

CHAPTER 3

Scalable Architecture



his chapter takes a look at some of the architectural (macro-level) decisions that can help (or hinder) you in achieving the sort of scalability and performance of which Oracle is capable. We will look at three main goals:

- **Achieving high concurrency** Dedicated server versus shared server connections and clustering
- **Handling storage space effectively** Locally managed tablespaces and partitioning
- **Taking advantage of parallel operations** Parallel processing with Oracle's various options for parallel operations

I will not go into the details of the Oracle server process architecture and memory structures, or explain how to set up partitioning or parallel processing. (For that information, I will refer you to the relevant documentation.) Rather, I will focus on how and when to use these Oracle's features, and also attempt to dispel some myths about them.

Understand Shared Server Versus Dedicated Server Connections

Using a dedicated server configuration is by far the most common method to connect to an Oracle database. It is the easiest to configure and offers many benefits when debugging or tracing a process. It is the connection option most people should be using day to day. A shared server configuration is more complex configuration, but it can help you get more sessions connected using the same hardware (number of CPUs and memory).



NOTE

Shared server configuration is also known as multithreaded server, or MTS, in earlier releases of Oracle. MTS and shared server are synonymous. This book uses the term shared server.

In this section, I'll briefly explain how the dedicated server and shared server connections work under the covers, and cover the pros and cons of each configuration. Then I'll deal with some common misconceptions with regard to the use of a shared server configuration.

"Should I be using the shared server configuration of Oracle?"

Unless you have more concurrent connections to your database than your operating system can handle, do not use shared server. Until the machine is so loaded with processes/threads that it cannot hold another one, use dedicated server. Then, when the machine hits that maximum threshold, where another connection is either not possible or adversely affects overall performance, that is the time to consider switching over to shared server. Fortunately, clients don't have a control over this, so switching back and forth is trivial to accomplish and transparent to the application.

There are actually many reasons why dedicated server should be your first choice, but the main one is that on a system where dedicated server has not yet overloaded the system with processes/threads, *it will be faster*. So, unless dedicated server is the cause of your performance issues (system overload), shared server will not perform as well.

How Do Dedicated Server Connections Work?

When you connect to an Oracle instance, in reality, you are connecting to a set of processes and shared memory. The Oracle database is a complex set of interacting processes that all work together in order to carry out requested operations.

Dedicated Server Connection Steps

When you connect with a dedicated server over the network, the following steps take place:

1. Your client process connects over the network to a listener that is typically running on the database server. There are situations (see the “Take Advantage of Clustering” section later in this chapter for an example) where the listener will not be on the same machine as the database server instance.
2. The listener creates a new dedicated process (Unix) or requests the database to create a new dedicated thread (Windows) for your connection. The choice of whether a thread or process will be created is decided by the Oracle kernel developers on an operating system-by-operating system basis. They will choose the implementation that makes the most sense on that particular operating system.
3. The listener hands the connection off to the newly created process or thread.
4. Your client process sends requests to this dedicated server and it processes them, sending the results back.

Figure 3-1 shows how dedicated server configuration works. The Oracle instance will be a single process with threads on some platforms (Windows, for example) or each “bubble” will be a separate physical process (Unix, for example). In a dedicated server configuration, each client has its own process/thread associated with it.

The dedicated server will receive your SQL and execute it for you. It will read datafiles and place data in the cache, look in the database cache for data, perform your UPDATE statements, and run your PL/SQL code. Its only goal is to respond to the SQL calls that you submit to it.

So, in dedicated server mode, there is a one-to-one mapping between processes in the database and a client process. Every dedicated server connection has a dedicated server allocated solely to it. That does not mean every single session has a dedicated server, it is possible for a single application, using a single physical connection (dedicated server) to have many sessions concurrently active. In general however, when in dedicated server mode, there is a one to one relationship between a session and a dedicated server.

Pros and Cons of Dedicated Server Connections

Here is a short list of the advantages of using a dedicated server connection to the database:

- It is easy to set up. In fact, it requires almost no setup.

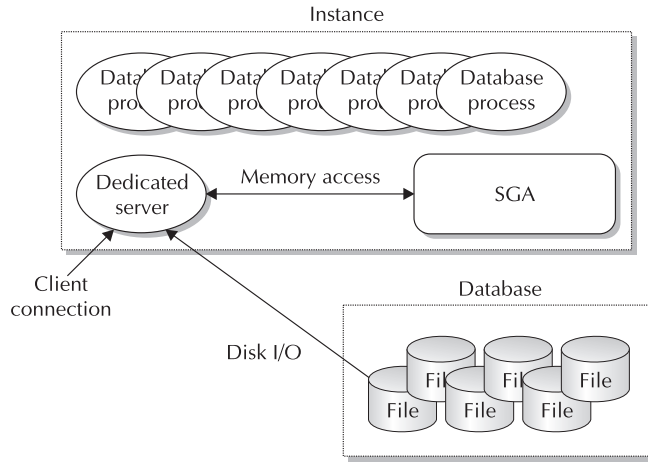


FIGURE 3-1. *The connection process in dedicated server configuration*

- It is the fastest mode for the database to operate in, using the shortest code path.
- Since trace files are tied to a process (a dedicated server, for example), facilities such as SQL_TRACE benefit from dedicated server mode. In fact, trying to use SQL_TRACE on a nondedicated server mode connection can sometimes be an exercise in futility.
- All administrative functions are available to you. (In shared server mode, certain administrative features—such as database startup, shutdown, and recovery features—are not available.)
- The memory for the user's user global area (UGA), which is session-specific memory, is allocated in the process global area (PGA) dynamically. It need not be configured in the shared global area (SGA).

Some of the disadvantages of using dedicated server connections are as follows:

- The client has a process/thread consuming resources (memory, CPU cycles for context switching, and so on) on the server from the time the session begins until it terminates.
- As the number of users increases, the operating system can become overloaded managing that many processes/threads.
- You have limited control over how many sessions can be active simultaneously, although Oracle9i has addressed this with the Resource Manager and the maximum active sessions per group rule.
- You cannot use this connection mode for database link concentration.

- If you have a high rate of actual connects/disconnects to the data, the overhead of process and thread creation and destruction may be very large. A shared server connection would have an advantage here. Note that for most web-based applications, this is not an issue, because they typically perform some sort of connection pooling on their own.

As you can see from these lists, for most systems, the dedicated server benefits outweigh many of the disadvantages. On most conventional database servers today (typically, machines with two to four CPUs), you would need 200 to 400 concurrent connections before considering a different connection type. On larger machines, the limit is much higher.

There is no set limit at which a dedicated server configuration becomes infeasible. I've seen the need to stop using this configuration with 100 users. On the other hand, I've seen systems running with more than 1,000 users using dedicated server configuration without breaking a sweat. Your hardware and how you use the database will dictate your decision to switch to shared server connections. As long as you have the CPU and, to a less extent, the RAM, dedicated server configuration is the easiest and most functional approach.

How Do Shared Server Connections Work?

The connection protocol for a shared server connection is very different from the one for a dedicated server configuration. In a shared server configuration, there is no one-to-one mapping between clients (sessions) and server processes/threads. Rather, there is a pool of aptly named *shared servers* that perform the same operations as a dedicated server, but they do it for multiple sessions as needed, instead of just one session.

Additionally, there are processes known as *dispatchers* that run when you are using a shared server connection. The clients will be connected to a dispatcher for the life of their session, and this dispatcher will facilitate the handing off of a client request to a shared server and getting the answer back to the client after the shared server is finished. These dispatchers running on the database are known to the listener process. As you connect, the listener will redirect you to an available dispatcher.

Shared Server Connection Steps

When you connect via shared server connections, here is what typically happens:

1. Your client process connects over the network to a listener running on the database server. This listener will choose a dispatcher from the pool of available dispatchers for you to connect to.
2. The listener will send back to your client the address of the chosen dispatcher for you to connect to, or you may be redirected directly to the dispatcher (in some cases, the connection may be handed off directly). This happens at the TCP/IP-connection level and is operating-system dependent. If you were not directly handed off to the dispatcher, your client disconnects from the listener and your client connects to the dispatcher.
3. Your client process sends request to this dispatcher.

Figure 3-2 illustrates the connection process in shared server mode.

So, as connection requests are received, first, the listener will choose a dispatcher process from the pool of available dispatchers. Next, the listener will send back to the client connection

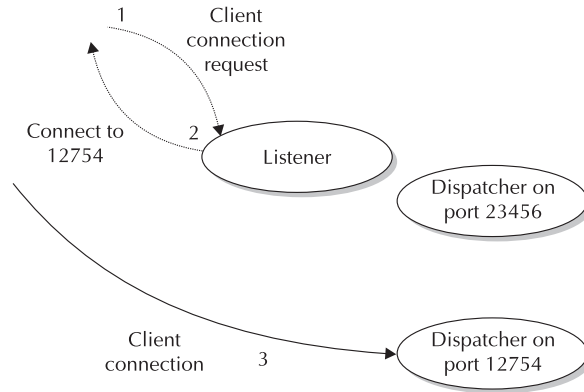


FIGURE 3-2. *The connection process in shared server configuration*

information describing how the client can connect to the dispatcher process. This must be done because the listener is running on a well-known hostname and port on that host, but the dispatchers will be accepting connections on randomly assigned ports on that server. The listener is aware of these random port assignments and picks a dispatcher for you. The client then disconnects from the listener and connects directly to the dispatcher. Finally, you have a physical connection to the database.

Shared Server Command Processing

Now that you are connected, the next step is to process commands. With a dedicated server, you would just send the request (`select * from emp`, for example) directly to the dedicated server for processing. With shared server connections, this is a tad more complex. You don't have a single dedicated server anymore. There needs to be another way of getting the request to a process that can handle it. The steps undertaken at this point are as follows:

1. The dispatcher places the request in a common (shared by all dispatchers) queue in the SGA.
2. The first shared server that is available will pick up and process this request.
3. The shared server will place the answer to the request in a response queue private to the queuing dispatcher in the SGA.
4. The dispatcher you are connected to will pick up this response and send it back to you.

Figure 3-3 illustrates this process.

It is interesting to note that these steps are performed for each call to the database. Therefore, the request to parse a query may be processed by shared server "1", the request to fetch the first row from the query may be processed by shared server "2", the next row by shared server "3", and finally shared server "4" might be the one that gets to close the result set.

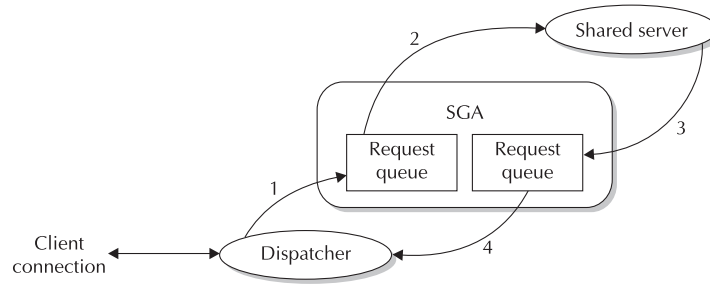


FIGURE 3-3. *Processing a request in shared server configuration*

As you can see, the code path from the client submitting the request (some SQL) to getting an answer is a long and winding one. However, this architecture works well if you need to scale up your system. With dedicated server connections, every session is a new process. With shared server connections, the number of processes is fixed (or, at least, set within a hard-and-fast range) and controllable. In order to get the $N+1$ connection, all you need is available memory in the SGA for that session's UGA (memory used by the session for state variables, cursors, and other objects).

Pros and Cons of Shared Server Connections

Here are some of the advantages of using a shared server connection to the database:

- It is more scalable as you need to scale up. On a machine that could physically handle 100 dedicated server connections, you might be able to handle 1,000 or more shared server connections, depending on the application. If you have a traditional client/server type of application where the vast majority of the connected sessions are inactive (for example, out of 1,000 connects, maybe only 50 to 100 are concurrently active, and the rest are experiencing “user think time”), then shared server configuration can be useful in scaling up your application. Note that this scaling refers to CPU scaling only—you still need sufficient memory for these 1,000 connections.
- It allows fine-grained control over how many sessions may be active concurrently; it imposes a hard limit on the number of overall active sessions.
- It uses marginally less memory than dedicated server configuration. The difference in memory usage is discussed in the section, “Shared Server Uses Significantly Less Memory” later in this chapter.
- It lets you get that $N+1$ user logged on when dedicated server configuration would fail.
- If you have a high rate of connects/disconnects, using shared server configuration may prove faster than dedicated server. The process and thread creation and destruction process is very expensive, and it may outweigh the overhead of the longer code path.

8 Effective Oracle by Design

Some of the disadvantages of shared server configuration are as follows:

- It is more complex to set up than dedicated server configuration, although the GUI tools provided with the database generally hide that complexity today. Most of the setup problems I see people encountering are when they try to edit the configuration files by hand, instead of using the tools.
- On a system where dedicated server connections will work fine, shared server connections will be slower. A shared server's code path is, by definition, longer than a dedicated server's code path.
- Some features of the database are not available when connected via shared server. You *must* connect to the database in dedicated server mode for these features. For example, using shared server connections you cannot shut down or start an instance, perform media recovery, or use Log Miner.
- The memory for the sum of all of the concurrently connected sessions UGA must be computed and configured as part of the LARGE_POOL init.ora file setting. You need to understand the maximum degree of connectivity and size appropriately. In Oracle9i Release 2, with the ability to dynamically resize the LARGE_POOL setting online, this is less of a concern as you can monitor your memory use and alter it (but this is a manual change, not automatic).
- Tracing (using SQL_TRACE) is currently something I would deem as not feasible when using a shared server connection. The trace files are specific to a process, and in shared server mode, a process is not specific to a session. Not only will your SQL statements be written to many different trace files, other session's SQL statements will be written to these same trace files. You cannot ascertain which statements are yours and which are theirs.
- There is a potential for artificial deadlocks. These occur because once a shared server starts processing a request, it will not return until it is finished with it. Hence, if the shared server is sent an UPDATE request and the row is already locked, that shared server will block. Suppose the blocking session was idle for a period of time (not doing any work in the database so it doesn't have a shared server allocated to it) and all of the other shared servers get blocked as well by this idle session (or any other session), then when the session holding the lock attempts to commit or rollback (which would release the locks), it will hang because there are no more free shared servers. This is an artificial deadlock situation, where the blocker cannot free the blockees, since the blockees have all of the shared servers tied up.
- There is a potential for shared server monopolization. This occurs for the same reason as the artificial deadlocks. If you use shared server with long-running transactions, your sessions will monopolize a shared resource, preventing others from using it. If you have too many of these long-running transactions, your system's performance will appear abysmal, because short transactions must wait for long transactions to complete.

