CHAPTER 1

The Role of PL/SQL in Contemporary Development
When building systems, it is critical to ensure that the systems will perform well. For example, after you build a web-based application, a user who clicks a button on a screen expects to receive a response within a reasonable amount of time. If, instead of waiting 1 to 2 seconds, the user needs to wait 1 to 1.5 minutes for a response, something is definitely wrong. The first instinct of most web developers is to call the DBA and say that the database is slow. However, that might not be the reason for the poorly performing application at all. This chapter discusses system performance tuning in general, where PL/SQL fits into the picture, and how to determine whether the problem lies within the realm of PL/SQL.

Many end users and web developers may be surprised to learn that the reason for an application’s poor performance is not related to problems in the database in general, or the PL/SQL code in particular. Often, the PL/SQL code is the least of the problems. The following sections describe how you can determine whether the PL/SQL code is the cause of poor performance and, if so, whether tuning the code will ameliorate the performance issues.

**Typical Web Application Process Flow**

Poorly written server-side code and a badly designed database will make any application run slower, but improving the performance of a slow running web application requires examination of the entire system, not just the database. A typical three-tier web application structure is shown in Figure 1-1. (The numbering of the areas in the diagram will be used throughout this chapter, so study the diagram closely.)
As shown in Figure 1-1, there are numerous possible places for web applications to experience bottlenecks or performance killers, as described in the following nine-step process:

- **Step 1** Code and operations are executed on the client machine. When a user clicks a Submit button, data is collected and bundled into a request that is sent to the application server.
- **Step 2** The client request is transmitted to the application server.
- **Step 3** Code in the application server is executed as a formulation of the client request to retrieve information from the database.
- **Step 4** The client request is transmitted from the application server to the database.
- **Step 5** The database receives and processes the information and prepares it for return to the application server.
- **Step 6** Information is transmitted over an internal network from the database to the application server.
- **Step 7** The application server processes the database response and prepares the response transmission to the client machine.
- **Step 8** Data is transmitted from the application server to the client machine.
- **Step 9** The client machine processes the returned request and renders the application page in the browser.

**Web Application Performance Problem Areas**

There is a temptation to focus tuning efforts on the database only, by looking at parameters, SQL queries, and PL/SQL code. However, tuning solely in the database only helps with Step 5 and ignores all of the other places where performance can degrade. This section describes how problems can occur at each step in the process.

**Step 1: Client Machine Performance Problems**

The formulation of a request in the client machine is usually the least likely source of system performance problems. However, it should not be dismissed entirely. In many commonly used modern system architectures, it is possible to place so much code in the client machine that a significant amount of time is required before the request is transmitted to the application server. This is particularly true for underpowered client machines with inadequate memory and slow processors.
Step 2: Client Machine to Application Server Transmission Problems

As is true for the client machine itself, the transmission between the client machine and the application server is a less common cause of slowly performing web applications. However, if the client machine is attempting to transmit a large amount of information, the time required to do so over the Internet may increase. For example, uploading large files (such as images) or transmitting a large block of data may slow down performance.

Step 3: Application Server Performance Problems

The application server itself rarely causes significant performance degradation. For computationally intensive applications such as large matrix inversions for linear programming problems, some performance slowdowns can occur, but this is less likely to be a significant factor in poorly performing applications.

Step 4: Application Server to Database Transmission Problems

Transmission of data from the application server to the database with 1 Gbps or better transmission speeds might lead you to ignore this step in the process. It is not the time needed to move data from the application server to the database that is the primary issue; rather, it is the time required to switch contexts from the application server to the database that is significant. As a result, a large number of requests between the application server and the database can easily add up to a significant source of performance degradation.

The trend in current web development is to make applications database-agnostic. This sometimes results in a single request from a client machine requiring many requests from the application server to the database in order to be fulfilled. What needs to be examined and measured is the number of round-trips made from the application server to the database.

Inexpert developers may create routines that execute so many round-trips that there is little tuning that a DBA can do to yield reasonable performance results. It is not unusual for a single request from the client machine to generate hundreds (if not thousands) of round-trips from the application server to the database before the transmission is complete. A particularly bad example of this problem encountered by the authors required 60,000 round-trips. Why would this large number be needed? Java developers who think of the database as nothing more than a place to store persistent copies of their classes use Getters and Setters to retrieve and/or update individual attributes of objects. This type of development can generate a round-trip for every attribute of every object in the database. This means that inserting a row into a table with 100 columns results in a single INSERT followed by
Chapter 1: The Role of PL/SQL in Contemporary Development

99 UPDATE statements. Retrieving this record from the database then requires 100 independent queries.

In the application server, identifying performance problems involves counting the number of transmissions made. The accumulation of time spent making round-trips is one of the most common places where web application performance can suffer.

