2

Using SQL

The end users of the applications you develop are almost never aware of the code used to retrieve their data for them, or insert and update changes to the data back into the database. Your application acts as a black box, handling all the interactions with the underlying database.

In some ways, we developers have our own black box, although the box is much smaller. We send SQL statements off to the database and the database does its magic. The aim of this chapter is to take a close look at what the Oracle database does in response to those SQL statements. Although you may have some understanding of what takes place on the Oracle server, this chapter also provides some practical explanations that will help you to avoid some of the most common pitfalls facing developers accessing an Oracle database.

This chapter concentrates on what is going on in the Oracle server, rather than on the code you use to trigger those actions from your application. These actions are the Oracle database responding to your requests, regardless of what programming language you use. Most languages use some form of application programming interface (API) calls to use SQL against the Oracle database, and those calls and code will be introduced later in this book.

The Processing Cycle for SQL Statements

To process an SQL statement you submit to your Oracle database, the database goes through several phases as part of the execution of the statement:

- Connect to the database, to establish communication between your client application and the Oracle database server.
- Create a cursor, to hold the context for an SQL statement.
- Submit an SQL statement to the Oracle database for processing, which includes checking the validity of the statement and potentially creating an optimized execution plan for it.
- Receive data back from the Oracle database in response to the submission of a query.

This cycle of events is shown in Figure 2-1, with mandatory events shown in white and potentially optional events shown as shaded. Each of these phases is discussed in more detail later in the chapter.

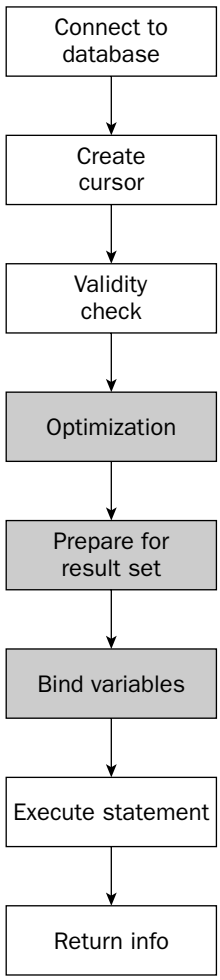

Figure 2-1: The processing cycle for an SQL statement.

Connecting to a Database

The first step in accessing an Oracle database from your application is to establish a connection to the database. This connection is the path that acts as the path from your client application to a shadow process in the Oracle database that handles the SQL requests from your application.

Normally, a connection goes from a client machine over a network to a server machine. The connection is implemented on top of a network protocol, such as TCP/IP. The connection request is actually received by the Oracle Listener, which listens on a port for connection traffic. Once the connection is established, the Listener passes requests to the shadow process associated with the connection, as shown in Figure 2-2.

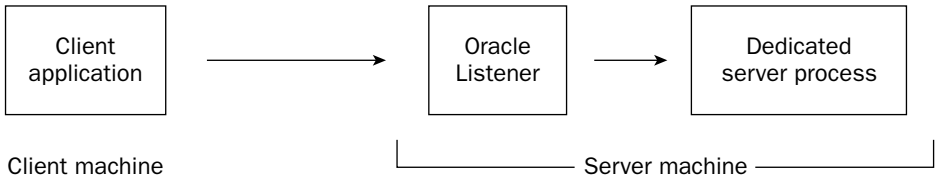

Figure 2-2: Establisihng a connection to an Oracle database from a client application.

If you are using Java as your program language, the connection will be executed with a driver, which is software designed to handle the complexities of communication over the network. The various types of Java drivers are described in Chapter 7, "Installing Oracle."

A connection to an Oracle database is always created on behalf of a user. Consequently, establishing a connection requires identifying and authenticating that user with a username and password. The details of authentication are described in Chapter 5, "Oracle Security."

A connection to an Oracle database is referred to as a *session*. The session is the overriding context for all SQL interactions that take place. When a connection is either terminated or is lost for any other reason, the context for that session, including any information in any uncommitted transactions within that session, is lost. Chapter 3, "Handling Multiple Users," describes in detail the use of transaction in an Oracle database.

Every session is supported by a shadow process on the Oracle server. Normally, this means that every session has its own process. But, as you can imagine, each shadow process uses some server memory resources, so the scalability of an individual Oracle instance might be limited by the number of sessions it can support. To address this issue, Oracle has a feature known as *shared servers*, which was referred to as multithreaded servers, or MTS, before Oracle9*i*.