As you can see, for most general-purpose uses, shared server configuration is not worth it. Only when your database cannot physically handle the load should you considering using a shared server configuration.

Common Misconceptions about Shared Server Connections

Through the asktom web site and Usenet newsgroup exchanges, I've become aware of some common misconceptions about shared server configuration. These misconceptions are being passed down from generation to generation of Oracle developers, and I would like to stop them in their tracks.

Shared Server Uses Significantly Less Memory

This is one of the most highly touted reasons for using shared server: It reduces the amount of memory. I've seen people enable shared server, change their `tnsnames.ora` file to request shared connections, and change nothing else (configure no extra memory in the SGA). These systems immediately fall over as the SGA runs out of memory. These DBAs forgot that the UGA will still be 100% allocated, but now it will be allocated in the SGA instead of in local process memory. So, while the amount of memory used by operating system processes will be significantly reduced, this is offset in most part by the increase in the size of your SGA.

When you enable shared server, you must be able to somewhat accurately determine your expected UGA memory needs and allocate that memory in the SGA via the `LARGE_POOL` setting. So, the SGA requirements for the shared server configuration typically are very large. This memory must be preallocated, and thus, it can be used only by the database. It is true that in Oracle9i Release 2, you can dynamically resize the `LARGE_POOL` setting online, but for all intents and purposes, this must be preset. Compare this with dedicated server configuration, where anyone can use any memory not allocated to the SGA.

So, if the SGA is much larger due to the UGA being located in it, where do the memory savings come from? They come from having that many fewer PGAs allocated. Each dedicated/shared server has a PGA for process information (sort areas, hash areas, and other process-related structures). It is this memory requirement that you are removing from the system by using shared server mode. If you go from using 5,000 dedicated servers to 100 shared servers, it is the cumulative sizes of the 4,900 PGAs you no longer need that you are saving with shared servers. However, even this is questionable, since all modern operating systems employ sophisticated paging mechanisms that would effectively page out the memory not currently in use by the 4,900 PGAs.

So, shared server configuration can save some memory, but not nearly as much as some people expect (before actually doing it and discovering the hard way!).

You Should Always Use Shared Server Configuration with Application Servers

Only if you have a large application server farm, where each application server has a large connection pool all connecting to a single database instance, should you consider using shared server configuration. Here, you have the same considerations as with any other system (the application server makes no difference). If you have a large number of connections and you are overloading the database server, it would be time to use shared server connections to reduce the number of physical processes.

"We are using connection pooling in the middle tier and shared server on the database tier. It is slow. Why?"

Well, it is slow because you are, in effect, caching a cache. This is like using an operating system cache *and* the database cache. If one cache is good, two must be twice as good, right? Wrong. If you are using a connection pool at the middle tier, the odds are you want to be using a dedicated server for speed. A connection pool will allocate a fixed (controllable) number of persistent physical connections. In that case, you do not have a large number of sessions and you do not have fast connect/disconnect rates—that is the recipe for dedicated server!

Shared Servers Are Only for Client/Server Applications

This is the converse of the application server misconception of the previous section. Here, the DBA would never consider shared server configuration for an application server, since the application server is doing pooling already. Well, the problem with that theory is that the application server is just the client. You still have clients (the application server connection pool) and a server (the database). From the database perspective, this whole paradigm shift from client/server to *N*-tier application models never happened! It is apparent to the developer that they are different, but to the database, this difference is not really visible.

You should look at the number of concurrent physical connections to the database. If that results in too many processes for your system to handle, shared server connections may be the solution for you.

An Instance Runs in Shared Server Mode (or Not)

This is probably the most common misconception: A database instance is in shared server mode or dedicated server mode. Well, the fact is, it may be both. You can always connect via a dedicated server configuration. You can connect via a shared server configuration if the necessary setup (dispatchers and shared servers) was configured. You can connect in one session using dedicated server mode, and then in another session, connect to the same database using shared server mode.

The database does not run in shared server or dedicated server modes. They are connection types, and a database can support both simultaneously.

Dedicated Server Versus Shared Server Wrap-up

Unless your system is overloaded, or you need to use shared server configuration for a specific feature, a dedicated server configuration will probably serve you best. A dedicated server is simple to set up and easier to tune with. There are certain operations that must be done in a dedicated server mode, so every database will have either both dedicated and shared servers or just a dedicated server set up.

On the other hand, if you know you will be deploying with shared server due to a very large user community, I would urge you to *develop and test* with shared server connections. Your likelihood of failure will increase if you develop under just a dedicated server configuration and never test on a shared server configuration. Stress the system, benchmark it, and make sure that your application is well-behaved under the shared server configuration; that is, make sure it does not monopolize shared servers for too long. If you find that it does so during development, it is much easier to fix the problem there than it is during deployment. You can use features such as advanced queues (AQ) to turn a long-running process into an apparently short one, for example, but you must design that into your application. These sorts of things are best done when you are developing your application.

If you are already using a connection-pooling feature in your application (for example, you are using the J2EE connection pool), and you have sized your connection pool appropriately,

using shared server connections will be a performance inhibitor in most cases. You already designed your connection pool to be the size you need concurrently at any point in time, so you want each of those connections to be a direct, dedicated server connection. Otherwise, you just have a connection-pooling feature connecting to yet another connection-pooling feature.

For details on dedicated server and shared server architecture, as well as the Oracle memory structures (the PGA, SGA, and UGA), see the Oracle *Concepts Guide*.

Take Advantage of Clustering

For high availability and the ability to scale horizontally, there is nothing like clustering. In order for that sentence to make sense, we'll need to define a few of the terms first.

Clustering is a hardware configuration whereby multiple computers are physically connected together to permit them to cooperate to perform the same task against a single set of shared disks. Clustering is a well-established technology, pioneered by Digital Equipment Corporation (DEC) with its VAX/VMS machines in the 1980s. Today, clusters can range from high-end machines or just a bunch of laptops running Linux hooked together with FireWire. The machines in the cluster have their own private network for communication, also known as a *high-speed interconnect*. Its implementation varies by hardware vendor, from the proprietary high-end solutions to just the network on Linux. This private network allows the machines in the cluster to communicate very rapidly.

In the past, clustering was a technology that was beyond the reach of most people. You needed specialized hardware configurations that hardware vendors typically supported only on the high end of their configurations. Hence, the cost barriers for the hardware to support clustering were high. Now, the ability to cluster commodity, off-the-shelf machines (such as those provided by Dell or Hewlett-Packard/Compaq), with operating systems such as Linux, opens the world of clustering to the masses. For hardware costs of less than \$10,000 to \$20,000, you could have a fully functional four- to eight-CPU cluster up and running overnight. All it takes is two servers and a cable.

Horizontal scaling is the ability to add capacity to your system by adding more machines. In the past, we have scaled vertically by putting additional CPUs into a single box, adding RAM, and so on. Vertical scaling works well until you hit the limits of your machine. For example, a 32-bit operating system cannot make significant use of memory beyond 4GB. A conventional Unix machine cannot scale beyond about 64 to 128 CPUs. Horizontal scalability breaks those barriers. Each machine might have hard limits, but together in a cluster, they can surpass those limits. Horizontal scaling provides the ability to add computer resources over time as the need arises.

High availability is one of the salient features of clusters. They are, by definition, highly available platforms. As long as one computer remains running in the cluster, the resources of the cluster are available. No longer does the failure of a single computer prevent access to your database.

In this section, we will look at a configuration of Oracle called Real Application Clusters (RAC) and how it works on clusters. We will also explore how RAC provides horizontal scalability and high availability.

“We are building the biggest database of all time, with the most users ever. It will start with 100 users for the first year, growing to 500 in two years, and getting way up there in users over the long haul. But this year, the needs are rather modest. What’s the best way to do this?”

12 Effective Oracle by Design

There are two approaches you could take in this case:

- **Buy the biggest machine you can** This will give you the capacity available two years from now. The downside is that you will spend a lot of money today for something that will be one-tenth the cost in two years.
- **Buy a machine that fits your needs today** Then buy another one in a year for half the price at twice the speed, and buy another when you need it, and so on. The upside is that you buy only what you need today and get what you need later for less. The only caveat is that you need to make sure the hardware you buy today is, in fact, clusterable. That is a question for your hardware vendor, and the answer varies from vendor to vendor.

How Does RAC Work?

A database may be mounted and opened by many instances simultaneously, and that is what RAC is all about. In order to understand how RAC works, we first need to clear up one of the more confusing topics in Oracle: the difference between an instance and a database.

An Instance versus Database Refresher

An *instance* is simply the set of processes and memory used by these processes (the SGA). An Oracle instance does not need to have a database associated with it at all. You can start an instance without any datafiles whatsoever. An instance is not very useful in that mode, but you can have one.

A *database* is the set of files—redo, control, data, and temporary files—that holds your data. A database has no processes or memory; it just has files.

Typically, an Oracle instance is started with the `startup` command, but I'll take you through the long way, so you can see when we have an instance and when we have an instance that has a database associated with it:

```
idle> startup nomount
ORACLE instance started.

Total System Global Area 303108296 bytes
Fixed Size      450760 bytes
Variable Size   134217728 bytes
Database Buffers 167772160 bytes
Redo Buffers    667648 bytes
idle>
```

So, here we have an Oracle instance. If you were to run `ps` at the operating-system level (or use the equivalent tool on Windows), you would see the Oracle processes are running—the SGA is allocated. Everything is started, except for a database! In fact, there are even some queries we can run:

```
idle> select * from dual;

ADDR      INDX INST_ID D
-----
0A62D1C0  0   1 X
```

```
idle> select * from v$datafile;
select * from v$datafile
      *
ERROR at line 1:
ORA-01507: database not mounted
```

So, DUAL is there (but it looks a tad strange, since it has extra columns), but not many other tables are available. For example, we do not have V\$DATAFILE yet, because no datafiles are associated with this instance.



NOTE

This special DUAL table is there for applications that need DUAL but are expected to work against databases that are not yet opened, such as RMAN. As soon as you open the database, the regular DUAL reappears.

So, the Oracle instance is up and running, but has yet to associate itself to a set of files and make them accessible. We'll do that now:

```
idle> alter database mount;
Database altered.
idle> select count(*) from v$datafile;

COUNT(*)
-----
        23
```

```
idle> alter database open;
Database altered.
```

```
idle> @login
sys@ORA920>
```

And there we go. Now, the database is open and ready for access. The ALTER DATABASE MOUNT command used the control files (specified in the init.ora file) to locate the redo, data, and temporary files. The ALTER DATABASE OPEN command made the database available to anyone via this instance.

Note that an instance here is a transient thing. An instance is simply a set of processes and memory. Once you shut down that instance, it is *gone* forever. The database persists, but instances are fleeting. Another interesting thing to note is that an instance can mount and open at the most *one* database during its life. We can see this simply by issuing these commands:

```
sys@ORA920> alter database close;
Database altered.

sys@ORA920> alter database open;
alter database open
```

14 Effective Oracle by Design

```
*
ERROR at line 1:
ORA-01531: a database already open by the instance

sys@ORA920> !oerr ora 1531
01531, 00000, " a database already open by the instance"
// *Cause: During ALTER DATABASE, an attempt was made to open
// a database on an instance for which there is already
// an open database.
// *Action: If you wish to open a new database on the instance,
// first shutdown the instance and then startup the
// instance and retry the operation.
```

The reason this is interesting is that the converse is not true. As noted at the beginning of this section, a database may be mounted and opened by *many* instances simultaneously. The rules for databases and instances in Oracle are as follows:

- An instance is transient. It exists only as long as the related set of processes and memory exists.
- A database is persistent. It exists as long as the files exist.
- An instance may mount and open a *single* database in its life.
- A database may be mounted and opened by many instances, either one after the other or by many simultaneously (that is RAC).

Instances in RAC

In an RAC environment, you run an instance of Oracle on each machine (also known as a *node*) in the cluster. Each of these instances mount and open the same database, since clusters share disks. It is important to keep in mind that there is only one database. Each Oracle instance has full read/write access to every byte of data in the database.

Each instance is a peer of every other instance in the cluster. You can update the EMP table from node 1. You can update the EMP table from node 2. In fact, you can update the record where `ename = 'KING'` on both node 1 and node 2, following the same rules you would if you used only a single instance of Oracle. The same locking and concurrency control mechanisms you are familiar with on a single instance of Oracle with a single database hold true for Oracle running RAC.

Figure 3-4 shows what an RAC might look like with a four-node (four-computer) cluster. Each node would be running its own instance of Oracle (the processes/memory). There is a single database image shared by all nodes in the cluster, each having full read/write access to every byte of data in that database.

