of I/O. Usually, you see excessive core dumps during this time in your core dump directory. If core dumps are choking your I/O subsystem, do the following:

- Move your core dump directory to a less busy file system, where it resides by itself.

- Use the following init.ora/SPFILE parameters to control core dumps in your system. Setting these parameters' values could reduce the size of a core dump to a few megabytes from a gigabyte or more.

```
Shadow_core_dump = partial /* or none */
Background_core_dump = partial /* or none */
```

- Investigate the core dumps and see if you can fix them by applying necessary Oracle and operating system patch sets.

Checkpoint Completed

The *checkpoint completed* wait event means that a session is waiting for a checkpoint to complete. This could happen when you're shutting the database down or during normal checkpoints.

DB File Scattered Read

The *DB file scattered read* wait event indicates that full tables scans (or index fast full scans) are occurring in the database. During full table scans or index fast full scans, the table data blocks are read into scattered rather than contiguous buffers. The number of blocks read at one time by Oracle is set by the initialization parameter *db_file_multiblock_read_count*. Although Oracle will read in multiblock chunks, it scatters them into noncontiguous cache buffers. If the full table scans are not high in number, and if they mainly consist of smaller tables, don't worry about it.

However, if this event is showing up as an important wait event, you need to look at it as an I/O-related problem—the database isn't able to cope with an excessive request for physical I/Os. There are two possible solutions. You can either reduce the demand for physical I/Os or increase the capacity of the system to handle more I/Os. You can reduce the demand for physical I/O by drilling down further to see if one of the following solutions will work:

- Add missing indexes on key tables (unlikely in a production system).
- Optimize SQL statements if they aren't following an efficient execution plan currently.

If you don't see any potential for reducing the demand for physical I/O, you're left with no choice but to increase the number of disks on your system. You also need to make sure you're reducing the hot spots in your system by carefully distributing the heavily hit tables and indexes across the available disks. You can identify the data files where the full table or index fast full scans are occurring with the help of a query using the V$FILESTAT view. In this view, two columns are of great use:

- *Phyrds:* The number of physical reads done
- *Phyblkrd:* The number of physical blocks read

Obviously, if the amount of phyrds is equal to or close to the amount of phyblkrds, almost all reads are single block reads. If the column phyrds shows a much smaller value than the phyblkrds column, Oracle is reading multiple blocks in one read—a full table scan or an index fast full scan, for example. Here's a sample query on the V$FILESTAT view:

```
SQL>  select file#, phyrds,phyblkrd
  2   from v$filestat
  3*  where phyrds != phyblkrd;
     FILE#     PHYRDS   PHYBLKRD
---------- ---------- ----------
         1       4458      36533
         7      67923     494433
        15      28794     378676
        16      53849     408981
SQL>
```

DB File Sequential Read

The *DB file sequential read* wait event signifies that a single block is being read into the buffer cache. This event occurs when you're doing an indexed read and you're waiting for a physical I/O call to return. This is nothing to be alarmed about, because the database has to wait for file I/O. However, you should investigate disk I/O if this statistic seems extraordinarily high. Because the very occurrence of this event proves that your application is making heavy use of an index, you really can't do much to reduce the demand for physical I/Os in this case, unlike in the case of the DB file scattered read event. You should see whether increasing the FREELISTS for the indexes would reduce the sequential reads. Increasing the number of disks and striping indexes across them may be your best bet to reduce DB file sequential read waits.

Direct Path Read and Direct Path Write

The *direct path read* and *direct path write* events are waits during performing a direct read or write into the PGA, bypassing the SGA buffer cache. Direct path reads indicate that sorts are being done on disk instead of in memory. They could also result from a busy I/O system. If you use automatic PGA tuning (see Chapters 5 and 7), you shouldn't encounter this problem too often.

Automatic tuning of the PGA by Oracle should reduce your disk sorts due to a low PGA memory allocation. Another solution may be to increase the number of disks, as this problem also results in an I/O system that can't keep up with the increased requests for reading blocks into the PGA. Of course, tuning the SQL statements themselves to reduce sorting wouldn't hurt in this case.

Free Buffer Waits

Free buffer waits usually show up when the database writer process is slow. The database writer process is simply unable to keep up with the requests to service the buffer cache. All buffers are dirty (modified) and the database writer process is trying to write their contents to disk. The number of dirty buffers in cache waiting to be written to disk is larger than the number of buffers the database writer process can write per batch. Meanwhile, sessions have to wait because they can't get free buffers to write to. You need to first rule out that the buffer cache is not too small, and check the I/O numbers on the server, especially the write time, using an operating system tool. A check of the database buffer cache and a quick peek at OEM's Buffer Cache Advisor may tell you that you're below your optimal buffer cache level, in which case you can increase the size of the buffer cache.

The other reason for a high number of buffer busy waits in your system is that the number of database writer processes is inadequate to perform the amount of work your instance needs to get done. As you know, you can add additional database writer processes to the default single DBWR0 process. You can reduce these waits in most cases by increasing the number of database writer processes using a value that is in between 2 and 10 for the *db_writer_processes* initialization parameter. Oracle recommends that you use one database writer process for every four CPUs on your system. You can't change this variable on the fly, so you'll need to perform a system restart to change the number of database writer processes.

Enqueue Waits

Enqueues are similar to locks in that they are internal mechanisms that control access to resources. High enqueue waits indicate that a large number of sessions are waiting for locks held by other sessions. You can query the dynamic performance view V$ENQUEUE_STAT to find out which of the enqueues have the most wait times reported. You can do this by using the cum_wait_time column of the view.

Note that the use of locally managed tablespaces will eliminate several types of enqueues such as space transactions (ST) enqueues. In a system with a massive concurrent user base, most common enqueues are due to infrequent commits (or rollbacks) by transactions that force other transactions to wait for the locks held by the early transactions. In addition, there may be a problem with too few interested transactions list (ITL) slots, which will also show up as a transaction (TX) enqueues. You can increase the value of the INITTRANS or MAXTRANS parameter for the key tables in your system that undergo a large number DML operations.

Latch Free

Latches are internal serialization mechanisms used to protect shared data structures in Oracle's SGA. You can consider latches as a type of lock that is held for an extremely short time period. Oracle has several types of latches, with each type guarding access to a specific set of data. The *latch-free* wait event is incremented when a process can't get a latch on the first attempt. If a required Oracle latch is not available, the process requesting it keeps spinning and retrying for the access (the spinning is limited by the spin count initialization parameter). The continuous spinning will increase the wait time and increase the CPU usage in the system. There are about 200 latches used by Oracle, but two of the important latches that show up in wait statistics are the *shared pool latch* (and the library cache latches) and the *cache buffers LRU chain*. It's normal to see a very high number of latch-free events in an instance. You should worry about this wait event only if the total time consumed by this event is high.

High latch waits will show up in your Statspack reports, or you can use the query shown in Listing 19-28 to find out your latch hit ratio.