Shared servers add another layer to the connection architecture, as shown in Figure 2-3. When a connection comes into the Listener, it passes the request to a dispatcher. The dispatcher assigns the request to a session that can be shared. Once the request is completed, the session becomes available to service other requests.

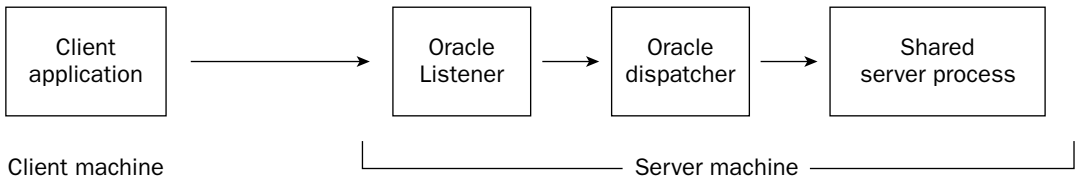

Figure 2-3: Shared servers.

Use of shared servers is completely transparent to your application. A single Oracle instance can use a combination of shared servers and dedicated servers — the instance has different connection identifiers for a dedicated session or a shared session.

When should you use a shared server? As with most issues involving performance, the answer depends on the particulars of your particular implementation.

If you are running out of memory because of a large number of connections, shared servers can help to address the problem by lowering the overall amount of memory required for sessions. Obviously, some overhead is also involved with the use of a dispatcher and the functions it performs. This overhead can be balanced against the more limited use of resources required by the shared server architecture.

Typically, a shared server can do the most good for applications that require periodic access to the database, since the connections used by these applications will not be performing work much of the time, and as such they are candidates to be shared with other sessions. A browsing application would fit this description, while a heads-down transaction processing application would not. The good news is that you can switch between using shared sessions and dedicated sessions by simply changing the connection parameters, so it is fairly easy to test the effect of shared sessions on your overall application performance.

You have to establish a connection to your Oracle database instance before you can send any SQL to the instance, but you do not necessarily have to create a connection each time you want to use one. You can reuse connections inside of an application, or a portion of an application, or you can use connection pooling to maintain a set of connections that a user can grab when he or she needs it. Using a connection manager with Java is covered in Chapter 18, "Introduction to Java Database Programming."

Establishing a Cursor

Once a connection is established, the next step is to open a *cursor* for your SQL statements. A cursor is a connection to a specific area in the Program Global Area (PGA) that contains information about a specific SQL statement.

> *For a description of the PGA, please refer to Chapter 1, "Oracle Architecture and Storage."*

The cursor acts as the intermediary between the SQL statements your application submits and the actions of the Oracle database as a result of those statements. The cursor holds information about the current state of the SQL statement, such as the parsed version of the SQL statement. For statements that return multiple rows of data, the cursor keeps track of where you are in terms of returning rows from the result set. This important piece of information is also called a cursor, in that the cursor for a particular result set is the pointer to the current row. But the cursor exists before any rows have been returned for a query, which is why advancing the cursor to the next row at the start of retrieving data places the cursor on the first row in the result set.

You don't necessarily have to explicitly open a cursor to execute an SQL statement, because the Oracle database can automatically take care of this for you. As with most aspects of application programming, explicit control over cursor creation and use will give you more options for implementing your logic efficiently. You may want to open more than one cursor to improve the operation of your application. For instance, if you have an SQL statement that is repeatedly used, you might want to open a cursor for the statement so that Oracle will not have to re-parse the statement. Of course, the Oracle database uses its own internal caches to hold parsed versions of SQL statements also, so creation of individual cursors may not be necessary for optimal performance in many cases.

Submitting SQL Statements

At this point in the processing cycle, your application has established a connection to the database and allocated a cursor to hold the context for your SQL operations. The next step is to submit an SQL statement to the Oracle database.

Submitting a statement is a single line of application code, but the Oracle database performs a series of actions in response to this submission before it responds to the request. The individual actions performed depend on the type of SQL statement submitted — a data definition language (DDL) statement for creating or modifying database objects, a write operation (`INSERT`, `UPDATE`, or `DELETE`), or a query using the `SELECT` verb. The actions taken for each of these are detailed in the following table.

Action	DDL	Write operation	Query
Validate	X	X	X
Optimize		X	X
Prepare for results			X
Bind variables	X	X	X
Execute statement	X	X	X
Return success code	X	X	X
Return result count		X	
Return rowset			X

Each of these actions is described in the following.