The basic premise behind Oracle RAC is that you run many instances of Oracle. Each instance is a stand-alone entity—a peer of the other instances. There is no master instance, so there is no single point of failure. Each instance will mount and open the same database—the same files. The instances will work together keeping their caches consistent with each other, to avoid overwriting changes made by other instances and to avoid reading old/stale information. If one of them fails for any reason—due to a hardware failure or a software failure—one of the

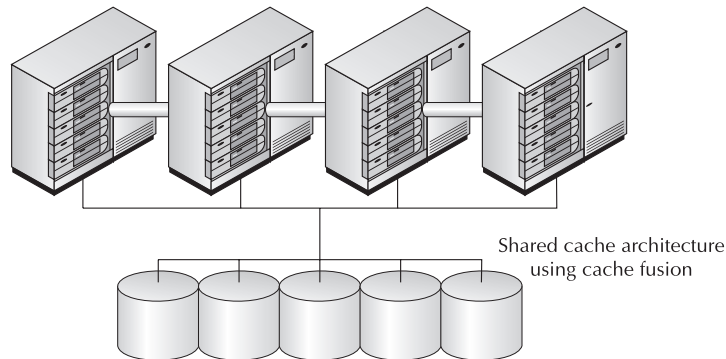


FIGURE 3-4. RAC with a four-node cluster

other remaining instances in the cluster will recover any work that the failed instance had in progress, and operations continue as before.

A Service Refresher

There is one more term we need to define to understand how RAC works: *service*. You may have noticed that, starting in the Oracle8.0 days, it was recommended that a `tnsnames.ora` entry look like this:

```
ORA920.US.ORACLE.COM =
  (DESCRIPTION =
    (ADDRESS_LIST =
      (ADDRESS = (PROTOCOL = TCP)(HOST = tkyte-pc-isdn)
        (PORT = 1521))
    )
    (CONNECT_DATA =
      (SERVICE_NAME = ora920)
    )
  )
```

Instead of this:

```
ORA920.US.ORACLE.COM =
  (DESCRIPTION =
    (ADDRESS_LIST =
      (ADDRESS = (PROTOCOL = TCP)(HOST = tkyte-pc-isdn)
        (PORT = 1521))
    )
    (CONNECT_DATA =
      (ORACLE_SID = ora920)
    )
  )
```

That is, we stopped using an `ORACLE_SID` to identify a database and started using a more generalized `SERVICE_NAME`. The reason behind that recommendation was to pave the way for features like RAC. The client application will now optimally connect to a service. The service is a collection of instances, the most common of which is a collection of one. Whereas a `SID` identifies an instance, a service identifies a collection of one or more instances.

In this fashion, the client connects to the database, not to a particular instance, since each instance is a peer of every other instance. Using this extra level of abstraction allows for features like load-balancing (connecting the client to the least utilized instance). It's also creates a less error-prone connection. If you connect to a specific instance and that instance is not up, you will be made aware of it. If you connect to a service and at least one instance is up, you will not be made aware of the fact that the other N instances are down.

At a rather high level, that is how RAC works. Physically, there is one database with many instances. To the client applications that connect, there is just one service they connect to, and that service will get them connected to an instance, which provides full read/write access to the database. All of this happens without the application knowing a thing about what is going on behind the scenes. You can take your existing applications and point them at a RAC database, and they will run just as before. Any application that runs against Oracle9i will run unmodified on Oracle9i RAC in a cluster.

It is true that there are performance implications to consider under RAC as well. There are extreme cases where applications may run slower under RAC. For example, if you have a system that performs excessive parsing (due to lack of proper bind variable use), you'll rapidly discover that since library cache operations are globally coordinated in the cluster, the problem you had in a single instance is simply magnified in a cluster. I have seen people take the approach of "we will use clusters to hide our performance issues" only to discover that clustering in many cases will amplify an already poorly performing, poorly designed system. With RAC, you cannot fix issues such as:

- **Library Cache Contention** If you had a problem with the library cache in a single instance, multiple instances will only amplify the problem.
- **Excessive IO** Since IO is coordinated globally, an IO issue under a single instance configuration will likewise be amplified in a multi-instance configuration.
- **Locking (blocking)** Especially concentrated locking issues of the type you might find in a system that uses a table with a counter to provide sequence numbers rather than using a sequence itself. This implementation causes contention in a single instance—this contention will be amplified in a clustered environment.

These design issues must be corrected using implementation techniques. For example, you will need to reduce library cache contention by proper use of bind variables and by parsing SQL statements only as often as you must (Chapter 5). You can reduce excessive IO by designing and implementing physical structures that are suitable for the way you use the data (Chapter 7). You can remove or reduce locking issues by using built-in database features such as sequences. Oracle9i RAC can scale, but it cannot fix an already broken system!

What Are the Benefits of RAC?

So now that you know how RAC works, you can see how RAC is to databases what RAID is to disks. It provides a level of redundancy. If a disk fails in a RAID array, no problem; in most

systems today, you can just hot-swap in a new disk. If an Oracle instance fails in RAC, no problem; the other instances will automatically take over. You can fix the failed instance, correcting the hardware or software problem that caused the failure.

RAC is a solution to make sure that once the database is up and running, nothing short of taking the entire data center out in a fire or other catastrophe will bring down your entire database. So, you can protect your disks with RAID or some other technology, networks with redundancy, and databases with redundant Oracle instances.

This is not something the company I work for just talks about, this is how we run our major systems ourselves. Internally, there is one email database for the company (yes, we store all of our email in Oracle). Now, the company is fairly large, with more than 40,000 users, all with email accounts. A single instance on the biggest hardware isn't big enough for us. As of the writing of this book, we have three instances of Oracle in a three-node RAC cluster, accessing this single database image. If you've ever interacted with Oracle (the company), you know it runs on email, and its email runs on RAC. We have failover scenarios using the Oracle9i Data Guard product as well, but to keep the email database up and running 24/7, we rely on RAC. The failover is only in the event the data center itself meets with some catastrophe.

Along with the benefit of high availability, the other main benefit of RAC is that it allows you to scale horizontally. As explained earlier, this means that rather than upgrading the existing machine, adding additional CPUs or whatnot, you can place another, small to medium-sized machine next to the existing hardware. Using RAC, you can buy what you need today and add capacity over time as you need it. That three-node RAC cluster we use at Oracle was a two-node RAC cluster at one point in time. We scaled it horizontally by adding another machine, not throwing out the existing hardware and getting bigger boxes.

Clustering Wrap-up

I really do envision RAC, and clustering technology in general, as being the current wave in computing. I remember back in the early 1990s, the contractor I worked for was very wary of a new technology that was coming out. It was called symmetric multiprocessing (SMP). In fact, that company was so wary of this technology that we got stuck with a bunch of single-CPU boxes. Well, today SMP is so prevalent that we just accept it as natural. Having a machine with two, four, eight, or more CPUs is normal.

I believe the same type of acceptance is happening now with clustering technology. The adoption rate is picking up. No longer do you need to buy large, expensive equipment to support a cluster. Today, you can get the benefits of it on small, commodity hardware. It is a very mature technology—the software has been in place for many years, and experience managing these systems in the real world exists. Clustering is no longer bleeding edge; it is becoming the norm.

If high availability and the ability to scale when you need to are what you want from your next system, or even your current system, consider clustering. The full concepts behind RAC are well-documented by Oracle. One guide in particular stands out—the *Real Applications Clusters Concepts Guide* (yes, another *Concepts Guide*). Here, you will find many of the details on the physical implementation of RAC, how the caches are kept consistent with each other (using a feature called *cache fusion*), and more. The *RAC Concepts Guide* has an entire section on nothing but scaling with RAC—a how-to guide if you will. For example, it describes how a parallel query might not only be parallelized on a single machine across all of the available CPUs, but also across all of the instances in a RAC cluster. The query may execute in parallel on each of the *N* nodes in a cluster.

Use Locally Managed Tablespaces

In the previous section on Oracle9i RAC, we looked at a process architecture implementation for scaling and availability. In this section, we'll get back to basics and take a look at how Oracle manages data on disk. Since Oracle8i release 1, Oracle has supported two types of tablespaces: dictionary-managed tablespaces (DMTs) and locally managed tablespaces (LMTs). My recommendation regarding the use of DMTs versus LMTs is straightforward: Pretend DMTs do not exist; use only LMTs.

For databases that have existing DMTs, work on moving the objects out of them into newly created LMTs as the opportunity arises. You don't need to run out and move everything tomorrow, but over time, if you do reorganize an object, put it in an LMT. Don't simply convert existing DMTs into LMTs. Instead, move the objects and then drop the old tablespace. Using `DBMS_SPACE_ADMIN.TABLESPACE_MIGRATE_TO_LOCAL` will convert a DMT into a user-managed LMT, but will not achieve one of the major benefits of LMTs—uniformly sized extents. The odd-sized extents the DMT had before the migration will still exist.

There are actually two kinds of LMTs: system-managed LMTs and LMTs that allow you to set uniform extent sizes. Here, I'll explain why you should use LMTs and when to use each type of LMT.

Why Are DMTs Obsolete?

With DMTs, space in datafiles is managed in data dictionary tables: `SYS.UE$` for used extents and `SYS.FE$` for free extents. Every time you need an extent added to a table or you free extents for reuse via a `DROP` or `TRUNCATE` command, Oracle will perform recursive SQL to insert or delete from these tables. This is very slow in many cases.

With DMTs, there are also issues with coalescing (joining together) contiguous free extents. In a DMT, free extents are stored as a row in the `FE$` table. If you drop a table with two 5MB extents, Oracle will record them using a row apiece. Now, suppose those two extents were actually contiguous (right next to each other). Until some process (SMON) actually combines them into a single record, Oracle does not recognize them as representing 10MB of free space; rather, it sees them as two separate extents of 5MB apiece. An allocation request for 10MB of storage would ignore these extents until they were coalesced. Additionally, with DMTs, you need to take care to avoid having too many extents, because that situation has a big negative impact on the performance of many administrative functions, such as rebuilding, moving, truncating, and dropping objects.

Oracle handles LMTs quite differently. With this type of tablespace, datafile space is managed via a bitmap in the header of the datafiles themselves. Rather than having a row per used extent in `SYS.UE$` and a row per free available extent in `SYS.FE$`, the datafile headers just have a series of zeros and ones. When you need a new extent, Oracle will go to the datafile header and scan the bitmap looking for free space. When you free an extent, Oracle just zeros out the corresponding bit(s). There are no issues with regard to coalescing free space. A set of contiguous zeros in the bitmap means all of this space is free and contiguous. You do not need to worry about avoiding thousands of extents in an object, because the space-management features are exceedingly fast compared with those for DMTs.

One of my favorite new features in Oracle9i Release 2 is the fact that if you create the default database using the Database Configuration Assistant (DBCA), `SYSTEM` will be installed as an LMT. Once that happens, *it is impossible to create a DMT*. This catches some people by surprise, but it should be a pleasant surprise. DMTs are obsolete; you should not be using them anymore. If you use the `CREATE DATABASE` command manually, you should specify "EXTENT MANAGEMENT LOCAL" in order to have your `SYSTEM` tablespace locally managed.

There has been a lot of FUD (fear, uncertainty, doubt) about LMTs. There are DBAs out there who refuse to use them still. Maybe it is because it makes this aspect of the database so much easier. I mean, what happens if you don't have tablespace fragmentation and you don't have considerations about INITIAL, NEXT, PCTINCREASE, and MAXEXTENTS? Aren't you just making the job of the DBA go away? Well, if that is all the DBA is doing, maybe so. But it is my experience that this is an aspect of database administration that most DBAs wish would disappear; they are tired of micromanaging the space allocation for each and every object.

The facts that LMTs preclude fragmentation, are faster, are easier to use, and have not a single identifiable downside when compared to DMTs should be more than enough reasons to start using them. The fact that in Oracle9i Release 2, by default, LMTs are the *only* kind of tablespace says a lot about where Oracle Corporation is going with this feature.

Use System-Managed LMTs When You Do Not Know How Big Your Objects Will Become

With a *system-managed LMT*, the system figures out exactly what the extent sizes will be for a table. Oracle will use an internal, undocumented algorithm to allocate space for every object in the tablespace. With this method, the first couple of extents will be small (64KB), and then the extents will get larger and larger over time. They will all be multiples of each other. That fact precludes free space fragmentation in these tablespaces, because every chunk of free space is potentially usable by any other object. This is in contrast to a DMT, where an object will request arbitrary extent sizes, and if there is not a contiguous free extent large enough to satisfy the request, the request will fail. It is true that in a system managed LMT, since there is more than one extent size, a request for space may fail even if there is existing free space. However, it is many times less likely due mostly to the fact that there is a very limited number of extent sizes. It would be rare to have free space that is not usable in a System managed LMT and in general, the space will be very small.

In the beginning, I was a little cautious about using this type of tablespace. It just didn't feel right. I was always taught that we should size our objects, watch their space usage like a hawk—micromanage the tablespace. Maybe I'm getting lazy, but I would prefer not to work that way if I can avoid it. With system-managed LMTs. I'm not kept awake at night wondering if a PCTINCREASE is going to go crazy on me or if my tablespace will be fragmented like Swiss cheese, with a lot of different-sized holes in it. Objects grow in a sensible fashion, without a lot of watching or handholding.

To see how space might be allocated in such a tablespace, let's use BIG_TABLE again (as noted in the appendix, this is a table created from ALL_OBJECTS and duplicated over and over to have quite a few rows). For this demonstration, I made a copy of this table in an auto-allocate LMT.