Another major cause of performance problems can occur in the network firewalls where the application server and the client are in different zones with packet inspection in between. For normal applications, these activities may not be significant, but for large, data-transfer-oriented applications, this activity could cause a serious lag. One such example could be a document management system where entire documents are loaded from client machines to the application server.

Step 5: Database Performance Problems

In the database itself, it is important to look for the same things that cause client/server applications to run slowly. However, additional web application features can cause other performance problems in the database.

Most web applications are stateless, meaning that each client request is independent. This leads to the loss of already gathered session-level information accumulated in global temporary tables and package variables. Consequently, when a user logs in to an application, the user will be making multiple requests within the context of the sign-on operation (logical session) to restore information that was already gathered by previous requests.

The information pertaining to the logical session must be retrieved at the beginning of every request and persistently stored at the end of every request. Depending on how this persistence is handled in the database, a single table may generate massive I/O demands, resulting in redo logs full of information, which may cause contention on tables where session information is stored.

Step 6: Database to Application Server Transmission Problems

Transferring information from the database back to the application server (similar to Step 4) is usually not problematic from a performance standpoint. However, performance can suffer when a Java program requests the entire contents of the table instead of a single row. If the entire contents of a database table with a large number of rows are brought into the middle tier and then filtered to find the appropriate record, performance will be inadequate. During development (with a small test database), the application may even work well as long as data volumes are small. In production (with larger data volumes), the amount of information transferred to the application server becomes too large and everything slows down.
Step 7: Application Server Processing Performance Problems
Processing the data from the database can be resource-intensive. Many database-agnostic Java programmers minimize work done in the database and execute much of the application logic in the middle tier. In general, complex data manipulation can be handled much more efficiently with database code. Java programmers should minimize information returned to the application server and, where convenient, use the database to handle computations.

Step 8: Application Server to Client Machine Transmission Problems
This area is one of the most important for addressing performance problems but often receives the least attention. Industry standards often assume that everyone has access to high-speed networks so that the amount of data transmitted from the application server to the client is irrelevant. Applications with a very rich user interface (UI) create more and more bloated screens of 1MB or more. Some available partial-page refresh capabilities mitigate this problem somewhat by reducing the amount of information that needs to be transmitted when only part of the screen is being refreshed.

Transmission between the application server and the client machine is one of the most frequent causes of poor web application performance. If a web page takes 30 seconds to load, even if it is prepared in 5 seconds rather than 10 seconds, users will not experience much of a benefit. The amount of information being sent must be decreased.

Step 9: Client Machine Performance Problems
How much work does the client machine need to do to render a web application page? This area is usually not a performance killer, but it can contribute to poor performance. Very processing-intensive page rendering can result in poor application performance, especially on under equipped client machines.

Finding the Cause of Slowly Performing Web Applications
To identify performance bottlenecks, timers must be embedded into a system to help ascertain at which of the nine possible places the application performance is being degraded. Most users will say “I clicked this button and it takes x seconds until I get a response.” This provides no information about which area or combination of areas is causing the slow performance.
Chapter 1: The Role of PL/SQL in Contemporary Development

Strategically placed timers will indicate how much time is spent at any one of the nine steps in the total process.

**Using Timers to Gather Data About Performance**

This section describes the strategy for collecting information to help pinpoint web application bottlenecks.

**Steps 1 and 9: Code in the Client Machine**

Placing timers in the client machine is a simple task. A timer can be added at the beginning and end of the client-side code. This will indicate precisely how much time is spent in the client. Even though this is an unlikely source of performance issues, it is so easy to measure that this should be your first step, if only to rule out the client machine as a source of the performance issue.

**Steps 2 and 8: Transmission Between the Client and Application Server**

Transmissions to and from the application server are difficult to measure directly. If you can ensure that the clocks on the client machine and application server are exactly synchronized, it is possible to put time stamps on transmissions, but precise synchronization is often very difficult. Within an organization, Network Time Protocol (NTP) can be used to synchronize machines. However, when portions of the system are managed by different organizations, such synchronization is nearly impossible. A better solution is to determine the sum of time required for these two transmissions by measuring the total time from transmission to reception at the client machine and subtracting the amount of time spent from when the application server received the request and sent it back.

This information will not reveal whether the problem is occurring during Step 2 or Step 8, but it will detect whether or not the problem is Internet related. If the problem is related to slow Internet transmission, the cause is likely to be large data volume. This can be tested by measuring the round-trip time required to send and retrieve varying amounts of information.

Problems in Step 2 and Step 8 can also be caused by latency issues. Sometimes, firewalls or other physical parts of the network can cause sudden bottlenecks. Large (and often widely varying) delays are usually an indication of network configuration issues.