Check Validity of Statement

Parsing is the process of checking an SQL statement to ensure that it is valid, as well as creating an optimal execution plan for the statement. Validity checks are used to ensure that an SQL statement will be able to be executed by the Oracle database.

The first step in the process is to check the syntax of the SQL statement. This syntax check includes checking the validity of the actual keywords used in the statement as well as whether the table and column names refer to valid entities that the submitting user can access properly. If you misspell a verb (`SELECT`), a column, or a table name, or try to access a database object you do not have privileges to access, Oracle will return an error at this phase.

For DDL and write operations, the statement is now loaded into the shared SQL area, since the statement is ready to run. For queries, the statement has to go through an additional step of optimization.

After an SQL statement has been checked, the Oracle database computes a hash algorithm to check whether a version of the statement exists in the shared SQL area. As mentioned in Chapter 1, the Oracle instance includes an area for shared SQL statements in the System Global Area memory (SGA) area. Since the next step in the processing cycle, optimization, can be a relatively expensive operation in terms of resource usage, Oracle can save a lot of work if it can avoid that step. If the statement is located, Oracle just uses the execution plan for the found statement.

If Oracle cannot find the statement in the shared pool, the instance proceeds to the next step of optimization. This type of parse is called a *hard parse*. If Oracle can find the statement in the shared pool, it simply retrieves the execution plan for the statement in what is called a *soft parse*. Later in this chapter, we explain how the use of bind variables can improve performance by increasing the chances that shared SQL will be reused.

Oracle will not bother to check for execution plans for DDL statements in the shared SQL pool, since these plans are not placed into the pool. The advantages of soft parses come into play when SQL statements are repeatedly executed, and since DDL statements are, in almost all cases, done only once, Oracle properly assumes that this portion of the process would be useless overhead for DDL.

Optimize

One of the great advances made by relational database technology was to give users the ability to request data without having to specify how the data was to be retrieved. Rather than telling a database exactly what structures to use to get to the data, you can simply send an SQL statement to the database and let Oracle figure out the most efficient way to retrieve the data.

In a complex production database, there may be many, many possible retrieval paths that could be used to retrieve the data for a query. The Oracle query optimizer analyzes the various paths and evaluates them in terms of how efficient they should be. This analysis takes into account data structures, such as indexes; selection criteria and how they might affect finding the requested rows' as well as environmental conditions, such as the amount of CPU and memory available.

The end result of the optimization process is an execution plan, which details exactly how your Oracle instance retrieves the data requested by the submitted SQL statement. Optimization is a nontrivial topic, covered at length in Chapter 28, "Optimization." At this point, you only have to know that optimization is both crucial to the efficient performance of your Oracle database and often fairly time-consuming. Because of this, the reuse of already optimized statements retrieved from the shared SQL pool can, in many cases, significantly improve the performance of your Oracle database.

Queries will always require optimization. Write operations may also require optimizations, since the first part of many write operations involves retrieving the rows that will be used in the write operation. DDL statements are not optimized.

Prepare for Results

If an SQL statement is a query, the Oracle database has to do some additional work to prepare for the return of the result set. This work involves describing the results of the query in terms of data type, data length, and data names for the query and associating each returning item with a variable for return.

Bind Variables

Once Oracle has prepared the SQL statement, it binds any variables into the SQL statement. A *bind variable* is a value in an SQL statement that is not known until the statement is run. In an SQL statement, the bind variable acts as a placeholder for this subsequent data. In Oracle SQL syntax, a bind variable is indicated by a colon (`:`) before the variable name.

For instance, the following SQL statement is used to insert data into the EMP table.

```
INSERT INTO EMP(ENAME, DEPT, SAL) VALUES(:sEname, ;sDept, :nSal);
```

The values listed in the `VALUES` clause are bind variables — the actual data used is sent later. You can use bind variables for values, selection criteria in the `WHERE` clause, and many other places in an SQL statement. You cannot use bind variables for SQL keywords.

When Oracle checks the syntax of your submitted statement, the validity of the value of bind variables cannot be checked. Because of this, a statement with bind variables could compile properly and still give you runtime errors. In the previous example, the value submitted for the `:nSal` bind variable could end up being invalid for the data type defined for the `SAL` column.