NOTE

Another feature I'm rather fond of is Oracle-managed files, which I'm using here. I do not need to specify the datafile information in the CREATE TABLESPACE command. I previously set up the DB_CREATE_FILE_DEST init.ora parameter, and Oracle will name, create, and manage the operating system files for me.

```
big_table@ORA920> create tablespace SYSTEM_MANAGED
2 extent management local;
```

20 Effective Oracle by Design

Tablespace created.

```
big_table@ORA920> create table big_table_copy
2 tablespace SYSTEM_MANAGED
3 as
4 select * from big_table;
```

Table created.

```
big_table@ORA920> select tablespace_name, extent_id, bytes/1024, blocks
2 from user_extents
3 where segment_name = ' BIG_TABLE_COPY'
4 /
```

| TABLESPACE_NAME | EXTENT_ID | BYTES/1024 | BLOCKS |
|-----------------|-----------|------------|--------|
| SYSTEM_MANAGED | 0 | 64 | 8 |
| SYSTEM_MANAGED | 1 | 64 | 8 |
| ... | | | |
| SYSTEM_MANAGED | 14 | 64 | 8 |
| SYSTEM_MANAGED | 15 | 64 | 8 |
| SYSTEM_MANAGED | 16 | 1024 | 128 |
| SYSTEM_MANAGED | 17 | 1024 | 128 |
| ... | | | |
| SYSTEM_MANAGED | 77 | 1024 | 128 |
| SYSTEM_MANAGED | 78 | 1024 | 128 |
| SYSTEM_MANAGED | 79 | 8192 | 1024 |
| SYSTEM_MANAGED | 80 | 8192 | 1024 |
| ... | | | |
| SYSTEM_MANAGED | 91 | 8192 | 1024 |
| SYSTEM_MANAGED | 92 | 8192 | 1024 |

93 rows selected.

As you can see in this example, the first 16 extents each are 64KB (but don't be surprised if you see something slightly different). The next 63 are each 1MB, and the remaining 14 are 8MB. As the object grew, the extents grew as well. That is a total of about 180MB of space in 93 extents, which is perfectly acceptable. For those who believe you must have your objects in one extent, I hope that this Oracle-supported algorithm, which promotes and encourages multiple extents, puts that notion to rest.

In this example, when we quadrupled the size of the BIG_TABLE_COPY to about 650MB, Oracle added another 64 extents, each 8MB, for a total of 512MB more. That table was just taking care of its space needs.

When you do not know how big your objects will become, system-managed LMTs, are the way to go. This type of space management is most useful when you are creating objects for an application, and depending on how the people using the application configure it, some tables might be empty, some might be 100MB, and some might be 2GB. On another installation, the tables that were 2GB are empty, the tables that were empty are 100MB, and the tables that were 100MB tables are 2GB. In other words, you have no idea how big these tables are going to be in

the real world. Here, having each object start at 64KB and stay that way for the first couple of extents lets the tables that are nearly empty stay very small. The tables that are going to get large will get large fast.

If you are using Oracle9i Release 2 with an LMT, you'll find that it uses system-managed extent sizing for the SYSTEM tablespace. For the SYSTEM tablespace, this extent-sizing strategy is the best thing ever. It will prevent dictionary tables from growing exponentially (as PCTINCREASE would have them doing after a while; even a small PCTINCREASE would cause a table to grow by huge amounts after a short period of time) and keep the number of extents at a reasonable maximum. Consider that if you install Oracle and use a ton of PL/SQL, your SYS.SOURCE\$ table and the tables that hold the compiled PL/SQL will be huge. On the other hand, your friend may install Oracle and not write any lines of PL/SQL code. With the system-managed approach, Oracle will let the database figure out how big the extents should be.

System-managed LMTs work well, as long as the objects are destined to be 10GB or less. At 10GB, you would be using about 300 extents for the object in this type of tablespace, which is fine. Segments that exceed 10GB are fairly rare. If you are using partitioning, the individual partitions are the segments; 10GB partitions or smaller would be a good size. For tables and indexes that are getting into the 10GB size range, consider partitioning them into smaller, more manageable segments. For segments larger than 10GB, or for those that you prefer to size yourself for some reason, consider using LMTs with uniform extent sizes.

Use Uniform Extent Sizes When You Know the Ultimate Size of an Object

The other type of LMT is one that supports uniformly sized extents. Using this strategy, each and every extent in the tablespace will be exactly the same size as every other extent in that tablespace. To see how this works, let's use the same example we did in the previous section with BIG_TABLE, but with a 5MB uniform extent size.

```
big_table@ORA920> create tablespace uniform_size
  2 extent management local
  3 uniform size 5m
  4 /
Tablespace created.
```

```
big_table@ORA920> create table big_table_copy
  2 tablespace uniform_size
  3 as
  4 select * from big_table
  5 /
Table created.
```

```
big_table@ORA920> select tablespace_name, extent_id, bytes/1024, blocks
  2 from user_extents
  3 where segment_name = ' BIG_TABLE_COPY'
  4 /
```

```
TABLESPACE_NAME    EXTENT_ID BYTES/1024  BLOCKS
-----
```

22 Effective Oracle by Design

```
UNIFORM_SIZE      0  5120  640
UNIFORM_SIZE      1  5120  640
...
UNIFORM_SIZE      34  5120  640
```

35 rows selected.

As expected, every extent that was and will ever be allocated in that tablespace will be 5MB—not a byte more, not a byte less.

So, when is this type of tablespace useful? It works well when you know the ultimate size of the object. For example, when you are loading a data warehouse, you have a good idea of what size the objects will be. You know that table will have fifty 5GB partitions.

The trick here is to pick an extent size that will hold your object with the least amount of waste. If this segment is 5GB + 1 byte, it will need to extend one more time for that last byte, effectively wasting .499999GB of storage. In a large data warehouse, that is a drop in the bucket, but every bit (or byte) counts. You might be willing to set a target of 500 extents, which allows you to use a 10MB extent and waste at most 10MB of space.

Another factor to consider when deciding on an extent size is the desired monitoring granularity. An extent size that implies a new extent each month makes abnormal growth easy to spot. If you were to size the extents so that they grow by dozens every day, this spot check would be more difficult.

Some LMT Caveats

Here, I will point out some caveats I've discovered along the way using LMTs. None of these issues are earth-shattering; none would make me stop using LMTs. I just wanted to make sure you are aware of these points.

The Magic Number for Uniformly Sized Extents Is 64KB

There was a common myth when LMTs first came out that an LMT with a uniform extent size would waste its first extent for the bitmap used to manage the file; that is, if you used a 1MB uniform extent size, the first 1MB of the tablespace would be taken over by Oracle. People thought, "Oh my gosh, if I use a 100MB uniform size, I'll waste 100MB of data!" Well, there is overhead, but it is only 64KB, not the size of the extent.

The problem was that most people were creating their datafiles as an integral multiple of the uniform extent size. For example, if they used a 5MB extent size, they would use 100MB as a datafile size ($20 \times 5 = 100$). Oracle would look at this 100MB that was just given to it, take away 64KB for itself, divide what was left by 5MB, and truncate the result (no partial extents here). Thus, there were only 19 extents available in this tablespace right after creation! Here is a simple example showing the issue and then how to fix it. We'll start by creating a tablespace with uniform extent sizes of 5MB and a datafile of 100MB.

```
ops$tkyte@ORA920> create tablespace five_meg
2 datafile size 100m
3 uniform size 5m
4 /
```

Tablespace created.

Now, we will query how much free space is available in this newly created tablespace.

```
ops$tkyte@ORA920> select sum(bytes/1024/1024) free_space
2 from dba_free_space
3 where tablespace_name = ' FIVE_MEG'
4 /

FREE_SPACE
-----
          95
```

So, it would appear that 5MB of overhead was taken, but this is not really the case. Let's increase that file by a measly 64KB.

```
ops$tkyte@ORA920> column file_name new_val f
ops$tkyte@ORA920> select file_name from dba_data_files
2 where tablespace_name = 'FIVE_MEG';
```

```
FILE_NAME
-----
/usr/oracle/ora920/OraHome1/oradata/ora920/ol_mf_five_meg_zc54bj51_.dbf
```

```
ops$tkyte@ORA920> alter database
2 datafile '&f' resize 102464k;
old 2: datafile '&f' resize 102464k
new 2: datafile
'/usr/oracle/ora920/OraHome1/oradata/ora920/ol_mf_five_meg_zc54bj51_.dbf'
resize 102464k
```

Database altered.

```
ops$tkyte@ORA920> select sum(bytes/1024/1024) free_space
2 from dba_free_space
3 where tablespace_name = 'FIVE_MEG';

FREE_SPACE
-----
        100
```

And there you go. We have all of our space. I'm not sure how many DBAs this feature scared off, but I hope they will come around after seeing this!

Note that this 64KB rule need not apply to system-managed extents! If you ran the above test without the `uniform size 5m` clause, you would find that there is initially 99.9375MB of available free space, and all but 64KB of it would be usable.

System-Managed LMT Allocates from Files Differently

“How can you achieve ‘poor-man’s striping’ of a table? That is, without using striping software at the operating-system level and without using partitioning, how would you stripe a table across multiple devices?”

To accomplish this, create a tablespace with many datafiles. Use an LMT, of course. Use a uniform extent size that will cause the table to go into 100 or so extents. Create the table in that tablespace. Allocate extents for it in a round-robin fashion, spreading the data out.

I had tested this in the past using a DMT, and Oracle always tended to go round-robin when allocating space. I had also tested this with an LMT using uniform extent sizes, and it worked the same. However, when the user tried to apply this technique, he used system-managed extents! The results were quite different than what I had experienced. But, after further testing, it seems that the round-robin technique will still kick in when the object gets large enough to mandate striping.

To see how the different types of LMT allocate from files, we will consider a small example. We will start by creating two tablespaces, each with four datafiles that are 64KB larger than 2MB (the extra 64KB you need for the bitmap). One will use 64KB uniform extents; the other will be system-managed.

```
ops$tkyte@ORA920> create tablespace uniform_extents
  2  datafile size 2112k, size 2112k, size 2112k, size 2112k
  3  uniform size 64k
  4  /
```

Tablespace created.

```
ops$tkyte@ORA920>
ops$tkyte@ORA920> create tablespace system_managed
  2  datafile size 2112k, size 2112k, size 2112k, size 2112k
  3  /
```

Tablespace created.

Next, we’ll create a table in each tablespace.

```
ops$tkyte@ORA920> create table uniform_size ( x int, y char(2000) )
  2  tablespace uniform_extents;
```

Table created.

```
ops$tkyte@ORA920>
ops$tkyte@ORA920> create table system_size ( x int, y char(2000) )
  2  tablespace system_managed;
```

Table created.

Now, we'll fill each table to capacity using a loop. Note that the following is an example to illustrate a particular point: how the LMTs will allocate space from multiple files. The goal is to cause the table to extend and grow to fill all available space. In order to accomplish that, we use simple row-at-a-time inserts, followed by a commit. Real production code would use bulk inserts and commit only after all of the inserts were performed, of course. Unfortunately, in order to demonstrate a point, we need to use two extremely bad practices.

```
ops$tkyte@ORA920> begin
2   loop
3       insert into uniform_size values( 1, 'x' );
4       commit;
5   end loop;
6 end;
7 /
begin
*
ERROR at line 1:
ORA-01653: unable to extend table OPS$TKYTE.UNIFORM_SIZE
        by 8 in tablespace UNIFORM_EXTENTS
ORA-06512: at line 3
```

```
ops$tkyte@ORA920>
ops$tkyte@ORA920> begin
2   loop
3       insert into system_size values( 1, 'x' );
4       commit;
5   end loop;
6 end;
7 /
begin
*
ERROR at line 1:
ORA-01653: unable to extend table OPS$TKYTE.SYSTEM_SIZE
        by 128 in tablespace SYSTEM_MANAGED
ORA-06512: at line
```

Now, when we inspect how the extents were allocated, we'll see something very different in the algorithm:

```
ops$tkyte@ORA920> select segment_name, extent_id, blocks, file_id
2   from dba_extents
3  where segment_name in ( 'UNIFORM_SIZE' , 'SYSTEM_SIZE' )
4  and owner = user
5  order by segment_name, extent_id
6  /
```

| SEGMENT_NAME | EXTENT_ID | BLOCKS | FILE_ID |
|--------------|-----------|--------|---------|
| SYSTEM_SIZE | 0 | 8 | 22 |

26 Effective Oracle by Design

| | | | |
|---|----|-----|----|
| SYSTEM_SIZE | 1 | 8 | 22 |
| SYSTEM_SIZE | 2 | 8 | 22 |
| ...note extent_id 3..14 were all in FILE_ID 22... | | | |
| SYSTEM_SIZE | 15 | 8 | 22 |
| SYSTEM_SIZE | 16 | 128 | 23 |
| SYSTEM_SIZE | 17 | 128 | 24 |
| SYSTEM_SIZE | 18 | 128 | 21 |
| SYSTEM_SIZE | 19 | 128 | 22 |
| SYSTEM_SIZE | 20 | 128 | 23 |
| SYSTEM_SIZE | 21 | 128 | 24 |
| SYSTEM_SIZE | 22 | 128 | 21 |

The output shows the first 16 extents for the system-allocated tablespace all came from the same file (FILE_ID 22 in this case). Oracle did this in order to keep all of the small extents together, to avoid allocating a lot of small extents in many different files in an attempt to reduce any potential fragmentation that might occur if we dropped an object from this tablespace. Only when the extents jumped from 8 blocks (64KB) to 128 blocks (1MB) did the round-robin algorithm come into play. Once we hit 1MB extents, the allocation went to file 23, 24, 21, 22, 23, and so on.

