

7

The Oracle Pathologist

Now that we have identified what the problem is, and have quantified it, we are ready to find the root cause. In order to accomplish this, the tuning specialist takes off the hat of the detective, and dons the hat of the pathologist. As the Oracle pathologist, the tuning specialist will endeavor to find the disease that is the primary cause of the performance problem.

To recap, this is where we are in the Physician-to-Magician approach:

Physician	Detective	Pathologist	Artist	Magician
Define Problem	Investigate	Isolate cause	Create solution	Implement

We are at a critical stage in the performance tuning process, but much good work has already taken place. Results from the previous phases will help the tuning analyst concentrate his attention on the key problem areas.

This phase of the Physician-to-Magician process is somewhat different from the previous stage, in that it is very close to pure analysis – the people side of things has more-or-less been taken care of. In other words, this part of this process is much closer to 100% science. There will still be room for creativity and ingenuity, but this stage emphasizes analytical skills rather than artistry and people skills.

The Oracle Pathologist will strive to find the root cause of the performance problem. The subject of root cause analysis can be very broad; thus, we have devoted four entire chapters to this important topic:

- ❑ Chapter 7: *The Oracle Pathologist*
- ❑ Chapter 8: *Analyzing SQL Bottlenecks*
- ❑ Chapter 9: *Analyzing SQL Joins*
- ❑ Chapter 10: *The Pathologist's Tool Kit*

In this lead off chapter, we first provide an overview of the main tasks of the Oracle Pathologist. We also look at exactly what it means to **isolate a root cause**. In other words, what exactly is the performance analyst trying to accomplish? Understanding the nature of a root cause helps us to understand the true objective of the database pathologist. This in turn leads us to a logical procedure for isolating root causes.

There are several techniques that the analyst can use to pinpoint the root cause. In this chapter, we discuss how to use the following methods:

- ❑ The **simplify** technique
- ❑ The **divide and conquer** technique
- ❑ The **timeline** technique

At the end of the chapter, we also briefly give a few other suggestions for finding the root cause

To help the reader get a general feel for the topic, we also provide a survey of the most common root causes. This will acquaint us with the various possibilities that often turn out to be the culprit.

The Oracle Pathologist has a big job waiting! Let's grab the evidence bag, scrub-up, and get our microscope ready...

Isolating the Root Cause?

The phrase "isolate the root cause" is often used to describe troubleshooting – whether in an Oracle database, user application, or some completely unrelated field. These exact words, or very similar words, are used when describing the effort to resolve a wide variety of problems.

The notion of isolating the root cause appears to fit just about any type of technical problem. We follow the same sort of reasoning, and ask the same type of questions, whether the problem is small or gigantic. The same general approach is suitable for dealing with a car problem or with a national security problem!

The above observation is very convenient for performance troubleshooters, since we can follow the same principles, regardless of the details surrounding the problem. What, then, are these principles and how do we isolate the root cause?

Some Useful Steps to Follow

We are going to begin this section with a hint that will best summarize what we have to do to isolate the root cause of a problem:

Isolating the root cause requires the performance analyst to separate the interactions, identify the source, and confirm the effect.

Let's take a more detailed look at each of the phrases of the above hint.

Separate the Subsystem Interactions

First, then, the analyst must clearly *distinguish* between many possible interactions. This means, for instance, that the tuning specialist should clearly identify and quantify the performance degradation that is due to network issues, as opposed to database-specific problems. It does no one any good to announce that the root cause of a serious performance problem is a slowdown due to network bottleneck combined with poorly optimized SQL statements. Further, statements such as this suggest that the performance analyst has not really completed the job.

The root cause that is ultimately identified will normally not be mixed with others. Of course, there are cases where several flaws contribute to a problem, but the Oracle analyst should be vigilant not to just "lump together" several causes. On rare occasions, a DBA will come across some system that is so badly designed that there are layer upon layer of problems. Fortunately, however, this is not the norm.

Identify the Source

Secondly, along with separating the various potential causes, the analyst needs to clearly identify the cause, or source, of the problem. The previous steps in our approach have provided us with many facts and possible suspects; now the Oracle Pathologist needs to point the finger at the guilty party. Of course, once the various ingredients in the performance puzzle can be distinguished, the job of identifying should be much easier.

As a practical matter, it is helpful for the Oracle expert to clearly announce what that root cause turned out to be. After all, the client usually is curious to know what caused the performance problem. Managers in the firm will naturally want learn how to prevent a recurrence, and technicians will want to know if they had guessed correctly. You owe it to your client to give them a clear summary of the issue.