Listing 19-28. Determining the Latch Hit Ratio

```
SQL> select  a.name "Latch Name",
       a.gets "Gets (Wait)",
       a.misses "Misses (Wait)",
       (1 - (misses / gets)) * 100 "Latch Hit Ratio %"
FROM   v$latch a
WHERE  a.gets   != 0
UNION
SELECT a.name "Latch Name",
       a.gets "Gets (Wait)",
       a.misses "Misses (Wait)",
       100 "Latch Hit Ratio"
FROM   v$latch a
WHERE  a.gets   = 0
ORDER BY 1;
SQL>
```

If the ratio is not close to 1, it's time to think about tuning the latch contention in your instance. There is only one shared pool latch for the database, and it protects the allocation off memory in the library cache. The library cache latch regulates access to the objects present in the library cache. Any SQL statement, PL/SQL code, procedure, function, or package needs to acquire this latch before execution. If the shared pool and library cache latches are high, more often than not that's because the parse rates in the database are high. The high parse rates are due to the following factors:

- An undersized shared pool (or an oversized shared pool)
- Failure to use bind variables
- Using dissimilar SQL statements and failing to reuse statements
- Users frequently logging off and logging back into the application
- Failure to keep cursors open after each execution
- Using a shared pool size that is too large

The *cache buffers LRU chain* latch-free wait is caused by very high buffer cache throughput either due to full table scans or the use of unselective indexes, which lead to large index range scans. Unselective indexes can also lead to yet another type of latch-free wait: the *cache buffer chain* latch-free wait. The cache buffer chain latch waits are often due to the presence of hot blocks, so you need to investigate why that might be happening. If you see a high value for row cache objects latch waits, it indicates contention for the dictionary cache, and you need to increase the shared pool memory allocation.

In most instances, latch waits tend to show up as a wait event, and DBAs sometimes are alarmed by their very presence in the wait event list. As with the other Oracle wait events, ask yourself this question: "Are these latch waits a significant proportion of my total wait time?" If the answer is no, don't worry about it—your goal isn't to try and eliminate all waits in the instance, because you can't do it.

Log Buffer Space

The *log buffer space* wait event indicates that a process waited for space in the log buffer. Either the log buffer is too small or the redo is being written faster than the log writer process can write it to the redo log buffer. If the redo log buffer is already at a large size, then investigate the I/O to the disk that houses the redo log files. There is probably some contention for the disk, and you need to work on reducing that. This type of wait usually shows up when the log buffer is too small, in which case you increase the log buffer size. A large log buffer will tend to reduce the redo log I/O in general. Note that Oracle's default value for this parameter could be as high as 4MB (for a 32-CPU system). Don't be afraid to set a fairly large size for this parameter, although Oracle claims that about 1MB should be sufficient.

The other way to attack this problem is to increase the number of log writer processes. You can configure more log writer processes in addition to the initial log writer process. You may have to do this if you determine the log writer process is unable to keep up with the amount of work.

Log File Switch

The *log file switch* wait event can occur when a session is forced to wait for a log file switch because the log file hasn't yet been archived. It can also occur because the log file switch is awaiting the completion of a checkpoint.

If the problem isn't due to the archive destination getting full, it means that the archive process isn't able to keep up with the rate at which the redo logs are being archived. In this case, you need to increase the number of archiver (ARCHn) processes to keep up with the archiving work. The default for the ARCHn process is 2, and this is a static parameter, so you can't use this fix to resolve a slowdown right away.

You also need to investigate whether too-small redo log files are contributing to the wait for the log file switch. If the log file switch is held up pending the completion of checkpoint, obviously the log files are too small and hence are filling up too fast. You need to increase the size of the redo log files in this case. Redo log files are added and dropped online, so you can consider this a dynamic change.

If you see high values for *redo log space requests* in V$SYSSTAT, that means that user processes are waiting for space in the redo log buffer. This is because the log writer process can't find a free redo log file to empty the contents of the log buffer. Resize your redo logs, with the goal of having a log switch every 15 to 30 minutes.

Log File Sync

You'll see a high number of waits under the *log file sync* category if the server processes are frequently waiting for the log writer process to finish writing committed transactions (redo) to the redo log files from the log buffer. This is usually the result of too-frequent commits, and you can reduce it by adopting batch commits instead of a commit after every single transaction. This wait event may also be the result of an I/O bottleneck.

Idle Events

You can group some wait events under the category *idle events*. Some of these wait events may be harmless in the sense that they just indicate that an Oracle process was waiting for something to do. These events don't indicate database bottlenecks or contention for Oracle's resources. For example, the system may be waiting for a client process to provide SQL statements for execution. The following list presents some common idle events:

- *Rdbms ipc message:* Used by the background process like the log writer process and PMON to indicate they are idle.
- *SMON timer:* The SMON process waits on this event.
- *PMON timer:* The PMON process idle event.
- *SQL*Net Message from Client:* User process idle event.

You should ignore many idle events during your instance performance tuning. However, some events, such as the SQL*Net Message from Client, may indicate that your application may not be using a very efficient database connection strategy. In this case, you need to see how you can reduce these waits, maybe by avoiding frequently logging on and off the applications.

NOTE *An excellent article that shows you how to follow a structured approach to performance tuning is "Oracle9*i *Performance Management: The Oracle Method," by Mughees A. Minhas (*`http://otn.oracle.com/products/manageability/database/pdf/OWPerformanceMgmtPaper.pdf`*). For a useful set of Oracle best practices to ensure high performance, please refer to the paper "Oracle9*i *Database Administration Best Practices," also by Mughees A. Minhas (*`http://otn.oracle.com/products/manageability/database/pdf/BestPractices9i.pdf`*).*

Know Your Application

Experts rely on hit ratios or wait statistics, or sometimes both, but there are situations where both the hit ratios and the wait statistics can completely fail you. Imagine a situation where all the hit ratios are in the 99 percent range. Also, imagine that the wait statistics don't show any significant waiting for resources or any contention for latches. Does this mean that your system is running optimally?

Well, your system is doing what you asked it to do extremely well, but there's no guarantee that your SQL code is processing things efficiently. If a query is performing an inordinate number of logical reads, the hit ratios are going to look wonderful. The wait events also won't show you a whole lot, because they don't capture the time spent while you were actually using the CPU. However, you'll be burning a lot of CPU time, because the query is making too many logical reads.

This example shows why it's important not to rely on just the hit ratios or the wait statistics, but also to look at the major consumers of resources on your instance with an intense focus—for example, why is this query doing a billion logical reads? Check the top session list (sorted according to different criteria) on your instance and see if there's justification for them to be in that list.

Above all, try not to confuse the symptoms of poor performance with the causes of poor performance. If your latch rate is high, there are initialization parameters that you might want to adjust right away—after all, isn't Oracle a highly configurable database? Well, you may succeed sometimes by relying solely on adjusting the initialization parameters, but it may be time to pause and question why exactly the latch rate is so high. More than likely, the high latch rate is due to application coding issues, rather than a specific parameter setting. Similarly, you may notice that your system is CPU bound, but the reason may not be slow or too few CPU resources. Your application may again be the likely culprit, because it is doing too many unnecessary I/Os, even if they're mostly from the database buffer cache and not disk.

When you are examining wait ratios, please understand that your goal is not make all the wait events go away, because that will never happen. Learn to ignore the unimportant and routine, unavoidable wait events such as the control file parallel wait event. As you saw in the previous section, wait events such as the SQL*Net Message from Client event reflect waits outside the database, so don't attribute these waits to a poorly performing database. Focus on the total wait time rather than the number of wait events that show up in your performance tables and Statspack reports. Also, if the wait events make up only a small portion of response time, there's no point in fretting about the waits. That is, if the response time is 20 minutes and the waits are 20 seconds, what's the point of wasting your time investigating the wait events any further? As Einstein might say, the significance of wait events is relative—relative to the total response time and relative to the total CPU execution time.