Bind variables are useful from both the logical and execution standpoint. In terms of logic, you can create a small set of SQL statements that can be used repeatedly to perform the same function for different data, and less SQL syntax means a smaller number of possible errors. In terms of performance, bind variables can increase the use of shared SQL — an important topic that is covered in detail at the end of this chapter.

Execute

Once the Oracle database has performed all of the steps detailed previously, the instance can execute the SQL statement using the execution plan that has been prepared. This execute phase is where Oracle actually performs the operation indicated by the submitted SQL statement.

Returns

Oracle returns information about each SQL statement it processes. All SQL statements return a simple boolean value to indicate whether the statement was successfully executed. For a write operation, Oracle also returns an integer that indicates how many rows were affected by the operation.

Queries return result sets — groups of rows that have been retrieved to satisfy the query. Retrieving those rows is the subject of the next section.

Receiving Data

A query is a request for data. In response to a query, the Oracle database prepares a result set that satisfies the conditions of the query. An application retrieves the rows of the result set by performing a series of fetch operations to return rows.

When you are using Java to access Oracle data, the Java API implements a pre-fetch. A pre-fetch returns a designated number of rows with a fetch command, when needed. By default, the pre-fetch returns 10 rows, but you can change the default for an individual connection programmatically.

When a fetch is first executed for a result set, the return includes 10 rows from the result set. For the next nine fetch operations, the data is already residing at the client. In this scenario, the 11th fetch result would retrieve another 10 rows.

Performance Considerations

The previous description of the execution cycle for SQL statements provides a brief description of how SQL statements are processed. But this factual description does not really help you understand the things you can do to create applications that will run effectively and perform optimally.

Understanding the SQL processing cycle is a prerequisite to understanding how to get the most out of your Oracle database. The rest of this book is essentially commentary on this topic. Performance tuning is an enormous subject in itself. This book devotes a chapter exclusively to the topic (Chapter 28), and many books have been devoted to achieving optimal performance with your Oracle database. Optimal performance comes from optimal design, optimal implementation, and optimal use of resources in terms of your particular logical and physical environment. In this introductory chapter, we cannot give you comprehensive advice on performance tuning, but there are three areas that have a broad effect on performance for all applications — how data is retrieved, the effect of bind variables on the processing cycle for SQL in an Oracle database, and the use of parallelism to improve performance.

Retrieval Performance

The final, and in many ways the most important, event in the SQL processing cycle is the return of data to the application that requested it. On one level, the speed of this final step is determined by your network, since the amount of bits that can move from the server to the client is ultimately limited by the amount of network bandwidth. The effects of this potential limitation cannot be overcome. But you can affect how much work the Oracle database performs before it starts returning data to the user.

Well, how much work does Oracle have to do before it returns data? You would think the answer to this question would be self-evident — "As much as it needs to do to get the data." But how much data does Oracle need to get before it *starts* to send data back to the application?

As developers, we tend to think of the end results of an action. A user requests data, so he or she obviously must want all of that data to do what he or she has to do. But users tend to be more, shall we say, immediate in their outlook. For us, performance is the time it takes to complete an SQL operation, such as a query. For a user, performance is how long they wait before something comes back to them.

You can take advantage of this dichotomy by setting the way that Oracle returns data. You can specify that Oracle should start returning rows to the user as soon as it gets the rows, or you can specify that Oracle will only start returning rows to the user once it has collected all the rows for the query.

You instruct your Oracle database as to which approach to take by setting a parameter called `OPTIMIZER_MODE`. The two settings for `OPTIMIZER_MODE` that are relevant to this example are `ALL_ROWS` and `FIRST_ROWS`, which tell Oracle to only return data once all rows have been fetched or as soon as it can, respectively. You can also use either one of these values as a hint for a particular query.

For more on optimization and hints, please refer to Chapter 28, which is dedicated entirely to the subject.

The best choice for this parameter obviously depends on your application. If a user is unable to do any work until he or she receives all the data, or if you don't want the user to do any work until he or she receives all the data, the `ALL_ROWS` parameter is the right choice. In applications that typically fetch data for the user to peruse and possible use, `FIRST_ROWS` may deliver better perceived performance without much logical downside. If your application is not retrieving large amounts of data, this particular optimizer choice shouldn't really affect performance.

Regardless of the setting of this parameter, there are some times when Oracle will properly wait until it has retrieved all rows until it returns any rows. One case is when a query includes aggregate values. Oracle knows that it has to get all the rows before it can calculate aggregate values, so no rows will be returned until all rows have been retrieved and the calculations performed.