However, with the uniformly sized extents, fragmentation is not a concern, so this method uses round-robin allocation from the start.

| | | | |
|--------------------|-----|---|----|
| UNIFORM_SIZE | 0 | 8 | 18 |
| UNIFORM_SIZE | 1 | 8 | 19 |
| UNIFORM_SIZE | 2 | 8 | 20 |
| UNIFORM_SIZE | 3 | 8 | 17 |
| UNIFORM_SIZE | 4 | 8 | 18 |
| UNIFORM_SIZE | 5 | 8 | 19 |
| UNIFORM_SIZE | 6 | 8 | 20 |
| UNIFORM_SIZE | 7 | 8 | 17 |
| UNIFORM_SIZE | 8 | 8 | 18 |
| ... | | | |
| UNIFORM_SIZE | 122 | 8 | 20 |
| UNIFORM_SIZE | 123 | 8 | 17 |
| UNIFORM_SIZE | 124 | 8 | 18 |
| UNIFORM_SIZE | 125 | 8 | 19 |
| UNIFORM_SIZE | 126 | 8 | 20 |
| UNIFORM_SIZE | 127 | 8 | 17 |
| 151 rows selected. | | | |

Most of the time, this difference in allocation doesn't really matter. The round-robin technique will kick in when the object gets large enough to mandate striping. I point it out mostly to show that the round-robin allocation will happen, but a simple test might not make that clear to you. If you allocate fewer than 16 extents in a test, you might be under the impression that it will never round-robin. (In fact, I was under that impression myself; it was only when making a larger test that I discovered the truth.)

How to Autoextend Datafiles

This is not a caveat so much as a recommendation. Some people like autoextensible datafiles (I do). Some people hate them. If you are among those who use autoextensible datafiles, you should follow these guidelines when using them with LMTs:

- With uniformly sized extents, set the next size equal to the extent size. Make sure to add that extra 64KB kicker we discussed. Set MAXSIZE to $N * \text{extent_size} + 64\text{KB}$, if you set MAXSIZE at all (recommended—I always set MAXSIZE).
- With system-managed extents, things get trickier. You don't know if the file wants to grow by 64KB, 1MB, or 8MB. I suggest you grow it 8MB at a time. Start the file at whatever size (the 64KB kicker doesn't come into play), but let it grow in 8MB intervals. Use the assumption that the larger segments are the ones that cause the file to autoextend.

Beware of Legacy Storage Clauses!

This is a common point of confusion: Some people think that STORAGE clauses are meaningless with LMTs. This is true after the initial CREATE TABLE statement. However, during the CREATE statement, the STORAGE clauses are fully in effect, and you must be cautious of their side effects. Oracle will calculate how much storage would have been allocated using a DMT, and then allocate *at least* that much space in the LMT.

For example, suppose that you had this STORAGE clause:

```
storage ( initial 1m next 5m minextents 3 )
```

Oracle would allocate at least $1\text{MB} + 5\text{MB} + 5\text{MB} = 11\text{MB}$ of space in the LMT—1MB for the initial extent, and then two more 5MB extents to satisfy the MINEXTENTS clause. Now, suppose you did that CREATE statement in a tablespace with uniform extent sizes of 10MB. You would have 20MB allocated initially. After the CREATE statement, however, the INITIAL, NEXT, PCTINCREASE, MINEXTENTS, and MAXEXTENTS settings are ignored. The LMT takes over from there.

So, beware of statements bearing STORAGE clauses. They could allocate a lot more than you think! Be especially careful with the EXP (database export) tool. By default, that tool uses the option COMPRESS=Y, which causes it to sum up the *currently* allocated extent sizes and generates a STORAGE clause with an initial extent allocation of that size. That is not desirable.

LMT and DMT Wrap-up

In summary, my advice for tablespaces and LMT versus DMT is pretty clear. Never use a DMT again. Use only LMTs. Here are the reasons:

- You cannot have a fragmented LMT. Fragmentation of the sort you would get easily in a DMT cannot happen in a LMT.
- The number of extents in an object is not relevant. You need not be concerned with objects that have many extents. In the past, you *may* have had a reason to be concerned, not because of query performance but because of DDL performance, such as DROP or TRUNCATE.
- LMTs perform in a superior fashion to DMTs. The recursive SQL overhead is virtually entirely gone.
- LMTs require less thought. You do not try to figure out what the optimal INITIAL, NEXT, PCTINCREASE, and MAXEXTENTS are. They are not relevant (well, if you use them, they are relevant and usually disastrous).

Furthermore, for most general-purpose uses, LMTs with system-managed extent sizes are fine. It is the “supersize” case that needs fine-tuning these days—the data warehouse-sized objects.

Setting storage parameters is a thing of the past. Remove the STORAGE clause from your CREATE statements—at least the INITIAL, NEXT, MAXEXTENTS, MINEXTENTS, and PCTINCREASE clauses. It really is getting easier!

For details on DMTs and LMTs, see the trusty Oracle *Concepts Guide*, Part II Database Structures.

Know When to Use Partitioning

Partitioning is the ability of the database to take very large tables or indexes and physically break them into smaller, more manageable pieces. Just as parallel processing can be useful to break a large process into smaller chunks that can be processed independently, partitioning can be used to physically break a very large table/index into several smaller chunks that can be *managed* independently.

In order so to use partitioning as a tool to solve your problems, you need to have a solid understanding of how partitioning is physically implemented. Just as important, you must be able to state what it is you want to achieve via partitioning. How to best implement partitioning will vary based on your goals.

Partitioning Concepts

Before we discuss when partitioning is useful, we need to review some of the basic partitioning concepts. These include the various types of partitioning schemes that Oracles uses for tables and indexes, and partition pruning.

Partitioning Schemes

In a nutshell, there are four partitioning schemes Oracle employs for tables:

- **Range-based** Data is separated into partitions using ranges of values. This is typically used on dates. For example, all data for “Q1 of the year” goes into this partition, and all data for “Q2 of the year” goes into another partition, and so on.
- **Hash-based** A hash function is applied to a column(s) in the table. The more distinct this column is, the better; a primary key, for example, makes an excellent hash key. The hash function will return a number between 1 and N that dictates in which of the N partitions the data will be found. Due to the particular hashing algorithm Oracle uses internally, the hash function works best when N is a power of 2.
- **List-based** You set up discrete lists of values to tell the database in which partition the data belongs. For example, you could use a CODE field and dictate that records with the codes A, X, and Y go into partition P1, while records with the codes B, C, and Z go into partition P2. List partitions are limited to single columns only as opposed to range or hash partitions that may work on multiple columns. What this means is that list partitioning can only look at a single column in a table in order to decide the partition to place the data in. This scheme is new with Oracle9i Release 1.
- **Composite** This is a hybrid partitioning scheme. The data will be partitioned initially by range. Then each range partition may be further subpartitioned by either a hash or a list. So, you could set up four range partitions, one for each quarter of a year, and then

have each of the quarterly partitions themselves partitioned by hash into eight partitions, resulting in a total of $4 \times 8 = 32$ partitions. Or, you might set up four range partitions, again by quarter, but then list partition each quarter by four regions, resulting in $4 \times 4 = 16$ partitions. Interestingly, with composite partitioning you don't need to have exactly the same number of subpartitions per primary partition. That is, one partition may consist of 4 subpartitions while another partition consists of 5 subpartitions.

There are two primary ways to partition indexes in Oracle, as illustrated in Figure 3-5:

- **Locally partitioned index** For each and every table partition created, there will be an index partition. The data in each index partition points to data in exactly one table partition by definition. It is said that the table and index are *equipartitioned*. There is a one-to-one mapping between the table partition and index partition, so if the table has N partitions, the index has N partitions as well.
- **Globally partitioned index** The index is partitioned by its own scheme. There is no parity between the index partitions and the table partitions. A single index partition may point to data in *any number of* table partitions, unlike the locally partitioned index. A globally partitioned index may only be range-partitioned. Indexes on partitioned tables are, by default, global indexes in a single partition (effectively, by default, they are not partitioned indexes at all).

That explains at a high level the physical storage mechanisms behind partitioning: We can partition a table by range, list, hash, or some combination of range and hash or list. Indexes can be locally or globally partitioned. The next concept we need to explore is partition elimination.

Partition Pruning

Partition elimination, or partition pruning, is the ability of the optimizer to preclude from consideration certain partitions in the evaluation of a query. For example, suppose we have a table partitioned by the `SALE_DATE` column. Every month of data is in its own partition, and

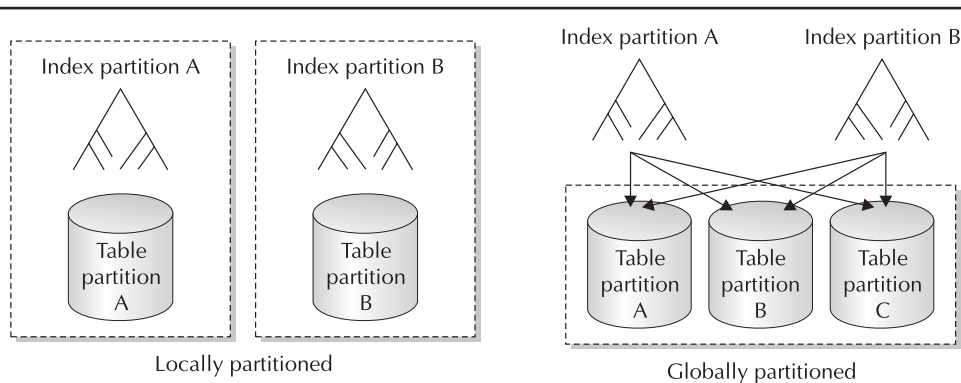


FIGURE 3-5. The differences between locally partitioned and globally partitioned indexes

there is a locally partitioned index on this SALE_DATE column. Now, suppose we have this query:

```
select * from t where sale_date = :x
```

This query may use one of two query plans, depending on the values in the SALE_DATE column. Suppose the optimizer chose a full-table scan. Since the table is partitioned by SALE_DATE, the optimizer already knows that $N-1$ of the partitions that make up this table cannot have any data we need. By definition, only one of the partitions could possibly have our data. Hence, instead of fully scanning the entire table, Oracle will fully scan a single partition, significantly reducing the amount of I/O it would otherwise have performed.

Alternatively, assume the query plan included an index-range scan. Again, the optimizer would be able to remove from consideration $N-1$ of the index partitions. It knows that all of the rows of interest reside in a single index partition, so it would only range-scan a single index partition, getting the ROWIDs of the rows that matched, and then accessing that single partition to retrieve the data.

This capability of the optimizer to remove partitions from consideration is extremely powerful. From a performance perspective, it's the main reason for using partitioning. It is something we'll keep in mind as we explore partitioning further.

The Partitioning Myth

Partitioning is not the proverbial `fast=true` setting (as I've stated before, there is *no* `fast=true`!). Partitioning is a very powerful tool that, when applied at the right time, solves a problem. Conversely, when inappropriately applied, partitioning only wastes resources, consumes additional time, and disappoints.

"I'm really disappointed in partitioning. I took our largest OLTP table and put it into ten partitions. I thought everything should just go faster. That is what I was told. But what we've discovered is that everything, in fact, is running much slower than before. Our machine just cannot keep up. How could that be? Partitioning is supposed to speed everything up, isn't it? Partitioning adds too great of an overhead to be practical."

In this case, they did not look at *how* partitioning worked under the covers; they did not understand what would happen to their queries at runtime. They heard partitioning should make things faster. They didn't really think about how it would affect their application by just turning it on, however. So, they took their biggest table, and not really having anything "logical" to partition it on, simply partitioned by hash on the primary key, which was a surrogate key populated by a sequence. They then proceeded to re-create all indexes as locally partitioned indexes. The problem was they did not always query the data by primary key; that is, the queries were *not* all of the form:

```
select * from t where primary_key=:x
```

Rather they would query:

```
select * from t where some_column = :x
```

A query against `SOME_COLUMN` would have to now inspect 10 index partitions, so it would need to scan 10 separate index structures!

The solution is twofold. The technical solution is to globally partition the index on `OWNER`. The other part is to establish a reasonable goal for partitioning from the start. In a transactional system, partitioning generally *cannot* speed up queries. What it can do is reduce contention for the shared resource.

In order to demonstrate problems that can arise with partitioning, let's use the `BIG_TABLE` example once again. We'll take that 1.8 million-record table and hash-partition it:

```
create table big_table_hashed nologging
partition by hash(object_id) partitions 10
as
select * from big_table;
```

Next, we'll create a new record in `BIG_TABLE` and `BIG_TABLE_HASHED`. This will be the record of interest. We will use some column other than the column that the table was hashed on to pull up a record, the `OWNER` column:

```
insert into big_table ( owner, object_name, object_id,
                      last_ddl_time, created )
select 'SPECIAL_ROW' , object_name, object_id, last_ddl_time, created
from big_table where rownum = 1;

insert into big_table_hashed ( owner, object_name, object_id,
                              last_ddl_time, created )
select 'SPECIAL_ROW' , object_name, object_id, last_ddl_time, created
from big_table_hashed where rownum = 1;
```

We have no users named `SPECIAL_ROW`, so this insert operation created a record with a unique owner value in each table. Next, let's implement the index and analyze the tables:

```
create index big_idx1 on big_table(owner);
create index big_hash_idx1 on big_table_hashed(owner) LOCAL;

analyze table big_table for table for all indexes for all indexed columns;
analyze table big_table_hash
for table for all indexes for all indexed columns;
```

Consider the results of our first attempts:

```
big_table@ORA920> delete from plan_table;
4 rows deleted.

big_table@ORA920> explain plan for select * from big_table where owner = :x;
Explained.

big_table@ORA920> select * from table(dbms_xplan.display);
```

32 Effective Oracle by Design

PLAN_TABLE_OUTPUT

| Id | Operation | Name | Rows | Bytes | Cost |
|----|-----------------------------|---------------|-------|-------|---------|
| 0 | SELECT STATEMENT | | 65493 | 5372K | 169 (1) |
| 1 | TABLE ACCESS BY INDEX ROWID | BIG_TABLE | 65493 | 5372K | 169 (1) |
| *2 | INDEX RANGE SCAN | BIG_TABLE_IDX | 65493 | | 153 (0) |

Predicate Information (identified by operation id):

2 - access("BIG_TABLE"."OWNER"=:Z)

13 rows selected.