In addition to the hit ratios and the wait event statistics, you need to monitor other areas of the database for possible performance implications. Among other things, you should try to achieve the following goals to maximize the performance of your database:

- Minimize row chaining and migration.
- Check for blocking locks in your system.
- Make sure that the redo log space requests are kept low by tuning your database writer process. If the number of requests is high, it means the database is slowing down due to an inadequate number/size of redo logs.
- Monitor checkpoint intervals to optimize their frequency.

Recently, there has been a surge in publications that expound the virtues of a wait event analysis–based performance approach (also called the *wait interface* approach). If you use this method for performance tuning, how important are the traditional hit ratios? Well, you can always use the buffer hit ratios and the other ratios for a general idea about how the system is using Oracle's memory and other resources, but an analysis of wait events is still a better bet in terms of improving performance. If you take care of the wait issues, you'll have taken care of the traditional hit ratios as well anyway. For example, if you want to fix a problem that is the result of a high number of free buffer waits, you may need to increase the buffer cache. Similarly, if latch-free wait events are troublesome, one of the solutions is to check if you need to add more memory to the shared pool. You may fix a problem due to a high level of waits caused by the direct path reads by increasing the value of the *pga_aggregate_target* parameter.

Operating System Memory Management

You can use the vmstat utility, as explained in Chapter 3, to find out if there is enough free memory on the system. If the system is paging and swapping, database performance will deteriorate and you need to investigate the causes. If the heavy consumption of memory is due to a non-Oracle process, you may want to move that process off the peak time for your system. You may also want to consider increasing the size of the total memory available to the operating system. You can use the *vmstat* command to monitor virtual memory on a UNIX system. The UNIX tool *top* shows you CPU and memory use on your system.

When a Database Hangs

So far in this chapter, you've looked at ways to improve performance—how to make the database go faster. Sometimes, however, your problem is not the more mundane one of tweaking extra performance from a database, but something much more serious: The database all of a sudden seems to have stopped! What do you do in such a circumstance? Performance was acceptable on a Friday, but Monday morning your manager comes to you to ask why users are complaining about a slow database. What to do? The following sections describe the most important reasons for a hanging or an extremely slow performing database and how you can fix the problem ASAP.

One of the first things I do when the database seems to freeze is check and make sure that the archiver process is doing its job. The following sections describe the archiver process.

Handling a Stuck Archiver Process

If your archive log destination is full and there isn't room for more redo logs to be archived, the archiver process is said to be *stuck*. The database doesn't merely slow down—it freezes in its tracks. As you are aware, in an archive log mode the database simply won't overwrite redo log files until they're archived successfully. Thus, the database starts hanging when the archive log directory is full. It stays in that mode until you move some of the archive logs off that directory manually.

How the Archiver Process Works

The archiver process is in charge of archiving the filled redo logs. The archiver process reads the control files to find out if there are any unarchived filled redo logs. The archiver process checks the redo log headers and blocks to make sure they're valid before archiving them. You may have archiving-related problems if you're in the archivelog mode but the archiver process isn't running for some reason. In this case, you need to start the archiver process by using the following command:

```
SQL> alter system archive log start;
```

If the archiver process is running but the redo logs aren't being archived, then you may have a problem with the archive log destination, which may be full. This will cause the archiver process to become stuck, as you'll learn in the next section.

What If the Archiver Process Is Stuck?

When the archiver process is stuck, all database transactions that involve any changes to the tables can't proceed any further. You can still perform select operations, because they don't involve the redo logs.

If you look in the alert log, you can see the Oracle error messages indicating that the archiver process is stuck due to lack of disk space. You can also query the V$ARCHIVE view. This view holds information about all the redo logs that need archiving. If the number of these logs is very high and increasing quickly, you know your archiver process is stuck and that you need to manually clear it. Listing 19-29 shows the error messages you'll see when the archiver process is stuck.

Listing 19-29. Database Hang Due to Archive Errors

```
[monitor] $ sqlplus system/monitor1
SQL*Plus: Release 9.2.0.1.0 - Production on Thu Jan 23 11:12:42 2003
Copyright (c) 1982, 2002, Oracle Corporation.  All rights reserved.
ERROR:
ORA-00257: archiver error. Connect internal only, until freed.
oracle@hp50.netbsa.org   [/u01/app/oracle]
[monitor] $ oerr ora 257
00257, 00000, "archiver error. Connect internal only, until freed."
//*Cause: The archiver process received an error while trying to
// archive a redo log.  If the problem is not resolved soon, the
// database will stop executing transactions. The most likely cause
// of this message is the destination device is out of space to
// store the redo log file.
// *Action:  Check archiver trace file for a detailed description
// of the problem. Also verify that the device specified in the
// initialization parameter ARCHIVE_LOG_DEST is set up properly for // archiving.
oracle@hp50.netbsa.org   [/u01/app/oracle]
[monitor] $
```

You can do either of the following in such a circumstance:

- Redirect archiving to a different directory.
- Clear the archive log destination by removing some archive logs. Just make sure you back up the archive logs to tape before removing them.

Once you create more space in the archive log directory, the database resumes normal operations and you don't have to do anything further. If the archiver process isn't the cause of the hanging or frozen database problem, then you need to look in other places to resolve the problem.

If you see too many "checkpoint not complete" messages in your alert log, then the problem isn't being caused by the archiver process. Your slowdown is being caused by the redo logs, which are unable to keep up with the high level of updates. You can increase the size of the redo logs online to alleviate the problem.

System Utilization Problems

You need to check several things to make sure there are no major problems with the I/O subsystem or with the CPU usage. Here are some of the important things you need to examine:

- Make sure your system isn't suffering from a severe paging and swapping problem, which could result in a slower performing database.
- Use top, sar, vmstat, or similar operating system–level tools to check resource usage. Large queries, sorting, and space management operations could all lead to an increase in CPU usage.
- Runaway processes and excessive SNP processes could gobble excessive CPU resources. Monitor any replication (snapshot) processes or DBMS_JOB processes, because they both use resource-hungry SNP processes. If CPU usage spikes, make sure there are no unexpected jobs running in the database. Even if there aren't any jobs executing currently, the SNP processes consume a great deal of CPU because they have to constantly query the job queue.
- High runqueues indicate that the system is CPU bound, with processes waiting for an available processor.
- If your disk I/O is close to or at 100 percent and you've already killed several top user sessions, you may have a disk controller problem. For example, the 100 percent busy disk pack might be using a controller configured to 16-bit, instead of 32-bit like the rest of the controllers, causing a severe slowdown in I/O performance.

Excessive Contention for Resources

Usually when people talk about a database hang, they're actually mistaking a severe performance problem for a database hang. This is normally the case when there's severe contention for internal kernel-level resources such as latches and pins.

If your database is performing an extremely high number of updates, contention for resources such as undo segments and latches could potentially be a major source of databasewide slowdowns, making is seem sometimes like the database is hanging. In the early part of this chapter, you learned how to analyze database contention and wait issues using the V$SESSION_WAIT view and the Statspack output. On Windows servers, you can use the Performance Monitor and Event Monitor to locate possible high resource usage.

Check for excessive library cache contention also if you're confronted by a databasewide slowdown.