Confirm the Effect

Thirdly, the Oracle analyst must confirm that the cause that has already been separated and identified is really the prime cause that has led to the performance problem. What good is it to separate and identify something that turns out *not* to be the prime cause of the problem?

Just like the witness who identifies the wrong suspect, presenting the wrong root cause is disastrous. Management will probably remember that a significant amount of time and money has been expended for no reason! This error can be avoided by remembering to confirm your theory as to what is causing the performance bottleneck.

We can sum up the three components of a root cause in our next hint:

The root cause is the identifiable source that has led to observable performance degradation.

If we grant the notion that the Oracle Pathologist is tasked with finding the root cause, then we have already found a logical way for him or her to proceed. These three ideas – separate the interactions, identify the source, and confirm the effect, suggest a reasonable approach to performing the job of Oracle Pathologist.

Now that we have established a theory for how the Oracle Pathologist should operate, let's look at how this works in practice.

Identify Performance Degradation Sources

The Oracle Pathologist must identify the performance degradation that is caused by each subsystem. Performance problems do not confine themselves to just one subsystem. For instance, there will generally be at least *some* sort of application that accesses the Oracle database over some sort of network. Clearly, Oracle databases cannot achieve anything by themselves; useful programs must be designed to retrieve and format the data in a manner helpful to the users.

Already, then, we have identified three subsystems that will possibly be involved in every performance problem: database, application, and network. In addition, there are often other subsystems involved, such as web servers, application servers, report servers, remote databases and data feeds, and so on. Theoretically, even the client setup can impact performance.

Amazingly, many analysts spend days troubleshooting a performance problem without even bothering to identify which subsystem could conceivably be responsible. Many hours are wasted chasing performance issues on systems that are completely uninvolved! (The old guessing approach strikes again!)

Identify the Real Contributors

Since there will always be multiple subsystems that could theoretically be the cause of a performance problem, it will be necessary to determine which subsystems are contributing in reality to the performance problem that is being analyzed:

Identify the approximate performance degradation due to each subsystem.

Of course, each subsystem will probably add some delay, which is normal. Even a perfectly operating network with massive bandwidth will induce a fraction of a second delay. Usually, such small delays can be safely ignored, but if the performance problem really did result from a sub-second issue, even these miniscule delays would have to be considered.

Putting aside these trivial performance delays, the analyst should look for large, unexpected delays in each candidate subsystem that are roughly the same size as the performance problem. Thus, if the users are complaining of a 10-minute delay, don't waste your time trying to tune SQL code that contributes a total of 10 seconds delay. The 10-second delay might become important later on, but is probably not relevant now:

Look for subsystem delays that are of the same order of magnitude as the overall performance problem.

When small performance problems, such as the above 10-second delay, are found, simply make note of them for future consideration. Although they do not warrant investigation right now, they probably will later. After the hot performance problem is resolved, the focus can shift to correcting even these smaller issues.

Performance Budgets

In aerospace companies that are constructing massive, complicated electronic systems, each subsystem within the machine is often assigned an error budget. This budget represents the subsystem's permissible contribution to the total error of the system as a whole. This is a common way of making sure that the total system error remains within the required specification.

For a radar tracking system, for example, suppose that the end customer has specified that the final tracking error must be no larger than 0.10 degrees. The chief designer might assign the following error budget:

Measurable Entity	Degrees Error, Maximum
Antenna	.004
Calculation error	.003
Optics	.015

The same principle applies to performance tuning; the total response time comprises various contributions from each of the components. Some of the contributions will be negligible, but each subsystem will contribute something to the total delay.

Normally, each subsystem causes slight degradation that is within our expectations. Performance bottlenecks occur when one or more of these subsystems begin to operate outside their error budget. So, when the total error (in our case, the elapsed *run-time*) is not acceptable, then a good first step is to identify the individual error contributors.

Let's look at an example:

Example: Slow Form

Consider a performance problem in an Oracle Forms application. A certain screen seems to delay about 45 seconds when a particular function or form is requested. The user reports that the screen eventually displays correctly, but the delay seems odd. Why should there be such a long delay for a relatively simple form?

The Oracle performance analyst, having read the chapter on using `SQL_TRACE` (Chapter 5), turns on SQL tracing and uncovers the SQL statement that was run for this function. (Alternatively, the application itself may have kept run logs of the SQL statements.)