Now, using the default init.ora parameters more or less, we find:

```
big_table@ORA920> delete from plan_table;
3 rows deleted.

big_table@ORA920> explain plan for
  2 select * from big_table_hashed where owner = :x;
Explained.

big_table@ORA920> select * from table(dbms_xplan.display);
```

PLAN_TABLE_OUTPUT

| Id | Operation | Name | Rows | Bytes | Cost | PSTART | PSTOP |
|----|--------------------|------------------|-------|-------|--------|--------|-------|
| 0 | SELECT STATEMENT | | 18319 | 2289K | 28 (0) | | |
| 1 | PARTITION HASH ALL | | | | | 1 | 10 |
| 2 | TABLE ACCESS BY | | | | | | |
| | LOCAL INDEX ROWID | BIG_TABLE_HASHED | 18319 | 2289K | 28 (0) | 1 | 10 |
| *3 | INDEX RANGE SCAN | BIG_HASH_IDX | 7328 | | 36 (0) | 1 | 10 |

Predicate Information (identified by operation id):

3 - access("BIG_TABLE_HASHED"."OWNER"=:Z)

14 rows selected.

At first, this looks reasonable—an index-range scan, just as before. But notice the partition start (PSTART) and stop (PSTOP) keys. It is scanning every index partition. It must do this, because every index partition could, in fact, contain any value of OWNER in it! The index is partitioned by OBJECT_ID, not by OWNER; hence, every index partition must be inspected. The results of this are dramatic. Just consider this query executed 1,000 times (we might be doing this many times per second).



```
select * from big_table where owner = :b1
```

| call | count | cpu | elapsed | disk | query | current | rows |
|---------|-------|------|---------|------|-------|---------|------|
| Parse | 1 | 0.00 | 0.00 | 0 | 0 | 0 | 0 |
| Execute | 1000 | 0.14 | 0.11 | 0 | 0 | 0 | 0 |
| Fetch | 1000 | 0.11 | 0.11 | 0 | 4000 | 0 | 1000 |
| total | 2001 | 0.25 | 0.23 | 0 | 4000 | 0 | 1000 |

Rows Row Source Operation

```
1000 TABLE ACCESS BY INDEX ROWID BIG_TABLE
1000 INDEX RANGE SCAN BIG_TABLE_IDX
```

```
select * from big_table_hashed where owner = :b1
```

| call | count | cpu | elapsed | disk | query | current | rows |
|---------|-------|------|---------|------|-------|---------|------|
| Parse | 1 | 0.00 | 0.00 | 0 | 0 | 0 | 0 |
| Execute | 1000 | 0.14 | 0.11 | 0 | 0 | 0 | 0 |
| Fetch | 1000 | 0.33 | 0.60 | 16 | 27000 | 0 | 1000 |
| total | 2001 | 0.47 | 0.72 | 16 | 27000 | 0 | 1000 |

Rows Row Source Operation

```
1000 PARTITION HASH ALL PARTITION: 1 10
1000 TABLE ACCESS BY LOCAL INDEX ROWID BIG_TABLE_HASHED PARTITION: 1 10
1000 INDEX RANGE SCAN BIG_HASH_IDX PARTITION: 1 10
```

That is a 675% increase in logical I/O. (Remember that a logical I/O implies a latch, a latch is a lock, and locks make an application less scalable.) It's also a 100% increase in CPU consumption. This simple operation has impacted performance in an extremely negative fashion. (Of course, this is something we should have caught during testing!)

NOTE

Some might question, "Why 27,000 logical IO's versus 4,000?" Shouldn't it be 40,000 (ten times as much) or perhaps 30,000? Actually, on your system you might find that you see slightly different logical IO results. It all has to do with the height of the B*Tree indexes. In this test, some of the index partitions had a height of two while others had a height of three, meaning sometimes it took two logical IO's to find "no data" and sometimes it took three. The single index in the non-partitioned example had a height of three, meaning every query read three index blocks and one table block (4 logical IO's per row). In the partitioned example, four of the index partitions had a height of two and six had a height of three, giving us $1000*((4*2) + (6*3) + 1) = 27,000$ logical IO's

So, what technically could we do to fix the problem (short of unpartitioning the table, that is)? This is where the global index comes in. We could either partition the index by range into many partitions or simply use a single index partition. For sake of demonstration, let's partition by range, splitting the index alphabetically into four more or less even ranges.

```
create index big_hash_idx1 on big_table_hashed(owner)
global partition by range (owner)
( partition values less than ( ' F' ),
  partition values less than ( ' M' ),
  partition values less than ( ' T' ),
  partition values less than ( MAXVALUE )
);
```

Now, the query plan for that same query is as follows:

```
-----
|Id| Operation                                | Name                | ...|Pstart|Pstop|
-----
| 0| SELECT STATEMENT                        |                     | ...|      |     |
| 1| PARTITION RANGE SINGLE                  |                     | ...| KEY  | KEY  |
| 2| TABLE ACCESS BY GLOBAL INDEX ROWID    | BIG_TABLE_HASHED    | ...| ROWID| ROW L|
|*3| INDEX RANGE SCAN                       | BIG_HASH_IDX        | ...| KEY  | KEY  |
-----
```

Predicate Information (identified by operation id):

```
-----
3 - access("BIG_TABLE_HASHED"."OWNER"=:Z)
```

That is looking more like it. We can now see the index-range scan start and stop partitions are based on KEY. The optimizer will not hit all ten index partitions; it will hit only one. It will get the ROWID(s) for the matching rows, and then access a partition to get the row data for that ROWID. Now, let's try that same 1,000-query simulation.

```
select * from big_table_hashed where owner = :b1
```

| call | count | cpu | elapsed | disk | query | current | rows |
|---------|-------|------|---------|------|-------|---------|------|
| Parse | 1 | 0.00 | 0.00 | 0 | 0 | 0 | 0 |
| Execute | 1000 | 0.14 | 0.11 | 0 | 0 | 0 | 0 |
| Fetch | 1000 | 0.11 | 0.11 | 0 | 4000 | 0 | 1000 |
| total | 2001 | 0.25 | 0.23 | 0 | 4000 | 0 | 1000 |

```
1000 PARTITION RANGE SINGLE PARTITION: KEY KEY
1000 TABLE ACCESS BY GLOBAL INDEX ROWID BIG_TABLE_HASHED PARTITION:
      ROW LOCATION ROW LOCATION
1000 INDEX RANGE SCAN BIG_HASH_IDX PARTITION: KEY KEY
```

What a difference that makes! CPU time is now the same as in the nonpartitioned example. The logical I/Os are the same. But shouldn't we still be disappointed, because it is not any faster?

No, because we would be looking in the wrong place for a performance increase. How much faster can a query that does four logical I/Os get?

The performance increase you may realize in this sample OLTP system comes from the fact that you now have four separate index structures and ten separate table structures. So, the next time you insert a record, you will experience perhaps 25% of the contention you would normally have on the index modification and 10% of the contention you would have on the table insert operation. This is also true with update and delete operations. You have more physical structures; you've spread contention for a shared resource out across many physical segments. Your queries might not run significantly faster, but your modifications may be accomplished much more quickly.

Consider the local unique index we'll maintain on the primary key in this example. In the past, there was one index structure and everyone was hitting the same "side" of it; that is, the users would insert 1, 2, 3, 4 ... N into this index in order. That is a lot of contention for the index blocks on the right side of the index, where the new values will go. Now, the index entry for primary key = 1 probably goes into a *different* index structure than primary key = 2, and so on. We are hashing these values and putting them into an index segment based on the hash values. We have just accomplished a very nice distribution of data across ten local index partitions, perhaps dramatically reducing contention for this shared resource.

Can partitioning speed up a query? Why, certainly it can, but typically *not in a transactional system*. A measurable reduction in query response time due to partitioning will be achieved when the original query takes a large amount of time to begin with. In a transactional system, you do not have those types of queries. However, in a data warehouse, you do have long-running queries.

Why Use Partitioning?

There are three main reasons to use partitioning:

1. Increased availability
2. Easier administration
3. Increased performance

You might wonder why I listed them with numbers. I put them in order of achievability and immediate payback. Let's start with the easiest goal.

Increased Availability

You get increased availability pretty much out of the box. The amount you achieve is a function of how much thought you put into your partitioning scheme, but you still instantly get more availability from partitioning. So, how does partitioning achieve the goal of higher availability? There are two main methods: partition elimination and divide-and-conquer.

For example, suppose we run a data warehouse, and a datafile goes bad (becomes media corrupt for whatever reason). We need to go to our backup and restore it. Meanwhile, our end users are still using this database. Since we used partitioning and have placed our partitions in separate tablespaces, only one partition in one tablespace is actually affected. We take that tablespace offline and recover it. In the end user world, anyone who attempts to run a query that accesses that tablespace would be affected. However, a large population of our end users is *not*

affected by this recovery, and they use the same table(s) and the same queries. It is just that their queries know that they do not need to access this data; it is removed from consideration, and the queries keep on going. As far as these end users are concerned, we never had a failure. Our system is more available from that perspective.

With partitioning, the system is also more available because you are managing things in smaller chunks. You might have a 2GB partition to restore, one out of fifty other partitions perhaps. You do not have a 100GB problem to deal with, just a little 2GB problem (or however big you make your partitions). Every administrative task that takes an object offline (makes it unavailable) is now affecting a much smaller object, so the operation runs faster. Suppose you need to rebuild a bitmap index after a couple of months of incremental loading (bitmap indexes do not respond well to incremental updates; they need to be rebuilt on a recurring basis). Would you like to rebuild the entire bitmap index in one big statement, requiring significant resources (you'll have two bitmaps for a while), or would you rather rebuild one-fiftieth of a bitmap index at a time?

Easier Administration

Performing operations on small objects is inherently easier, faster, and less resource-intensive than performing the same operation on a large object. For example, if you discover that 50 percent of the rows in your table are migrated rows, and you would like to fix this, having a partitioned table will facilitate the operation. In order to fix migrated rows, you must typically rebuild the object, in this case, a table. If you have one 100GB table, you will need to perform this operation in one very large chunk, serially using ALTER TABLE MOVE. On the other hand, if you have fifty 2GB partitions, you can rebuild each partition one by one. Alternatively, if you are doing this during off-hours, you can even do the ALTER TABLE MOVE PARTITION statements in parallel, in separate sessions, potentially reducing the amount of time the operation takes. If you are using locally partitioned indexes on this partitioned table, the index-rebuild operations will take significantly less time as well. Virtually everything you can do to a nonpartitioned object, you can do to an individual partition of a partitioned object.



NOTE

There is a rather neat package, DBMS_PCLXUTIL, documented in the Supplied PL/SQL Packages and Types Reference that automates a parallel build of local indexes as well.

Another factor that comes into play with partitioning and administration is a concept called a *sliding window*. This is applied in data warehousing and in some transactional systems, especially to audit trails whereby the system must keep N years of audit information online. Here, we have time-based information—a load into a warehouse of last month's sales, or an audit trail where each record is timestamped. Every month we would like to take the oldest month's data, purge it, and slide in the newest month's data. Using a conventional table, this would be a huge DELETE of the old data, followed by a huge INSERT of the new data. The continuous purge via DELETE and load via INSERT processing would consume a great deal of resources (both generate significant redo and undo information). They would tend to break down the indexes and table itself, with the delete operations leaving holes that may or may not be reused over time, introducing the possibility of fragmentation and wasting space.

With partitioning, the purge of the old data becomes trivial. It becomes a simple ALTER TABLE command that drops the oldest data partition. To slide the new data in, you would

typically load and index a separate table, and then add that table to the existing partitioned table via an `ALTER TABLE ... EXCHANGE PARTITION` command, which swaps a table with a partition. In this case, you would swap a full table with an empty partition, just adding the data to the table. The downtime for the end users can be very near zero here, especially if care is taken to avoid constraint validation, which is entirely possible.

Increased Performance

As you saw in the example in the previous section, partitioning for increased performance is a two-edged sword. Getting additional, measurable performance out of partitioning is the hardest (not hard, just the hardest) goal to achieve from partitioning because it requires the most thought.

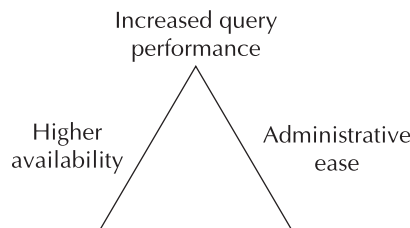
Partitioning must be applied in such a way as to solve a real problem you are having. Just applying partitioning without some planning will more often than not slow you down. Think of partitioning like medicine. Medicine can make sick people healthier, stronger, and faster. Medicine can make healthy people die when administered improperly. Partitioning will do the same thing to your database. Before applying it, you need to elucidate what your goals in using partitioning are. Do you want to use partitioning to ease administrative overhead? Do you want to use partitioning to increase availability? Is your goal to increase performance (after identifying a performance issue, of course)?

Partitioning Wrap-up

Partitioning can be a great thing. Breaking a huge problem down into many smaller problems (the divide-and-conquer approach) can, in some cases, dramatically reduce the processing time. Partitioning can also be a terrible thing. When inappropriately applied, it can dramatically increase processing time.