**Steps 3–7: Round-Trip from the Application Server to the Database and Back Again**

The time spent going from the application server to the database and back is easy to measure by calculating the difference between the time stamps at the beginning and
end of a routine. Depending on the system architecture, breaking down the time spent between Steps 1–3 can be challenging.

The total time moving to/from the database can be very difficult to measure. Most Java applications directly interface with the database in multiple ways and in many places in the code. There may be no isolated servlet through which all database interaction passes. In the database itself, if the application server sends many requests from different sessions, the database cannot determine which information is being requested by which logical session, making it very difficult to get accurate time measures.

If the system architecture includes Java code that makes random JDBC calls, there is no way to identify where a performance bottleneck is occurring between Steps 3–7. Time stamps would be needed around each database call to provide accurate information about performance during this part of the process.

A more disciplined approach for calling the database is needed. This can be handled in either the application server or the database. In the application server, you could create a single servlet through which all database access would pass to provide a single location for placing time stamps. This servlet would gather information about the time spent in the application server alone (Steps 3 and 7), as well as the sum of Steps 4, 5, and 6 (to/from and in the database). Since the time spent in the database (Steps 4 and 6) will be negligible, this is an adequate solution.

To measure the time spent in the database, create a single function through which all database access is routed. The session ID would be passed as a parameter to the function to measure the time spent in the database and the number of independent calls.

**Measuring Performance**

Simply understanding a nine-step tuning process is not enough to be able to make a system work efficiently. You need a formal, quantitative way to measure performance. You also need some specific vocabulary to avoid any possible misunderstanding. The vocabulary list may vary somewhat, but the following terms are fundamental:

- **Command** An atomic part of the process (any command on any tier).
- **Step** A complete processing cycle in one direction (always one-way) that can be either a communication step between one tier and another or a set of steps within the same tier. A step consists of one or more commands.
- **Request** An action consisting of a number of steps. A request is passed between different processing tiers.
- **Round-trip** A complete cycle from the moment the request leaves the tier to the point when it returns with some response information.
Chapter 1: The Role of PL/SQL in Contemporary Development

Under the best of circumstances (when you can acquire complete information about every step), the concept of a round-trip is redundant, but in the real world, getting precise measurements for all nine steps is extremely complicated, because in reality, there are two completely different kinds of steps:

- Steps 1, 3, 5, 7, 9—Both the start and end of the step are within the same tier and the same programming environment.
- Steps 2, 4, 6, 8—The start and end are in different tiers.

Having entry points in different tiers means that if time synchronization does not exist between tiers, making time measurements is useless. This problem can be partially solved in closed networks (such as military or government-specific ones), but for the majority of Internet-based applications, a lack of time synchronization is a roadblock because there is no way to get reliable numbers.

The concept of a “round-trip” enables us to get around this issue. The nine-step model shown in Figure 1-1 could also be represented as shown in Figure 1-2.

**FIGURE 1-2.** Round-trip timing of nine-step process
As you can see, a full request-response cycle can be also represented as five nested round-trips, one within the other. Here are their descriptions from the innermost one to the outermost one:

1. From the moment that a request is accepted in the database to the moment when a response is sent back from the database (start of the PL/SQL block to end of the PL/SQL block)—40 seconds in Figure 1-2.

2. From the moment that a request was sent to the database to the moment that a response was received from the database (start of JDBC call to end of JDBC call)—50 seconds (40 + 4 + 6).

3. From the moment that a request was accepted to the moment that a response was sent back to the client machine (start of processing in the servlet to end of processing in the servlet)—75 seconds (50 + 10 + 15).

4. From the moment that a request was sent to the application server to the moment when a response was received from the application server (start of servlet call to end of servlet call)—80 seconds (75 + 2 + 3).

5. From the moment that a request was initiated (user clicked the button) to the end of processing (a response is displayed)—100 seconds (80 + 15 + 5).

Now there is a “nested” set of numbers that is completely valid because all numbers are measured on the same level. This allows calculation of the following:

- Total time spent between the client machine and the application server both ways (Step 2 + Step 8) = round-trip 4 (80 seconds) minus round-trip 3 (75 seconds) = 5 seconds.
- Total time spent between the application server and the database both ways (Step 4 + Step 6) = round-trip 2 (50 seconds) minus round-trip 1 (40 seconds) = 10 seconds.

Although there is no way to reduce this to a single step, it is significantly better than no data at all, because two-way timing provides a fairly reliable understanding of what percentage of the total request time is lost during these network operations. These measurements provide enough information to make an appropriate decision about where to utilize more tuning resources, which is the most critical decision in the whole tuning process.
Solving Web Application Performance Problems
Solving the performance problems in each of the nine web application process steps requires different approaches, depending upon the location of the problem.