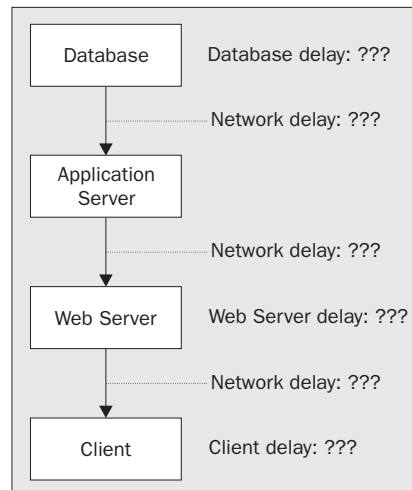
We observe that the SQL code for this part of the application simply finds the total revenue for each contract that began in the past year:

```
SELECT Contract_Name, C.Contract_Number, SUM (Revenue)
FROM Contract C, Acts_Received A
WHERE C.Contract_Number = A.Contract_Number
AND C.Start_Date > (Sysdate - 365)
GROUP BY Contract_Name, Contract_Number;
```

The subsystems involved in this application include not only the database, but many others as well. There are at least seven subsystems involved, namely:

- database
- database to application server
- application server to web server
- web server to client
- web server
- application server
- client web browser

The complete database-network-application system is shown in the following diagram:



At this point, the analyst cannot (and should not) form any conclusions concerning which subsystem is the source of the problem; the performance bottleneck could originate at *any* of these subsystems. Perhaps the database is responsible for the 45-second delay; on the other hand, any of the network traffic could cause a large delay. Another possibility is that the client's workstation is the real bottleneck.

The Oracle analyst can easily assign a number to the delay directly attributable to the database. In our example, assume that the DBA tested the previous query in SQL*Plus. He has modified the original SQL so that the results are placed in another Oracle table, rather than displayed on the screen. This is important so that the elapsed time is not simply a measure of how fast the CRT can display the rows.

To check the database performance for this query, the DBA started SQL*Plus, set TIMING ON, and then ran the following transaction:

```
SQL> CREATE TABLE X UNRECOVERABLE AS
  2  SELECT Contract_Name, Contract_Number, SUM (Revenue)
  3  FROM Contract C, Acts_Received A
  4  WHERE C.Contract_Number = A.Contract_Number
  5  AND C.Start_Date > (Sysdate - 365)
  6  GROUP BY Contract_Name, Contract_Number;
```

Note the keyword UNRECOVERABLE. This was added so that there would be no logging for this transaction. That is, there will be no delay due to writes to the redo logs. Although this was not absolutely necessary for this test, logging might slow down the transaction slightly, and give a run-time a bit too high. In addition, remember that the SQL in the actual application is a query, not a transaction; thus, there is no logging in the real application.

The DBA completed his test; the elapsed run-time for this query was 1.80 seconds. (Second and subsequent queries were faster, since much of the data was cached.) This value can now be assigned to the database delay shown in the previous diagram. It is now apparent that the database alone cannot possibly be responsible for a delay of 45 seconds; one of the other subsystems must be the culprit.

While running the SQL test, the DBA discovered an interesting fact; the sample SQL caused about 10,000 contract values to be inserted into table X. As the firm only has a few dozen customers, this result is strange. How can the query return such a large result set? Since the test SQL uses the same data as the actual application, this means that this part of the application requires 10,000 rows to be transferred over several network hops: from the database to the application server, to the web server, then finally to the client.

Estimating that each row is about 50 bytes (roughly 10 bytes each for contract number and dollar value, plus 30 bytes for the contract name); this means that about 500 KB would be transferred several times, as the data hops between the various servers. Having discovered this fact, a delay of 45 seconds begins to look very fast!

Further investigation revealed that the `Contracts` table contained nearly 10,000 extra contracts. These bogus entries were actually just test data that had been inadvertently left in the `Contract` table. (Note that the books for this firm are arranged such that the number of contracts is always relatively small. Should there *really* be 10,000 contracts, the issue would have to be revisited.) After removal of the extra entries, the application was re-run. The run-time for the form in question was now less than three seconds.

Let's now look at the first of the promised methods for isolating a root cause. The first suggestion is to **simplify**.

Simplify

Besides identifying the delays due to the various subsystems, it is often helpful to simplify. This approach is especially useful when applied to SQL tuning, but the concept can be applied to many technical issues. When applied to SQL code, the idea is to remove irrelevant distractions from the query, such as formatting, irrelevant functions, simple lookup tables, and so on.

Often, what begins as a very formidable problem looks less intimidating once these distractions are removed. This simple step helps the analyst focus on the real core of the performance problem.