Locking Issues

If a major table or tables are locked unbeknownst to you, the database could slow down dramatically in very short order. Try running a command such as *select * from persons*, for example, where *persons* is your largest table and is part of just about every SQL statement. If you aren't sure which tables (if any) might be locked, you can run the following statement to identify the table or index that's being locked, leading to a slow database:

```
SQL> select l.object_id,l.session_id,
  2  l.oracle_username,l.locked_mode,
  3  o.object_name
  4  from v$locked_object l,
  5  dba_objects o
  6* where o.object_id=l.object_id;
OBJECT_ID  SESSION_ID ORACLE_USERNAME  LOCKED_MODE  OBJECT_NAME
 6699       22         NICHOLAS           6        EMPLOYEES
SQL>
```

As the preceding query and its output show, user Nicholas has locked up the Employees table. If this is preventing other users from accessing the table, you have to quickly remove the lock by killing the locking user's session. You can get the session_id from the preceding output and the V$SESSION view will give you the serial# that goes with it. Using the *alter system kill …* command, you can kill the offending session. The same analysis applies to a locked index, which will prevent users from using the base table. For example, an attempt to create an index or rebuild it when users are accessing the table can end up inadvertently locking up the table.

If there is a table or index corruption, that could cause a problem with accessing that object(s). You can quickly check for corruption by running the following statement:

```
SQL> analyze table employees validate structure cascade;
Table analyzed.
SQL>
```

Abnormal Increase in Process Size

On occasion, there might be a problem because of an alarming increase in the size of one or more Oracle processes. You have to be cautious in measuring Oracle process size, because traditional UNIX operating system–based tools can give you a misleading idea about process size. The following sections explain how to measure Oracle process memory usage accurately.

What Is Inside an Oracle Process?

An Oracle process in memory has several components:

- *Shared memory:* This is the SGA that you are so familiar with.
- *The executable:* Also known as TEXT, this component consists of the machine instructions. The TEXT pages in memory are marked read-only.
- *Private data:* Also called DATA or heap, this component includes the PGA and the UGA. The DATA pages are writable and aren't shared among processes.
- *Shared libraries:* These can be private or public.

When a new process starts, it requires only the DATA (heap) memory allocation. Oracle uses the UNIX implementation of shared memory. Shared memory means that all processes attach to shared memory segments. The SGA and TEXT are visible to and shared by all Oracle processes, and they aren't part of the cost of creating new Oracle processes. If 1,000 users are using Oracle Forms, only one set of TEXT pages is needed for the Forms executable.

Unfortunately, most operating system tools such as ps and top give you a misleading idea as to the process size, because they include the common shared TEXT sizes in individual processes. Sometimes they may even include the SGA size. Solaris's pmap and HP's glance are better tools from this standpoint, as they provide you with a more accurate picture of memory usage at the process level.

NOTE *Even after processes free up memory, the operating system may not take the memory back, indicating larger process sizes as a result.*

Measuring Process Memory Usage

As a result of the problems you saw in the previous section, it's better to rely on Oracle itself for a true indication of its process memory usage. If you want to find out the total DATA or heap memory size (the biggest nonsharable process memory component), you can do so by using the following query:

```
SQL> select value, n.name|| '('||s.statistic#||')' , sid
     from v$sesstat s, v$statname n
     where s.statistic# = n.statistic#
     and n.name like '%ga memory%'
     order by value;
```

If you want to find out the total memory allocated to the PGA and UGA memory together, you can issue the following command. The query reveals that a total of over 367MB of memory is allocated to the processes. Note that this memory is in addition to the SGA memory allocation, so you need to make allowances for both types of memory to avoid paging and swapping issues.

```
SQL> select sum(value)
     from v$sesstat s, v$statname n
     where s.statistic# = n.statistic#
     and n.name like '%ga memory%';
SUM(VALUE)
---------------
3674019536
1 row selected.
SQL>
```

If the query shows that the total session memory usage is growing abnormally over time, you might have a problem such as a memory leak. A telltale sign of a memory leak is when Oracle's memory usage is way outside the bounds of the memory you've allocated to it through the initialization parameters. The Oracle processes fail to return the memory to the operating system in this case. If the processes continue to grow in size, eventually they may hit some system memory barriers and fail with the ORA-4030 error:

```
[finance1] $ oerr ora 4030
04030, 00000, "out of process memory when trying to allocate %s bytes (%s,%s)"
// *Cause:  Operating system process private memory has been exhausted
[finance1] $
```

Note that Oracle tech support may request that you collect a *heap dump* of the affected Oracle processes (using the oradebug tool) to fix the memory leak problem.

If your system runs out of swap space, the operating system can't continue to allocate any more virtual memory. Processes will fail when this happens, and the best way to get out of this mess is to see if you can quickly kill some of the processes that are using a heavy amount of virtual memory.

Delays Due to Shared Pool Problems

Sometimes, database performance deteriorates dramatically because of inadequate shared pool memory. Low shared pool memory relative to the number of stored procedures and packages in your database could lead to objects constantly aging out of the shared pool and having to be executed repeatedly.

Problems Due to Bad Statistics

As you know by now, the Oracle cost-based optimizer (CBO) needs up-to-date statistics so it can pick the most efficient method of processing queries. It is common for most DBAs to analyze tables and indexes on a regular basis. Of course, you can also analyze the tables in a more efficient way by using the DBMS_STATS package.

If you don't execute the DBMS_STATS package (or run the *analyze* command) regularly while lots of new data is being inserted into tables, your old statistics will soon be out of whack, and the performance of critical SQL queries could head south. DBAs are under time constraints to finish the analyze of table overnight or over a weekend. Sometimes, they may be tempted to use the *estimate* option to analyze the tables. As I advised earlier in this chapter, always do a full analyze of tables, with the *compute* option. The *estimate* option is totally unreliable, and it leads to the generation of bad statistics. The database could slow down considerably as a result.

Collecting Information During a Database Hang

It sometimes can be downright chaotic when things crawl down to a standstill in the database. You might be swamped with phone calls and anxious visitors to your office who are wondering why things are slow. Oftentimes, especially when serious unknown locking issues are holding up database activity, it's tempting to just bounce the database because usually it clears up the problem. Unfortunately, you really don't know what caused the problem, so when it happens again, you're still just as ignorant as you were the first time around regarding the cause of the problem. Bouncing the database also means that all the users currently connected will be forced out of the database, which may not always be a smart strategy.

It's important that you collect some information quickly for two reasons. First, you might be able to prevent the problem next time or have someone in Oracle tech support (or a private firm) diagnose the problem using their specialized tools and expertise in these matters. Second, most likely a quick shutdown and restart of the database will fix the problem for sure (as in the case of some locking situations, for example). But a database bounce is too mighty a weapon to bring to bear on every similar situation. If you diagnose the problem correctly, simple measures may prevent the problem or help you fix it when it does occur. The following sections describe what you need to do to collect information on a very slow or hanging database.

Gathering Error Messages

The first thing you do when you find out the database suddenly slowed down or is hanging is to look in some of the log files where Oracle might have sent a message to. Quickly look in the alert log file to see if there are any Oracle error messages or any other information that could pinpoint any problems. You can check the background dump directory (bdump) for any other trace files with error messages in them. I summarize these areas in the following discussion.

Getting a Systemstate Dump

A systemstate dump is simply a trace file that is output to the user dump directory. Oracle (or a qualified expert) can analyze these dumps and tell you what was going on in the database when the hanging situation occurred. For example, if logons are very slow, you can do a systemstate dump during this time, and it may reveal that most of the waits are for a particular type of library cache latch. These dumps tend be very large, so make sure your *user_dump_dest* parameter is bumped up to a large value (such as 200MB) if it is set to a small value right now. To get a systemstate dump, run the following command:

```
SQL> alter session set events 'immediate trace name systemstate, level 10';
```

CAUTION *Oracle Corporation strongly warns against customers setting events on their own. You may end up causing more severe problems when you set events sometimes. Please contact Oracle technical support before you set any event. For example, the event 10235 has been known to cause heavy latch contention.*

You can send the resulting output to Oracle so they can analyze it for you. Note that at this stage, you need to open a technical assistance report (TAR) with Oracle technical support through MetaLink (`http://metalink.oracle.com`). (The hanging database problem will get you a priority level 1 response, so you should hear from an analyst within minutes.) Oracle technical support may ask you for more information, such as a core dump, and ask you to run a debugger or another diagnostic tool and FTP the output to them.