Partitioning, in order to be properly used, must be properly understood. It is vital that you have a goal you want to achieve with partitioning. Don't use it until you do!

In many cases, the three things you can achieve with partitioning could be thought as points on a triangle, like this:



You can get to any point on or inside the triangle, but that might mean that the closer you get to any one point, the further you travel from the others. You need to first and foremost figure out why you want to use partitioning, and then with that goal in mind, design your physical schemas. For that, you will need to use your knowledge of the “physics” behind partitioning, what the database is able to do under the covers for you. Remember, this is an analogy. How big that triangle is, and thus how far away from any given point, is entirely a matter of implementation and need. In some cases, the triangle is very small meaning you can achieve all three. In other cases, you may find that due to your needs, you have to trade off one capability at the favor of another.

There are several Oracle manuals that provide details on partitioning: the *Oracle Concepts Guide*, *Data Warehousing Guide*, and *Administrators Guide*.

Know When to Use Parallel Operations

Parallelism is something we apply to large jobs all of the time in real life. You never see a single person build a building. You see a large team of people build a building, all working simultaneously.

Oracle supports a wide variety of parallel operations:

- **Parallel query** The ability to take a single SQL statement, break it into a series of operations that can be performed at the same time by different processes/threads, and executing them concurrently.
- **Parallel DML (modifications)** Similar to parallel query, but for INSERT, UPDATE, and DELETE statements.
- **Parallel DDL** For administrators, the ability to execute a wide variety of statements such as CREATE INDEX in a parallel fashion, not by running 15 CREATE INDEX statements at the same time (Oracle can, of course, do that), but rather by using *N* processes to create a single index in parallel.
- **Parallel DIY** Do It Yourself parallelism, which isn't really traditional Oracle parallelism. Oracle isn't divvying up the work, you are. This is one of the more overlooked opportunities for tuning! Oracle is great at supporting multiuser situations, taking advantage of that in your batch processing is another thing entirely.

Here, we will explore when and why you might use each of these options (and when and why you might not). But first, we will look at a couple other topics: the myth that all parallel operations are quicker than serial operations, and parallel administration, where I personally find parallelism to be most useful in day-to-day work.

The Parallel Myth

Parallelism can be a great thing. Breaking a huge problem down into many smaller problems (the divide-and-conquer approach) can, in some cases, dramatically reduce the processing time.

Parallelism can be a terrible thing. Applying parallel processing to many classes of problems makes them run slower (or not any faster) but with much greater resource consumption. Parallelism, when inappropriately applied, can dramatically increase processing time. Like partitioning—and everything else—parallelism is just a tool. It works well when used correctly and can have disastrous results when used incorrectly.

“I’m trying to test the parallel query option. We have a two-CPU machine and enough disk controllers to support the disk. We’ve created a partitioned parallel table. It’s partitioned by MONTH and looks like ALL_OBJECTS. We put 1,471,488 records in there via: `insert /*+ append */ into tb_part_par select * from dba_objects`. Then we created a nonpartitioned table TBL_NO_PAR as `select * from tb_part_par`. Next, we use AUTOTRACE and timing to see the difference:

```
SQL> set timing on
SQL> set autotrace on
SQL> select count(1) from tb_part_par;
```

```

COUNT(1)
-----
1471488
```

Elapsed: 00:00:07.43

Execution Plan

```

-----
0 SELECT STATEMENT Optimizer=CHOOSE (Cost=1064 Card=1)
1 0 SORT (AGGREGATE)
2 1 SORT* (AGGREGATE)           :Q115000
3 2 PARTITION RANGE* (ALL)      :Q115000
4 3 TABLE ACCESS* (FULL) OF ' TB_PART_PAR'
   (Cost=1064 Card=1471488)    :Q115000

2 PARALLEL_TO_SERIAL SELECT /*+ PIV_SSF */ SYS_OP_MSR(COUNT(*))
   FROM (SELECT /*+ NO_EXPAND ROWID(A2)

3 PARALLEL_COMBINED_WITH_PARENT
4 PARALLEL_COMBINED_WITH_PARENT
```

That is the parallel plan and results. In serial mode, we see:

```
SQL> select count(1) from tbl_no_par;
```

```

COUNT(1)
-----
1471488
```

Elapsed: 00:00:06.18

Execution Plan

```

-----
0 SELECT STATEMENT Optimizer=CHOOSE
1 0 SORT (AGGREGATE)
2 1 TABLE ACCESS (FULL) OF ' TBL_NO_PAR'
```

So, how come the time is almost the same? The table that has parallel should be faster, right (we have four disk controllers and two CPUs)?”

There are two things wrong here: assuming “parallel means faster” and using a flawed test. In short, there is overhead with a parallel query operation that makes itself very evident in things that run fast. *Fast* is a relative term. Setting up for a parallel query and the additional

coordination that a parallel query must do can add seconds to a query's runtime. For a query that ran in hours and now runs in minutes, those seconds aren't meaningful. But, for a query that runs in less than ten seconds, this overhead is very noticeable. There is a high startup cost for a parallel query, meaning the query you want to make go faster needs to be really slow to begin with (and six to seven seconds just isn't slow enough).

The other problem with their test was that all of the I/O was going against a single tablespace that probably had a single file (they didn't really spread anything out; all of the partitions were in the same tablespace). Making the problem even worse is that they will find most of their objects in DBA_OBJECTS created on exactly the same day, and that is what they partitioned by; hence, a single partition here contained most of the data. So, even if the partitions were on different devices, in different tablespaces, this setup would not have affected the I/O distribution!

```
ops$tkyte@ORA920> select trunc(created, ' mm' ), count(*) from dba_objects
2 group by trunc(created, ' mm' );
```

```
TRUNC(CRE  COUNT(*))
-----
01-MAY-02 29680
01-AUG-02  9
01-SEP-02  7
01-OCT-02 29
01-NOV-02 91
```

The day you installed or the day the sample datafiles used to seed your database was stamped out will be the most popular date by far in DBA_OBJECTS.

Forget about the disk layout for a moment, assuming that they used striped disks so there was an even I/O distribution. Why would this query run the same or marginally slower using parallel query? The reason is the overhead of parallel query. The sample set of data is only about 100MB to 125MB, and it is fully scanned serially in six seconds. The query just isn't big enough.

Parallel Processing Overhead

To get an understanding of the overhead entailed for parallel processing, think of it like this: You are a manager, and you have a dozen people that work for you. Your boss asks you to write a one-page summary of a meeting. You can either:

- Write it yourself
- Have a dozen people do it
 - Call a meeting
 - Get the 12 people together
 - Assign someone to write the first paragraph (this takes time, because you need to explain what you want and divvy up the work)

- Repeat the paragraph assignments for the remaining 11 people
- Collect all of their subresults
- Put them together into a single paper
- Deliver the results

For this one-page summary, it would be infinitely more efficient to do the job in serial mode, by yourself. You just sit down and write the report from start to finish. You can do it much faster than getting everyone together (starting the parallel query slaves), coordinating their work (playing the query coordinator), collecting and merging their pieces of the answer, and returning the result.

Now, later your boss asks you to write a book that is 12 chapters long. Each chapter is relatively independent of the other chapters. They could be written in isolation, but there will be some coordination necessary to build the table of contents, the index, and such at the end. Now you can either:

- Write the book yourself (at about one hour per finished page)
- Do it in parallel
 - Divvy up the 12 chapters among your 12 employees
 - As they finish each chapter, review and edit it
 - Assemble the book

Now, doing it in parallel would not take one-twelfth the time of doing it yourself, because you would have the overhead required to coordinate 12 people. However, it would be much faster than writing it yourself; maybe you could get it done in one-tenth the time.

Parallel query operations are the same way. They are for taking massively hard, long things and making them faster. If you have a short problem (seconds), parallel query operations will not make it faster; in fact, they may make it much slower. If you have a long problem (minutes, hours, days) and *all other* tuning processes have been exhausted, using parallel query may be the saving grace.

Parallel Processing Scalability

The other thing people get wrong with parallel query is its scalability. Parallel query is not a scalable solution! Parallel query was initially designed so that a single user in a data warehouse could completely and totally consume 100% of the resources on that machine. So, if you had a 16-CPU big box with gobs of RAM and tons of disk space, you could write a single query that would be able to make use of the entire machine. Well, if you have two people doing that, they will both be competing for the same resources—they will both try to consume 100% of the machine. Take that up to 100 users, and you can easily see the problem.

This is one of the reasons I like the parallel automatic tuning with the adaptive multiuser routines. If I am the only one on the machine, I can fully consume all of the resources. If, on the other hand, I run the same query when there are 50 other users logged in and running queries, I will most likely get far fewer resources (parallel threads) than I did when I was the sole user of the machine. The degree of parallelism is not fixed. It adapts to the current workload, making parallel query a more scalable solution than before. Instead of killing the machine or having queries fail because they could not get sufficient resources, I can let the machine adjust its resource usage

over time. This provides the best of both worlds *and* what end users expect from a system. They know that when it is busy, it will run slower.

Parallel Administration

This is by far my favorite use of the parallel operations. It is probably the most underutilized and overlooked feature as well. Here, the DBA is able to use parallel operations to speed up statements such as ALTER INDEX REBUILD (after a mass load), ALTER TABLE MOVE, or CREATE INDEX. It also works for database recovery, loading, collecting statistics, and so on. This is a use of parallel operation that benefits all system types—OLTP, data warehousing, or mixed workload. The faster the DBAs can complete their administrative functions, the more uptime the end users get. If you have an eight-CPU machine but only use one of them to create an index, you are missing the boat big time.

“Why shouldn’t I just use ANALYZE TABLE? It’s so easy. So what if my analyze window is three hours long now and tends to interfere with my hot backups (I/O contention)?”

The fact is a simple ANALYZE cannot be parallelized. DBMS_STATS, a better, more powerful way to collect statistics, can be parallelized. You can analyze a table in parallel using DBMS_STATS, you cannot with ANALYZE. Well, you can apply DIY parallelism in order to parallelize an ANALYZE. The individual ANALYZE commands will execute serially, but there is nothing stopping you from running many of them simultaneously. That is another perfectly valid approach.

The next time you execute any of the following operations, stop for a moment and ask yourself, “Would I benefit from parallelizing this operation—letting Oracle use the entire machine?”

- CREATE TABLE AS SELECT (CTAS)
- CREATE INDEX
- ALTER INDEX REBUILD
- ALTER TABLE MOVE
- ALTER TABLE SPLIT PARTITION
- ALTER INDEX COALESCE
- Statistics collection

In most cases, the answer would be a resounding, yes. The operation will run faster since these operations are generally done on big objects and would benefit from parallel operations.

For example, consider the case of what is known as a parallel direct-path load. This is perhaps the fastest way to load large amounts of data into the database. Suppose we want to load a table T using a parallel direct-path load. In order to accomplish this in the past, we would have needed to do this:

- Create a control file.
- Create a table, setting to NOLOGGING if necessary.

- Decide what the optimum degree of parallelism should be (say N).
- Split the input file into N files, one to be fed into each of the SQLLDR processes.
- Write a script to startup N copies of SQLLDR, each with different log filenames, bad filenames, discard filenames, and so on.
- Run and monitor the script.

Furthermore, if it were determined that a different degree of parallelism were to be used, all of these steps would need to be redone.

In Oracle9i, we only need to issue the following commands to achieve this same parallel direct-path load:

- `create external_table` (the ability to use SELECT on flat files, to use a file as if it were a database table)
- `create table T nologging parallel as select * from external_table`

Here, the degree of parallelism could be chosen by Oracle itself based on available resources (you could choose to override this, if you like). The file, as long as it is a fixed-length file format (fairly common in large bulk loads), is split dynamically by Oracle based on the chosen degree of parallelism. A CTAS statement is a direct-path operation. There is no script to write—no muss, no fuss.

I recently used this method to load a data warehouse for a benchmark I was working on. What used to take a day to set up and test—with all of the scripting, file splitting, and so on—took literally minutes. Set up the external tables, and away we go. We just fired off a couple of these at a time, let the machine go to 100%, and then submit some more. It was the easiest load I had ever done (oh, and it was fast, too).

Parallel Query

Parallel query is suitable for a certain class of large problems: very large problems that have no other solution. Parallel query is my last path of action for solving a performance problem; it's never my first course of action.

Parallel Query Settings

I will not discuss the physical setup of parallel query operations. That topic is well covered in both the Oracle *Concepts Guide* and *Data Warehousing Guide*. As I mentioned earlier, my current favorite way to set up parallelism in Oracle is using the automatic tuning option first introduced in Oracle8i Release 2 (version 8.1.6): `PARALLEL_AUTOMATIC_TUNING = TRUE`. With this setting, the other parallel settings are automatically set. Now, all I need to do is set the `PARALLEL` option on the table (not `PARALLEL <N>`, just `PARALLEL`) and Oracle will, when appropriate, parallelize certain operations on that table for me. The degree of parallelism (how many processes/threads will be thrown at a problem) will be decided for me and vary over time as the load on the system varies. I have found that, for most cases, this achieves my desired goal, which is usually to get the best performance, with the least amount of work, in a manner that is most manageable. Setting a single parameter is a great way to get there.