Another case is when you ask for the rows to be returned in sorted order. Normally, Oracle cannot return the first rows until the sort has been performed, since the sort determines what the first row is, not the order that the rows are retrieved from the database. The exception to this rule is when the query has requested a sort order that is already implemented in an index. For instance, a user may request employee names in alphabetical order based on the last name, and there is an index that sorts the rows on that criterion. The Oracle database knows that the index has already sorted the rows, so it does not have to sort them and the first rows can be returned as soon as they are retrieved.

Using Bind Variables

The previous performance tip centered around how the Oracle database returns data to the user. This next tip has to do with the exact way that the Oracle database processes SQL statements. Unlike the previous tip, which did not affect your application code, this area requires that you implement your code in a particular way to reduce the use of resources and improve the performance of your Oracle database.

Earlier in this chapter, we discussed how Oracle stores SQL statements in the shared pool area of memory. When the Oracle database receives an SQL statement, it checks the shared pool to see if an optimized execution plan already exists for the statement. If the plan exists in the pool, the plan is simply retrieved from memory, which is much more efficient than reoptimizing the statement. Use of the shared pool helps Oracle to scale for large numbers of users and perform well with any load.

If you have to reoptimize every statement, each statement adds to the overall workload of the target Oracle database. This means that the overhead on your system increases with the number of statements and that, eventually, you will run into resource limitations that will decrease performance. If every statement has to go through the complete cycle of execution, your scalability will be limited.

Fortunately, a real-world application is not a series of unique SQL requests. In fact, most applications use the same SQL statements over and over again. Theoretically, this would mean that the repeated SQL will be picked up from the shared pool, which is much less expensive from a resource standpoint. Since Oracle needs exclusive access to some resources when optimizing an SQL statement, optimizing SQL statements cannot be done in parallel, so the more optimizations Oracle has to do, the greater the elapsed time for a query to be processed.

The way to reduce this potential bottleneck is to help your Oracle database to avoid performing hard parses as much as possible. You can help this to occur by using bind variables to help re-use SQL statements.

Remember that the method used to identify a statement is a comparison of the hash algorithm created from the statement. The value of this hash is derived from the characters in the statement.

Consider the following two statements:

```
SELECT ENAME FROM EMP WHERE EMP_ID = 7

SELECT ENAME FROM EMP WHERE EMP_ID = 5
```

You can quickly tell that these two statements should use identical optimizer plans. If the execution plan for the first statement is still in the shared pool, Oracle should use it, right? Unfortunately, Oracle may not be able to find the plan, since the hash value created by the second statement will very likely be different from the hash value created by the first statement, based on the different characters.

At this point, bind variables come to the rescue. A bind variable is a placeholder in an SQL statement. Oracle can process a statement containing a bind variable but can only execute the statement when it receives the value of the variable.

You identify a bind variable by preceding it with a colon in the SQL statement. The previous SQL statements could both be represented by the single SQL statement following, which uses a bind variable.

```
SELECT ENAME FROM EMP WHERE EMP_ID = :n_EmpID
```

If you use this syntax twice, instead of using the two different SQL statements shown previously, Oracle will be able to retrieve the execution plan from the shared pool, saving lots of resources.

The statements used to illustrate the use of bind variables previously are not that complex, so you may doubt how much the use of bind variables could help. But remember that your own SQL statements are considerably more complex, where the creation of an execution plan could likely be the biggest resource hog in the entire sequence of SQL processing. Also, remember that not only will your application likely repeat the same SQL statement over and over again but that this effect is multiplied by the use of your application by many different users.

Those of you who think of yourself as primarily as programmers and don't give no never mind about databases may be rising up in protest at this point. There is some common wisdom that says the code required to use bind variables, a `PreparedStatement` call in Java, executes slower than the code to execute a hard-coded statement, a `Statement` call in Java. We certainly can't argue this simple fact (although others have), but remember that there is more to performance than simply the time required to execute a call.

The use of bind variables can have an enormous impact on the time Oracle needs to return data to the user. Since performance is an aggregate, it doesn't make sense to save time in the execution of a single call if those savings are going to be outweighed by slower performance from the database.

The bottom line is that you should train yourself to use bind variables whenever you have SQL statements that only differ in the value of a particular item in the statement and that are repeatedly executed in your application.