Using the Hanganalyze Utility

The systemstate dumps, while useful, have several drawbacks, including the fact that they dump out too much irrelevant information and take too much to complete the dump, leading to inconsistencies in the dump information. The newer hanganalyze utility is more sophisticated than a systemstate dump. Hanganalyze provides you with information on resources each session is waiting for and what is blocking access to those resources. The utility also provides you with a dependency graph among the active sessions in the database. This utility is not meant to supplant the systemstate dumps; rather, you should use it to help make systemstate dumps more meaningful. Again, use this utility in consultation with Oracle technical support experts. Here is a typical *hanganalyze* command:

```
SQL> alter session set events 'immediate trace name HANGANALYZE level 3';
```

The Promise and the Performance

The Immigration and Naturalization Service (INS) of the United States government created a new $36 million Student and Exchange Visitor Information System (SEVIS) to replace the old paper-based methods the INS had used for years to track foreign students in U.S. educational institutions. More than 5,400 high schools, colleges, and universities have to use SEVIS to enter the necessary information about enrolled students from other countries.

The INS had imposed a deadline of January 31, 2003, by which all educational institutions had to switch over fully to the SEVIS system. However, it extended the deadline by at least 2 weeks amidst several complaints about the system working slowly, if at all. Here are a few of those complaints from users across the country:

- Some employees in Virginia could enter data only in the mornings, before the West Coast institutions logged onto the system. In the afternoons, the system slowed to a crawl.
- From the University of Minnesota came complaints that that the officials were "completely unable" to use the system at all. The users mentioned that the system "was really jammed with users trying to get on." They also complained that the system was "unbelievably slow." An INS spokesperson admitted that the system had been "somewhat sluggish" and that schools were having trouble using the SEVIS system.
- The University of North Carolina complained that the situation, if it continued any further, was going to be "a real nightmare" and that it was already "starting to cause some problems."
- One worker at a college in Michigan was quoted as saying this in frustration: "Please tell me what I'm doing wrong, or I am going to quit."

The INS realized the colleges and universities weren't going to meet the deadline, and they announced a grace period after saying that "upgrades to the system" had greatly improved performance.

Why am I discussing INS's problem in this chapter? Well, behind the SEVIS system is an Oracle9*i* database, which was performing awfully slowly. The system apparently couldn't scale well enough. When large number of users got on, it basically ground to a halt. Obviously, the system wasn't configured to handle a high number of simultaneous operations. Was the shared server approach considered, for example? How were the wait statistics? I don't know the details. I do know that the Oracle9*i* database is fully capable of meeting the requirements of an application such as this. I picked this example to show that even in high-profile cases, DBAs sometimes have to eat humble pie when the database isn't tuned properly and consequently performance doesn't meet expectations.

A Simple Approach to Instance Tuning

Most of the instance tuning that DBAs perform is in response to a poorly performing database. Although a well-designed system with well-written SQL code might avoid many performance issues, here you are, on a busy day, with your customers complaining about slow performance. What do you do? The following sections present a brief summary of how you can start analyzing the instance to find out where the problem lies.

First, examine all the major resources such as the memory, CPUs, and storage subsystem to make sure your database isn't being slowed down by bottlenecks in these critical areas.

NOTE *Collecting baseline data about your database statistics, including wait events, is critically important for troubleshooting performance issues. If you don't have baseline numbers, what are you going to compare the present system data with? If you have baseline data, you can immediately check if the current resource usage patterns are consistent with the load on the system.*

What's Happening in the Database?

It isn't rare for a single user's SQL query to cause an instancewide deterioration in performance if the query is bad enough. SQL statements are at the root of all database activity, so you should look at what's going on in the database right now. The following are some of the key questions you need to find answers to:

- Who are the top users in your TopSessions display?
- What are the exact SQL statements being executed by these users?
- Is the number of users unusually high compared to your baseline numbers for the same time period?
- Is the load on the database higher than what your baseline figures show for the time of the day or the time of the week or month? You can judge loads fairly quickly by using the V$SYSSTAT view to find out the number of logical reads, physical reads and writes, parse counts (hard and soft), DB block changes, and the size of the undo that is being generated.
- What top waits can you see in the V$SESSION_WAITS view? This is a real-time wait event view that shows the wait events that are happening right now or have just happened in the instance. You have already seen how you can find out the actual users responsible for the waits by using other V$ views.

My colleague Don Rios correctly points out that a critical question here is whether the performance problem du jour is something that is "sudden" without any forewarnings or if it is caused by factors that have been "gradually" creeping up on you. Under the latter category are things such as a growing database, a larger number of users, and a larger of number of DML operation updates than what you had originally designed the system for. These types of problems may mean that you need to redesign at least some of your tables and indexes, with different storage and other parameters such as FREELISTS. If, on the other hand, the database has slowed down suddenly, you need to focus your attention on a separate set of items.

Are There Any Long-Running Transactions?

You can use the V$SQL view as shown in the following example to find out which of the SQL statements in the instance are currently taking the most time to finish and are the most resource intensive. The query ranks the transactions on the total number of elapsed seconds. You can also rank the statements according to CPU seconds used.

```
SQL> select hash_value, executions,
  2  round (elapsed_time/1000000, 2) total_time,
  3  round (cpu_time/1000000, 2) cpu_seconds from (
  4  select * from v$sql order by elapsed_time desc);
HASH_VALUE EXECUTIONS   TOTAL_TIME  CPU_SECONDSS
---------- ----------    ---------- ------------
 238087931    168          9.51       9.27
1178035321    108          4.98       5.01
…
SQL>
```

Once you have the value for the hash_value column from the query you just ran, it's a simple matter to find out the execution plan for this statement, which is in your library cache. The following query uses the V$SQL_PLAN view to get you the execution plan for your longest running SQL statements.

```
SQL> select * from V$SQL_PLAN where hash_value = 238087931;
```

You can also use the hash value from the previous example to get a Statspack SQL report. To get the SQL report, you need to execute the sprepsql.sql script, which is located in the $ORACLE_HOME/rdbms/admin directory. The script will prompt you for the *begin* and *end* snapshot IDs (2 and 4 in the snapshot example), as well as the hash value for a single SQL statement. Statspack will provide you with the execution plan used by the SQL statement and a detailed accounting of the resources used during the execution of the statement. Statspack will place the output in a report for you. Here's an example:

```
STATSPACK SQL report for Hash Value: 238087931
DB Name         DB Id    Instance     Inst Num Release  Cluster Host
------------ ----------- ------------ -------- ---------- ------- -
MONITOR       2029096430 monitor          1   9.2.0.1.0   NO     hp5
 Start Id  Start Time       End Id    End Time     Duration(mins)
--------- ------------------ --------- ------------------ --------
     1    7-Feb-03 09:13:45   2    17-Feb-03 09:26:24   12.65
```

The report then provides detailed execution statistics for the SQL statement you specified, as shown here:

```
-> CPU and Elapsed Time are in seconds (s) for Statement Total and in
   milliseconds (ms) for Per Execute
                                                       % Snap
                     Statement Total      Per Execute   Total
                     ---------------  ---------------  ------
        Buffer Gets:          10,038            418.3   63.33
         Disk Reads:               0              0.0     .00
     Rows processed:             528             22.0
     CPU Time(s/ms):               1             47.9
 Elapsed Time(s/ms):               1             46.4
              Sorts:               0               .0
        Parse Calls:              24              1.0
      Invalidations:               0
      Version count:               1
    Sharable Mem(K):              23
```

The SQL report will also show you the execution plan for the SQL statement that was found in the library cache during the begin and end snapshots of Statspack.