For the novice user wanting to play with parallel query for the first time, parallel automatic tuning is a good way to get started. As you develop an understanding of what parallel query does and how it does it, try tweaking some of the other parallel settings:

- **PARALLEL_ADAPTIVE_MULTI_USER** Controls whether the degree of parallelism should vary over time as the load on the system does; should the algorithm for assigning resources “adapt” to the increase in load.
- **PARALLEL_EXECUTION_MESSAGE_SIZE** Sets the size of the message buffers used to pass information back and forth between the processes executing the parallel query.
- **PARALLEL_INSTANCE_GROUP** Applies only to Oracle RAC configurations (Oracle Parallel Server, OPS, in Oracle8i and earlier). Allows you to restrict the number of instances that will be used to perform a parallel operation (as opposed to the number of processes an instance will use).
- **PARALLEL_MAX_SERVERS** Sets the maximum number of parallel query slaves (like dedicated servers but for parallel operations) your instance will ever have.
- **PARALLEL_MIN_PERCENT** Useful if you would like to receive an error message when you request a specific degree of parallelism but insufficient resources exist to satisfy that request. You might use this to ensure that a process that takes days unless it gets what it wants doesn’t run unless it gets what it wants.
- **PARALLEL_MIN_SERVERS** Sets the number of servers to start when the instance starts and to keep started permanently. Otherwise, you may need to start the parallel processes in response to a query.
- **PARALLEL_THREADS_PER_CPU** Determines the default degree of parallelism and contributes to the adaptive parallelism algorithms, to determine when to back off on the amount of parallel resources.
- **RECOVERY_PARALLELISM** For crash recovery, sets how many parallel threads should be used. This setting can be used to speed up the recovery from an instance crash.
- **FAST_START_PARALLEL_ROLLBACK** Sets how many processes would be available to perform a parallel rollback after the recovery takes place. This would be useful on systems where many long-running transactions are constantly processing (which would need correspondingly long rollback times in the event of an instance crash).

When Are Parallel Queries Useful?

A parallel query means the problem is really big. I find, most times, the problem shouldn’t be that big. For example, you are full-scanning a huge table because the predicate on the table involves a low-cardinality column (you are probably thinking that I’m going to suggest a bitmap index here). Well, you should probably be full-scanning a small partition, because the table should have been partitioned by that value (or it could that a bitmap index does come into play). In any case, look at all of the advanced features of the database for data warehousing before using parallel queries—maybe a materialized view, rebuilt in parallel, is the answer; maybe a bitmap join index does it.

Many problems are solvable via other database technologies. Parallel query is just one of the tools. Think of it as a power tool you might rent for a special project at home, not something you want to use casually.

Parallel query is something that I find no use for in a transactional system—just forget about it there. In a transactional system, you generally have a lot of users, running many concurrent sessions. Parallel query, which creates a lot of sessions to answer a question, just doesn't scale there. It does not make sense to let one user run a parallel operation when you have dozens of other users competing for the same resources.

But nothing is ever black and white—don't discount parallel query immediately after someone puts a "transactional" tag on your system. Most systems suffer from multiple-personality disorder. They are one thing by day, and another by night. It may well be that from midnight to 6:00 A.M., parallel query is a feature that is vital, but from 6:01 A.M. to 11:59 P.M., it is a tool that just doesn't make sense to use.

Parallel query is a feature I find useful in data warehouses where the concurrent user count is low (so there are plenty of resources to go around). If you have a 500 concurrent user data warehouse with 64 CPUs, parallel query probably is not going to apply. If you have a 5 concurrent user data warehouse with 64 CPUs, parallel query is going to be the coolest thing since sliced bread. It is all a matter of how many spare cycles you typically have when not using parallel query. If you have some, parallel query will help you use them up, which is a good thing—you want your system to be 100% utilized.



NOTE

I frequently hear from people who are concerned that their systems are peaking at 100% utilization from time to time. I ask them, "Well, if it were running at 90%, would you be able to put that 10% in the bank for later?" No, you cannot. It's best to use it now while you can.

So, my advice for parallel query is fairly easy and straightforward:

- Just set relevant tables to PARALLEL; do not specify a degree of parallelism.
- Use parallel automatic tuning.
- Look for opportunities to not have to use parallel query.

Parallel DML

Parallel DML (PDML) is really just an administrative feature similar to CREATE TABLE, CREATE INDEX, or loading. It is something you would use infrequently, to perform a large bulk UPDATE or DELETE one time to fix some legacy data. You may be using a parallel MERGE command to refresh data monthly (or weekly) in a data warehouse. However, since there are many caveats regarding PDML, I address it separately from the standard administrative functions I covered earlier in the "Parallel Administration" section.

It used to be that in order to make use of PDML in Oracle, you needed to have tables themselves partitioned, and the degree of parallelism was set to the number of underlying partitions. In Oracle9i Release 2 and up, this is no longer true, but that probably isn't relevant. The reason I say that is because, like parallel query, you would use PDML against very large

objects, and very large objects will be partitioned (at least, they would be if they were my large objects). So, even though you may modify in parallel objects that are not partitioned, it is not really that exciting.

The tables we want to use PDML on are so large that if we are not using partitioning, we should probably be adding it to our schema at this point. For example, instead of updating a billion-row table, you would CTAS in parallel a newly partitioned table that included the updated data in the SELECT statement itself.

There are many restrictions regarding PDML. The following are the major ones to consider:

- There can be no triggers on the tables being modified. In a data warehouse, this is generally okay. In a transactional system, if you attempt to use PDML to perform a mass fix of data, you must be aware of this.
- PDML cannot be replicated (since that is done via triggers).
- There are restrictions on the types of integrity constraints that are supported with PDML. For example, self-referencing constraints cannot be in place, since the table is being modified in parallel by many “sessions,” and the sessions cannot see other sessions’ changes.
- You cannot use database links (distributed transactions) with PDML.
- Clustered tables (B*Tree clusters and hash clusters) are not supported with PDML.

If you attempt a PDML operation and any of these rules are violated, you will not receive an error message. Rather, the statement will execute serially. You will want to monitor the parallel processing of your statement with the dynamic performance views, such as V\$PX_PROCESS.

DIY Parallelism

With DIY parallelism, we are asking not what Oracle can do for us, but rather what we can do with Oracle.


“I’ve been trying to resolve an issue whereby a PL/SQL procedure in a package takes a weekend to process 26,000 rows. It processes each of those 26,000 rows with reference to a specific period of time (one month), and for that it performs up to 40 calculations. Each of these calculations can involve a number of sums. So we have 26,000 rows, 40 calculations, each for every month from January 1999 to October 2002. It can take up to three hours to run one month, and that is perceived as a problem. We have a multi-CPU machine, but it only seems to be using one of them. Shouldn’t the machine be 100% used by this process? We have plenty of spare CPU and plenty of RAM. We just don’t seem to be able to use it.

I’ve spent lots of time over several days looking at Statspack reports, SQL_TRACE reports, and used DBMS_PROFILER, as you suggested. I’ve looked at init.ora parameters and modified some of those, also looking at the NT Performance Monitor. This time is the result of that tuning; it used to take longer! Statspack reports shows the biggest number of waits in a 10-minute period as 496 waits for log file parallel write, with a wait time of 22 cs. This doesn’t seem a big deal. There is spare memory, spare CPU capacity, and it doesn’t seem to be massively waiting on disk. So what IS taking the time?”


Well, what we have is the classic serial batch process. Here, parallel query, PDML, or whatever isn't going to do anything. They have a linear piece of code that processes 40 calculations a row at a time. They have 26,000 rows times 40 calculations. That is 1,040,000 calculations. In three hours, that is an average of 96 PL/SQL calculations per second. Coupled with the fact that this includes retrieving who knows how much data and doing some modifications to the database, this doesn't sound too bad. Remember that this is not 96 machine instructions per second (that would be really slow); this is 96 complex PL/SQL calculations that request data from the database and everything else.

Now, since there are no significant wait events, it must be the algorithm. It is the algorithm that is the issue here, not the ability of the database to process in parallel. No amount of parallel query will make this go faster (maybe slower, but not faster). The machine has not reached its limit; it has plenty of capacity. They need to change the algorithm to allow it to take advantage of this. My suggestion is to parallelize the problem. They have 26,000 rows. Divvy that up based on some predicate and run four or five copies of the stored procedure at the same time. They have plenty of extra CPU they aren't even using. Parallelize this operation over four "threads" and see what happens.

In many cases, when you see a batch process that looks like this pseudo code:

```
 Create procedure batch_process
As
Begin
  For x in ( select * from T )
  Loop
    Process...
    Update T for that row or Insert that row
    Elsewhere...
  End loop
End;
```

And you cannot just make it a single SQL statement due to the complexity of the process, my suggestion to make it faster is to have more processes working on it. Even in a single-CPU machine, there are frequently spare CPU cycles while you are processing. You spend some amount of time waiting for I/O. Let another process make use of that. In this case, the solution to speed this up over three times is as follows:

```
 Create procedure batch_process(start_id in number, stop_id number)
As
Begin
  For x in ( select *
            from T
            where id > start_id and id <= stop_id)
  Loop
    Process...
    Update T for that row or Insert that row
    Elsewhere...
  End loop
End;
```

And to run this procedure in N different sessions, feeding each session a different, nonoverlapping set of IDs. Generally, this parallelization can be done with little to no modification of the existing algorithms, short of adding these parameters and predicate.

All that is left now is to generate N sets of IDs to do this in parallel. Using the new PERCENTILE_DISC function in Oracle9i, we can do this easily. PERCENTILE_DISC is a statistical function that will return a discrete (hence the DISC) value from a set. You pass the percentile function a percentage X , and it will return a value such that $X\%$ of the rows in the table have a value less than that value.

For example, we have a table for testing called BIG_TABLE. It has 1.8 million rows in it. Suppose we want to run the above process against it. We need four ranges. We can get them right from the table itself using this technique:

```
big_table@ORA920> column c0 new_val c0
big_table@ORA920> column c1 new_val c1
big_table@ORA920> column c2 new_val c2
big_table@ORA920> column c3 new_val c3
big_table@ORA920> column c4 new_val c4
big_table@ORA920>
big_table@ORA920> SELECT min(object_id)-1 c0,
2 PERCENTILE_DISC(0.25) WITHIN GROUP (ORDER BY object_id) c1,
3 PERCENTILE_DISC(0.50) WITHIN GROUP (ORDER BY object_id) c2,
4 PERCENTILE_DISC(0.75) WITHIN GROUP (ORDER BY object_id) c3,
5 max(object_id) c4
6 FROM big_table
7 /
```

| C0 | C1 | C2 | C3 | C4 |
|----|--------|--------|---------|---------|
| 0 | 458464 | 916928 | 1375392 | 1833855 |

And then this query using those values shows we've covered the entire range of values in the table. We can use c0 through c4 as inputs into a procedure to divvy up the table to be processed in parallel:

```
big_table@ORA920> select
2 sum( case when object_id > &c0 and object_id <= &c1
3 then 1 else 0 end ) range1,
4 sum( case when object_id > &c1 and object_id <= &c2
5 then 1 else 0 end ) range2,
6 sum( case when object_id > &c2 and object_id <= &c3
7 then 1 else 0 end ) range3,
8 sum( case when object_id > &c3 and object_id <= &c4
9 then 1 else 0 end ) range4,
10 count(*)
11 from big_table
12 /
```



```
RANGE1 RANGE2 RANGE3 RANGE4 COUNT(*)
```

```
-----
```

```
458464 458464 458464 458464 1833856
```

In this case, we were able to achieve between a 250% and 300% increase in throughput, without doing much work at all.

Now, bear in mind, you may not be able to immediately apply this to your process. It does depend on the underlying logic of the PL/SQL, C, Java, or other language routines. The point is that you should not overlook this approach. In fact, the best time to look at this approach is well in advance of tuning an application. This approach works best when designed in from the very beginning.

Parallel Processing Wrap-up

Parallel query is not a fix to a bad design; a good design is a fix for a bad design. If a physical redesign would obviate the need for parallel query, by all means, that would be the method to take. For example, it could well be that introducing partitioning in a manner that allows partition elimination to remove 90% of the data from consideration would obviate the need to run a parallel query. Look at the question and justify parallel operations on it, not the other way around.

In the early days of data warehousing, we saw a lot of large databases with a small number of users. Parallel query was a huge deal back then: small number of users + big machine = excess capacity. Today, warehouses are so mainstream, so functional, that many times, we have as many users connected to them as we do to our transactional systems. The abilities of parallelism may well be limited to administrative features at that point. There just won't be sufficient capacity to do otherwise.

The best sources for more information about parallel processing are the Oracle documentation once again. Here, I will point out a couple of more obscure pieces that you might want to look at in addition to the standard fare:

- *Oracle Concepts Guide* But, of course, I recommend this guide. It includes a very good chapter on parallel processing in general and parallel features in Oracle.
- *Data Warehousing Guide* Here, you will find two chapters on parallelism. They cover when to use it to how to use it in your data warehouse.
- *Supplied Packages Guide* A hidden nugget is the DBMS_PCLXTUIL package. This package implements a "more parallel" local index create operation.
- *Supplied Packages Guide* The DBMS_STATS package allows you to gather statistics on a table in parallel.
- *Administrators Guide* This provides information about managing processes for parallel execution, parallelizing table creation, index creation, PDML, and so on.

Summary

In this chapter, we took a look at how the architecture of Oracle can affect your application. It answered questions from simply, "How should I connect to the database?" to "Will partitioning make my database operations faster?"

Many of these topics could be books unto themselves. Much of the point here is that you need a good understanding of how these tools—these architectural implementations Oracle provides—work in order to be successful. For example, what if you just turn on parallel query to fix a performance problem? That isn't the answer. You will not accomplish your goal. Use parallel query wisely, in appropriate places, with a firm understanding of how it works and what it does, and you can reap incredible benefits.

Much like the old saying, "When all you have is a hammer, everything looks like a nail," these architectural implementations of Oracle's are simply a wide variety of tools in your toolbox. When used correctly, they work well. When used improperly, or with insufficient training, testing, or knowledge, they can be deadly.