Solving Client Machine Performance Problems (Steps 1 and 9)
Performance degradations in the client machine are usually caused by page bloat burdening the client with rich UI components that could be eliminated. Determine whether all functionality is needed in the client machine. Can some processing be moved to the application server, moved to the database, or eliminated entirely?

Resolving Performance Issues Between the Client Machine and Application Server (Step 2)
If the performance slowdown occurs during the transmission of information from the client machine to the application server, you need to decide whether any unnecessary information is being sent. To improve performance, either decrease the amount of information being transmitted or divide that information into two or more smaller requests. This will reduce the perceived performance degradation. Making web pages smaller or creating a larger number of smaller web pages is also a possible solution. If the issue is delays or bottlenecks within the network, you need to identify the component that is causing the delay.

Solving Performance Problems in the Application Server (Steps 3 and 7)
If the application server is identified as a bottleneck, examine the code carefully and/or move some logic to the database. If too many round-trips are being made between the application server and the database, are Getters/Setters being overused? Is one record being retrieved with a single query when a set can be retrieved? If performance cannot be improved because the program logic requires bundles (or thousands of round-trips), rewrite the Java code in PL/SQL and move more code into the database.

Solving Performance Problems in the Client Machine (Step 9)
If too much information is being moved to the client machine, the only solution is to reduce the amount of information. Changing architectures, making web pages...
smaller, and removing or reducing the number of images may help. Analyze each web page to determine whether it is too large and, if it is, reduce its memory size or divide it into multiple smaller pages.

**Lessons Learned**

There is much more to tuning a web application than simply identifying slow database queries. Changing database and operating system parameters will only go so far. The most common causes of slow performance are as follows:

- **Excessive round-trips from the application server to the database** Ideally, each UI operation should require exactly one round-trip to the database. Sometimes, the framework will require additional round-trips to retrieve and make session data persistent. Any UI operation requiring more than a few round-trips should be carefully investigated.

- **Large pages sent to the client** Developers often assume that all of the system users have high-speed Internet connections. Everyone has encountered slow-opening web pages that take multiple seconds to load. Occasionally, these delays are acceptable in situations with graphic-intensive web browsing. However, this type of performance degradation (for example, waiting 3 seconds for each page refresh) in a production application (such as a data entry-intensive payroll application) is unacceptable. From the initial design phase, the application architecture should take into account the slowest possible network that it will need to support.

- **Performing operations in the application server that should be done in the database** For large, advanced systems with sufficient data volumes, complete database independence is very difficult to achieve. The more complex and data intensive a routine is, the greater the likelihood that it will perform much better in the database than in the application server. For example, the authors encountered a middle-tier Java routine that required 20 minutes to run. This same routine ran in 2/10 of a second when refactored in PL/SQL and moved to the database. In some organizations, this may be the primary reason why web applications perform slowly. This situation often occurs when Java programmers are also expected to write a lot of SQL code. In most cases, the performance degradation is not caused by a single slow-running routine, but by a tendency to fire off more queries than are needed.
Chapter 1: The Role of PL/SQL in Contemporary Development

Summary

Performance tuning must be approached holistically by looking at the system in its entirety. It is unlikely that performance issues are caused by PL/SQL code problems, so before you do any PL/SQL tuning, you need to eliminate other possibilities. If a particular user interface operation or batch routine is doing tens of thousands of context switches between Java code and the database, no amount of PL/SQL tuning will fix the problem. Always start by looking for the more obvious sources of performance degradation. The nine steps described in this chapter should be used to determine with certainty that the problem lies in the database. If the database is indeed the source of the performance issues, you must look at the length of time that queries take to execute in order to rule out any SQL problems. Finally, if the fault is truly in the PL/SQL, you need to instrument your code by adding timing mechanisms to pinpoint where the problem exists. Keeping all nine of the potential areas for encountering performance problems in mind and investigating each one carefully can help to identify the cause of a slowly performing web application and indicate ways in which that performance can be improved.

The important points to keep in mind regarding the role of PL/SQL in the current development environment as discussed in this chapter are as follows:

■ In contemporary IT systems, database processing is only one out of nine steps in the logical round-trip initiated by an end user. Performance problems may occur in any of the nine steps.

■ Before starting any optimization efforts, you must determine where the most time is being consumed.

■ A large number of round-trips between the application server and the database is often overlooked as a source of performance problems.

■ Large page sizes are usually no longer problematic, but may still be if developers load a page with images or bring back hundreds of rows of data without using pagination. Large pages will always be a problem where high-speed Internet is not available.

■ Appropriate usage of PL/SQL is the best way to improve performance of a slow-running web application. Data-intensive logic is usually faster when written in PL/SQL and stored in the database than when implemented in Java and executed in the application server.