Is Oracle the Problem?

Just because your database users are complaining, you shouldn't be in a hurry to conclude that the problem lies within the database. After all, the database doesn't work in a vacuum—it runs on the server and is subject to the resource constraints and bottlenecks of that server. If the non-Oracle users on the server are using up critical resources such as CPU processing and disk I/O, your database may be the victim of circumstances, and you need to look for answers outside the database. That's why it's critical that DBAs understand how to measure general system performance, including memory, the disk storage subsystem, the network, and the processors. In the following sections you'll take a look at the system resources you should focus on.

Is the Network Okay?

One of the first things you need to do when you're investigating slowdowns is to rule out network-related problems. Quite often, users complain of their being unable to connect to or being abruptly disconnected from the system. Check your round-trip ping times and the number of collisions. Your network administrator should check the Internet connections and routers.

On the Oracle end, you can check the following dynamic views to find out if there is a slowdown due to a network problem. The V$SESSION_EVENT view shows the average amount of time Oracle waits between messages, in the average wait column. The V$SESSION_WAIT view, as you've seen, shows what a session is waiting for, and you can see if waits for network message transport are higher than normal.

If the time for SQL round-trips is extremely long, it could reflect itself as a high amount of network-related wait time in the V$ views. Check to see if your ping time for network round-trips has gone up appreciably. You should discuss with your network administrator what you can do to decrease the waits for network traffic.

You may explore the possibility of setting the parameter *tcp.nodelay=true* in your sqlnet.ora file. This will result in TCP sending packets without waiting, thus increasing response time for real-time applications.

If the network seems like one of your constant bottlenecks, you may want to investigate the possibility of using the shared server approach instead of the dedicated server approach for connecting users to your database. This will help your application scale more efficiently to very large user bases.

Is the System CPU Bound?

Check the CPU performance to make sure a runaway process or a valid Oracle process is not hogging one or more processes and contributing to the system slowdown. Often, killing the runaway processes or the resource-hogging sessions will bring matters to a more even keel. Using OEM, you can get a quick idea about the breakdown of CPU usage among parse, recursive, and other usage components, as shown in Figure 19-4.

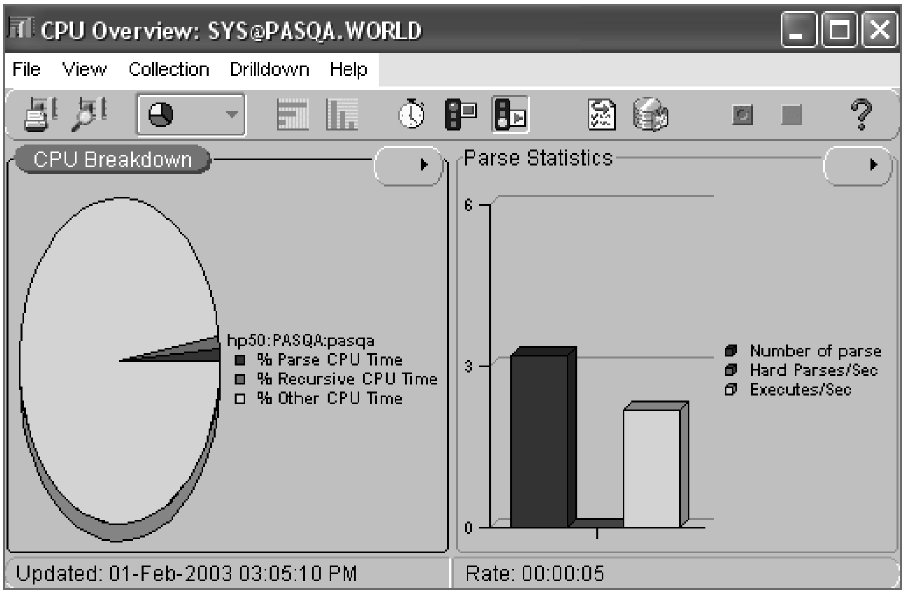


Figure 19-4. Using OEM to analyze CPU usage

Normally, you should expect to see no more than 20 to 25 percent of total CPU usage by the system itself, and about 60 to 65 percent usage by the Oracle application. If the system usage is close to 50 percent, it is an indication that there are too many system calls, for example, which leads to excessive use of the processors.

As you learned earlier in this chapter, the V$SESSTAT view shows CPU usage by session. Using the following query, you can find out the top CPU-using Oracle sessions. You may want to look into the actual SQL that these sessions are executing.

```
SQL> select a.sid,a.username,
  2  s.sql_text
  3  from v$session a,v$sqltext s
  4  where a.sql_address = s.address
  5  and a.sql_hash_value = s.hash_value
  6  and a.username  = '&USERNAME'
  7  AND A.STATUS='ACTIVE'
  8* order by a.username,a.sid,s.piece;
```

You can also view the Oracle Performance Manager's Database Health Overview Chart to examine CPU usage and see if a few users are using up most of the available processing resources. As indicated in earlier sections of this chapter, you should break down the CPU usage into parse, recursive, and other types of usage. If the parse usage is predominant, for example, you know that the database is performing excessive parsing for some reason. High CPU usage is not necessarily a problem; it is the type of use the CPU is being put to that is more important. Figure 19-5 shows a database that has a large number of waits, with the total wait time outweighing the total CPU time taken to execute the queries in the database.

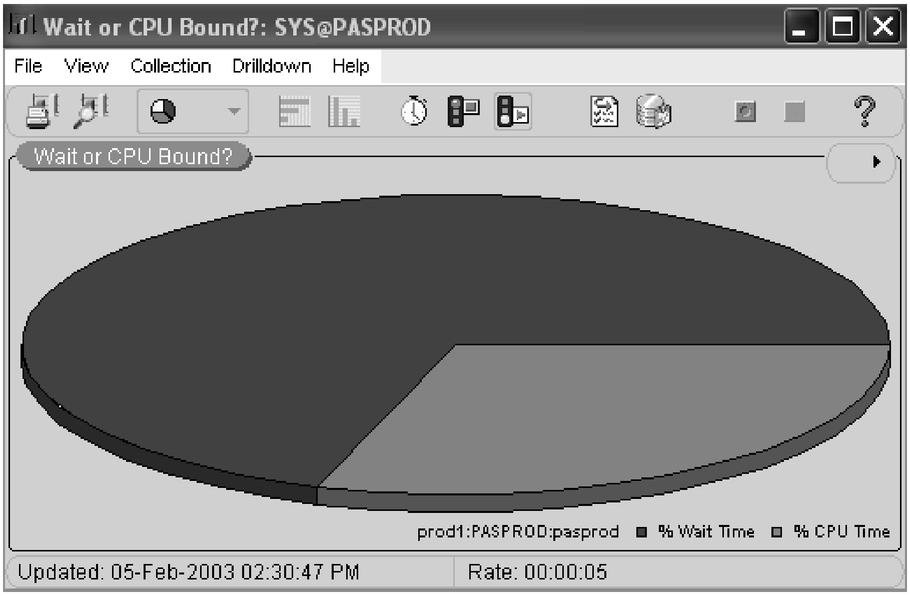


Figure 19-5. Using the OEM Performance Manager to examine if the system is CPU or wait bound

Is the System I/O Bound?