The use of bind variables is so important that Oracle introduced a setting called `CURSOR_SHARING`. You can set the value of this parameter to specify what type of SQL statement should be forced to share cursors — for instance, if the SQL statements are exactly alike except for literal values. You can set the value for this parameter in either the initialization file for your Oracle database (`INIT.ORA` or the shared parameter file `SPFILE`) or with the `ALTER SESSION` command for your session; Oracle will automatically search out places where a literal could be substituted with a bind variable and perform the translation for you. This

setting is handy, but it works across all literals, including some that are truly serving the function of literals in that they represent a value that does not change. Substituting these true literals with bind variables could cause Oracle to make an inappropriate optimizer decision. Training yourself to properly use bind variables is a better route to creating applications that perform and scale well.

Parallel Operations

Have you moved recently? If you moved yourself, you probably found that having more people working together resulted in reducing the overall time that it took to complete the job. Of course, there was no doubt some overhead involved in getting everyone to work together, and there may have been a point where getting even more people involved actually increased the time to complete the job.

Oracle implements a similar type of work sharing called parallelism. Parallelism makes it possible for Oracle to split up the work of a specific SQL statement among multiple worker tasks to reduce the elapsed time to complete the SQL operation.

Oracle has had the capability since version 7, although the scope of this functionality has been continuously improved. Oracle now supports parallel operations for queries, updates, and inserts, as well as for backup, recovery, and other operations. Parallelism for batch-type jobs like loading data is also supported. Parallel operations are only available with the Enterprise Edition of Oracle 10*g* Database.

Oracle implements parallelism on the server, without requiring any particular code to work. This final section will look at how Oracle implements parallel operations to improve performance.

How it Works

Parallel execution in Oracle divides a single task into smaller units and co-ordinates the execution of these smaller units. The server process for a user becomes a parallel coordinator, rather than simply executing the SQL statement.

The Oracle database provides parallel operations for each step in the processing of a query. Oracle can provide parallelism for data retrieval, joins, sorting, and other operations to provide performance improvements across the whole spectrum of SQL processing.

You have to use the `ALTER SESSION ENABLE PARALLEL` call, followed by the keyword `DML` or `DDL` to turn on parallel operations. The corresponding command `ALTER SESSION DISABLE PARALLEL` will turn off parallel operations for the session. You can specify a degree of parallelism with these statements.

Managing Parallel Server Processes

The Oracle database has a pool of parallel server processes available for execution. Oracle automatically manages the creation and termination of these processes. The minimum number of parallel processes is specified in the initialization parameter `PARALLEL_MIN_SERVERS`. As more parallel execution processes are requested, Oracle starts more parallel execution processes as they are needed, up to the value specified in the `PARALLEL_MAX_SERVERS` initialization parameters. The default for this parameter is 5, unless the `PARALLEL_AUTOMATIC_TUNING` parameter is set to TRUE, in which case the default is set to the value of the `CPU_COUNT` parameter times 10.

Caveats

The discussion of parallel processing above is extremely short and is meant as a mere introduction to the capabilities of parallel processing You should be aware that not every situation can benefit from parallel processing. For instance, a particular operation might be I/O bound, where the performance bottleneck is the interaction with the data on disk. This type of operation will not benefit from parallel processing, which can only reduce the overall CPU time spent by dividing the work up between multiple processes. In fact, this type of operation may actually get slower, as the process of coordination of different parallel processes can add to the already overburdened I/O workload.

In addition, if your SQL statement is already running optimally, adding parallelism may not help the performance, and may harm it.

The saving grace with parallelism is that you can easily run SQL operations with parallelism on or off, and with varying degrees of parallelism. Although you should be aware of the potential provided by parallel operations, you should not see them as a panacea for all performance problems—you should test the potential effect of parallelism if you have any doubts as to whether this option can help in a particular scenario.

Summary

This chapter covered the steps that your Oracle database performs when it receives an SQL statement from your application. Although these steps are not visible to either the user or yourself, understanding how Oracle works internally can help you to design and implement applications effectively.

For SQL queries, Oracle receives the query, either parses and optimizes it or retrieves an already prepared version of the query, executes the statement, retrieves the desired rows, and then returns those rows to the user. Write statements and data definition language do not perform all of these steps.

Once you understand that an SQL query can either be parsed and optimized, or its prepared version simply retrieved, you can understand why the proper use of bind variables can have an important impact on the performance of your applications.

Finally, this chapter gave an overview of parallel operations and how, in some circumstances, they can provide better response time for your SQL operations.