Before you go any further analyzing other wait events, it's a good idea to rule out that you are not limited by your storage subsystem by checking your I/O situation. Are the read and write times on the host system within the normal range? Is the I/O evenly distributed, or are there hot spots with one or two disks being hit really hard? If your normal, "healthy" I/O rates are 40–50/ms and you're seeing an I/O rate of 80/ms, obviously something is amiss. Figure 19-6 shows how you can view I/O times (disk read and disk write) times by data file. This will usually tip you off about what might be causing the spike. For example, if the temporary tablespace data files are showing up in the high I/O list often, that's usually an indication that disk sorting is going on, and you need to investigate that further.

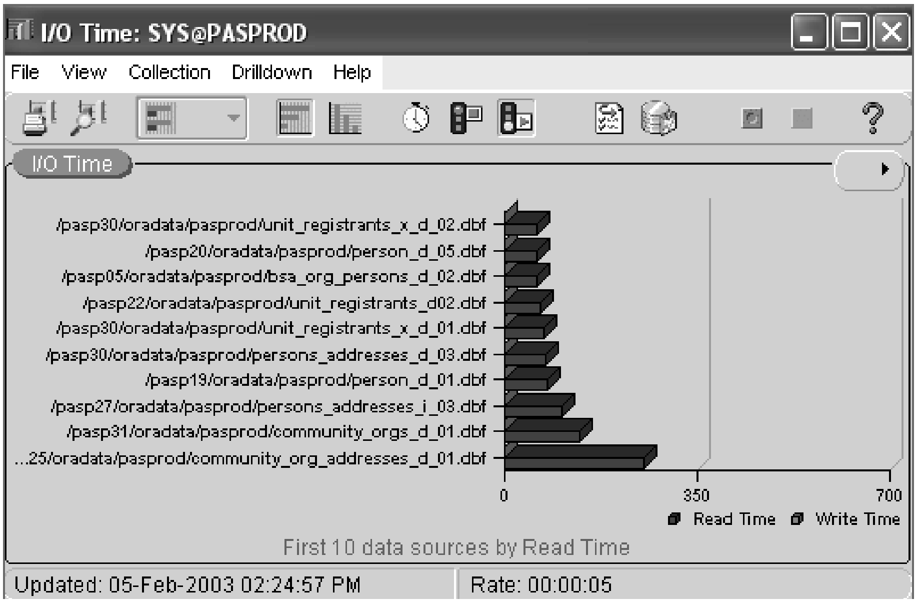

I/O Time: SYS@PASPROD
File
View
Collection
Drilldown
Help
I/O Time
/pasp30/oradata/pasprod/unit_registrants_x_d_02.dbf
/pasp20/oradata/pasprod/person_d_05.dbf
/pasp05/oradata/pasprod/bsa_org_persons_d_02.dbf
/pasp22/oradata/pasprod/unit_registrants_d02.dbf
/pasp30/oradata/pasprod/unit_registrants_x_d_01.dbf
/pasp30/oradata/pasprod/persons_addresses_d_03.dbf
/pasp19/oradata/pasprod/person_d_01.dbf
/pasp27/oradata/pasprod/persons_addresses_i_03.dbf
/pasp31/oradata/pasprod/community_orgs_d_01.dbf
...25/oradata/pasprod/community_org_addresses_d_01.dbf
0
350
700
Read Time
Write Time
First 10 data sources by Read Time
Updated: 05-Feb-2003 02:24:57 PM
Rate: 00:00:05

Figure 19-6. Top ten I/O sources in the database

Too often, a batch program that runs into the daytime could cause spikes in the I/O rates. Your goal is to see if you can rule out the I/O system as the bottleneck. Several of the wait events that occur in the Oracle database, such as the DB file sequential read and DB file scattered read waits, can be the result of extremely heavy I/O in the system. If the average wait time for any of these I/O related events is significant, you should focus on improving the I/O situation. There are two things you can do to increase the I/O bandwidth: Reduce the I/O workload or increase the I/O bandwidth. In Chapter 18 you learned how you can reduce physical I/Os by proper indexing strategies and the use of efficient SQL statements.

Improving SQL statements is something that can't happen right away, so there are other things you need to do to help matters in this case. This means you need to increase to increase the I/O bandwidth by doing either or both of the following:

- Make sure that the key database objects that are used heavily are spread evenly on the disks.
- Increase the number of disks.

Storage disks are getting larger and larger in terms of their capacity, but the I/O rates aren't quite keeping up with the increased disk sizes. Thus, servers are frequently I/O bound in environments with large databases. Innovative techniques such as file caching might be one solution to a serious I/O bottleneck. The idea behind the file cache accelerators is simple: Because on average, about 50 percent of I/O activity involves less than 5 percent of the total data files in your database, why not cache these limited number of "hot files"? Caching gives you the benefit of read/write operations at memory speeds, which could be 200 times faster than a mechanical disk. What are good candidates for the list of hot files? You can include your temp, redo log, and undo tablespace files as well as the most frequently used table and index data files on file cache accelerators. For a state-of-the-art file caching accelerator, please visit the Imperial Technology Web site (`http://www.imperialtech.com`).

Checking Memory-Related Issues

As you saw earlier in this chapter, high buffer cache and shared pool hit ratios are not guarantees of efficient instance performance. Sometimes, an excessive preoccupation with hit ratios can lead you to allocate too much memory to Oracle, which opens the door to serious problems such as paging and swapping at the operating system level. Make sure that the paging and swapping indicators don't show anything abnormal. High amounts of paging and swapping will slow down everything, including the databases on the server.

Due to the virtual memory system used by most operating systems, a certain amount of paging is normal and to be expected. If physical memory isn't enough to process the demand for memory, the operating system will go the disk to use its virtual memory, and this results in a page fault. Processes that result in high page faults are going to run very slowly.

When it comes to Oracle memory allocation, don't forget to pay proper attention to PGA memory allocation, especially if you're dealing with a DSS-type environment. The database will self-tune the PGA, but you still have to ensure that the *pga_aggregate_target* value is high enough for Oracle to perform its magic.

See if you can terminate a few of the top sessions that seem to be consuming inordinate amounts of memory. It's quite possible that some of these processes are orphan or runaway processes.

Is the System Wait Bound?

If none of the previous steps indicated any problems, chances are that your system is suffering from a serious contention for some resource such as library cache latches. Check to see if there is contention for critical database resources such as locks and latches, for example. Contention for these resources manifests itself in the form of wait events. The wait event analysis earlier in this chapter gave you a detailed explanation of various critical wait events. You can use the output of the Statspack utility to see what the top wait events in your database are. A quick-and-dirty way to find out what waits are causing the system slowdown is to run the query shown in Listing 19-30.

Listing 19-30. Current Wait Events in the Instance

```
SQL>  select event,count(*) from v$session_wait
  2*  group by event;
EVENT                                COUNT(*)
----------------------------------------------------------------
PL/SQL lock timer                           1
SQL*Net message from client               569
SQL*Net message to client                   2
SQL*Net more data from client               2
buffer busy waits                           1
db file scattered read                      2
db file sequential read                     7
latch free                                  1
8 rows selected.
SQL>
```

An even more efficient method of viewing current waits in the system is to use OEM's Performance Manager. For example, if the Latch Analysis Chart shows much more than 1 percent latch contention in the system, you may have some type of latch issue. Figure 19-7 shows how to use OEM's Performance Manager to examine the latch wait levels in your database.

You can also use the Performance Manager to view the Wait Analysis Overview tool, which shows you summary information on various wait events in the database. You can view a chart showing the number of sessions currently waiting for a specific event. You can also look at charts that show the top five wait events that have finished waiting for the sample period. OEM's wait analysis screen will also show you the top waits by time waited, as shown in Figure 19-8.

Eliminating the Contention

Once you identify wait events due to contention in the system, you need to remove the bottleneck. Of course, this is easier said than done in the short run. You may be able to fix some contention problems right away, whereas you may need more time with others. Problems such as high DB file scattered read events, which are due to full table scans, may indicate, as you have seen, that the I/O workload of the system needs to be reduced. However, if the reduction in I/O requires creating new indexes and rewriting SQL statements, obviously you can't fix the problem right away. You can't add disks and rearrange objects to reduce hot spots right away either. Similarly, most latch contention requires changes at the application level. Just make sure you don't perform a whole bunch of changes at once—you'll never be able to find out what fixed the problem (or in some cases, what made it worse!).

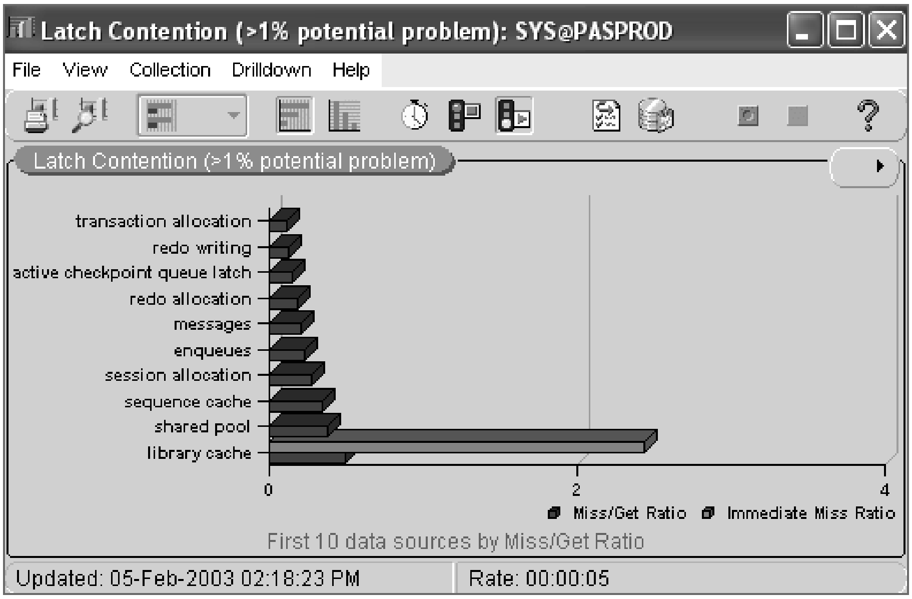

Latch Contention (>1% potential problem): SYS@PASPROD
File View Collection Drilldown Help
Latch Contention (>1% potential problem)
transaction allocation
redo writing
active checkpoint queue latch
redo allocation
messages
enqueues
session allocation
sequence cache
shared pool
library cache
0
2
4
Miss/Get Ratio
Immediate Miss Ratio
First 10 data sources by Miss/Get Ratio
Updated: 05-Feb-2003 02:18:23 PM
Rate: 00:00:05

Figure 19-7. Using the Performance Manager to examine latch waits

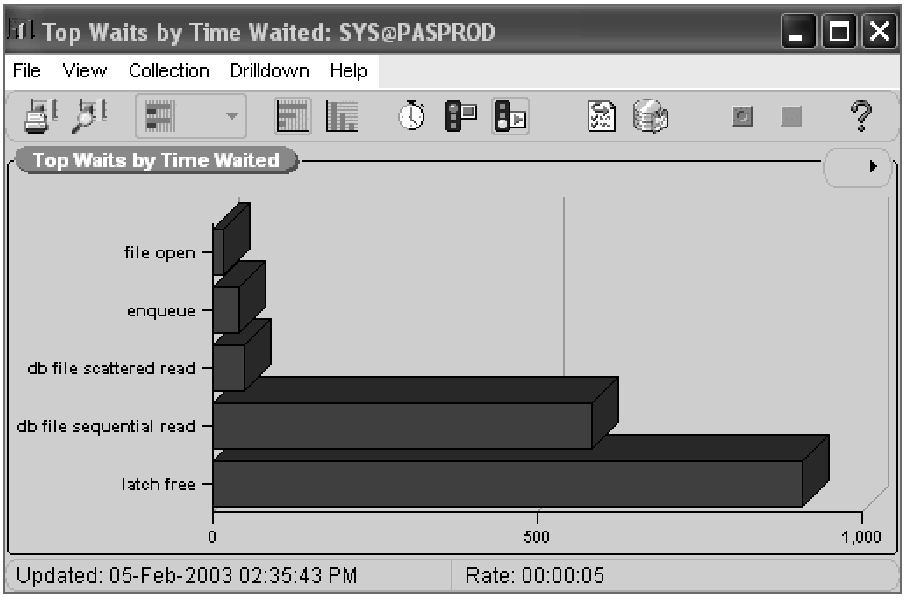


Figure 19-8. Top Waits by Time Waited analysis

The trick, as usual, is to go after the problems you can fix in the short run. Problems that you can fix by changing the memory allocation to the shared pool or the buffer cache you can easily handle almost immediately by dynamically adjusting the cache values. You can also take care of any changes that concern the redo logs right away. If you notice one or two users causing a CPU bottleneck, it may be a smart idea to kill those sessions so the database as whole will perform better. As you know, prevention is much better than a cure, so consider using the Oracle Database Resource Manager tool (Chapter 11 shows you in detail how to use the Database Resource Manager) to create resource groups and prevent a single user or group from monopolizing the CPU usage.

If intense latch contention is slowing your database down, you probably should be setting the *cursor_sharing* parameter's value to force or similar to ameliorate the situation.

Most other changes, though, may require more time-consuming solutions. Some changes may even require major changes in the code or the addition or modification of important indexes. However, even if the problem isn't fixed immediately, you have learned your craft, and you're on the right path to improving instance performance.

Summary

In an operational database, the scope for SQL tuning is extremely limited in most cases, unless the poor SQL code is bringing the entire system to a standstill. Most of the time, you have to work your way around less-than-optimal SQL code. As a DBA, you can impact instance performance considerably, however, and it's important to know how to evaluate system performance.

This chapter provided you with detailed analyses of database hit ratios and Oracle wait events. The issue of whether you should use the traditional hit ratios or the more sophisticated wait ratio analysis isn't hard to resolve. You need to look at both the hit ratios and wait ratios to assess performance problems. In some cases, the problem may be elusive, even when you employ both of these performance indicators. A better methodology is sometimes to simply focus on the top resource-consuming SQL statements in your system and see if you can tune them. Remember that in order to reduce the response time of your users, you'll need to bring down the wait time involved. If high wait times are causing response times to go down, you need to focus intensely on the type of waits and what's causing them. Just make sure you don't mistake the symptoms of high wait times for the maladies themselves.

The chapter showed you how to handle a database hang and offered a systematic methodology for you to follow when your database response times are slow.