Simplifying SQL Queries

SQL queries will often contain a huge number of fields in the `SELECT` list. To further complicate the issue, a variety of different functions may also be used – such as `TRIM`, `UPPER`, `TO_CHAR`, and so forth.

The large number of fields and functions often makes the task look harder than it really is. If nothing else, a huge `SELECT` list makes it difficult to see the entire SQL statement on one page, forcing the analyst to shift his attention from page to page. It is amazing how much easier a problem seems when the essential problem can be cleanly listed on a single page:

Reduce a complicated SQL query to the essence of what the query is doing.

The object of simplifying is to reduce the SQL code down to the essence of the query (or transaction). This is really half the battle! When the core code can be clearly identified, the analyst has already gone a long way towards finding the root cause. Of course, this simplification step must be done carefully, so as not to remove anything of substance inadvertently.

Once a satisfactory correction has been formulated, based on the simplified code, the analyst can confirm the solution by repeating the test with the original SQL.

Remove Irrelevant Information

A good first step in simplifying is to note which parts of a complicated SQL query have little to do with the performance of the entire SQL statement. For example, certain columns in the `SELECT` list may be duplicated in other parts of the `SELECT` list.

Consider the simple query below. In this case, it looks as though the application required that the column `Last_Name` be listed in two places:

```
SELECT Last_Name LASTNAME, Last_Name || First_Name FULLNAME
FROM Emp
WHERE Emp_No > 1000;
```

Let's look at the work that the Oracle engine has to perform to run this query. Does it retrieve each row twice, since the SQL query specifies the `Last_Name` column twice? No, it does not! The optimizer simply uses the value that was retrieved the first time. In other words, for purposes of analyzing the performance, the duplicate columns do not materially affect the optimizer.

This observation is good news for the performance analyst, since many SQL queries contain duplicate fields. The simplification step is clear; for purposes of performance tuning, the original query can be rewritten as:

```
SELECT Last_Name, First_Name
FROM Emp
WHERE Emp_No > 1000;
```

Note that the column titles, `LASTNAME` and `FULLNAME` have also been removed. The application needed these column titles, but we do not. This is another simple step to make things a bit easier (and shorter). The optimizer does not alter its operation just because the output is going to be given a special name. Why not get rid of these distractions?

Irrelevant Functions

Functions that are placed in the `SELECT` clause are also an unwelcome distraction. Consider the following SQL:

```
SQL> SELECT DECODE (Acctg_Number, 1, 3), Acctg_Dept || '-001'
2 FROM Acctg
3 WHERE User_id > 100;
```

In this example, the `decode` function and the concatenate operator are distractions that complicate our work. The `decode` function simply substitutes one value for another, and the concatenate operator `||` just appends a few characters. Clearly, neither has any relevance to the execution plan for the SQL statement. The Oracle optimizer will normally not be affected by functions in the `SELECT` clause.

A possible exception would be if the DBA has created one or more indexes on a function. Then, the presence of that exact function in the `SELECT` clause could modify the execution plan, since the optimizer could use the special index to retrieve the information more rapidly.

The revised SQL code is:

```
SQL> SELECT Acctg_Number, Acctg_Dept
2 FROM Acctg
3 WHERE User_id > 100;
```

An observer might wonder, "Don't these functions cause the CPU to perform more work?" Yes, they do. The database engine must calculate each function for each row retrieved. If many thousands or millions of rows were being retrieved, this fact would certainly have to be considered. Practically speaking, however, these minor delays will normally be far less than the real performance bottleneck. As always, the performance analyst is looking for contributed delays that are in the ballpark of the total system delay.

One caution, however; some applications may make use of user-created PL/SQL functions. These functions cannot simply be removed from the `SELECT` list. In fact, the function could be the root cause of the problem! For instance, an application might employ a function that actually performs many full table scans, with the associated logical and physical reads. By removing this custom function from the `SELECT` list, the entire performance issue is changed.

Apart from this, you may be saying to yourself that the presence of one or two functions in the `SELECT` clause does not complicate matters too much; why bother removing them? For one thing, the ability to find the essential SQL is a useful technique to encourage. Get in the habit of zooming in on the core issue while discarding the noise.

The habit of simplifying SQL will prove most valuable when very large SQL queries need to be analyzed. What seemed like a trivial change in a short SQL statement could become a substantial simplification when faced with a SQL statement that joins a dozen tables.

Simplify, but not too Far!

The trick in the simplification approach is to make the original SQL simpler – without changing the essence of the code. That is, the revised SQL should require the Oracle engine to accomplish the same work as the original SQL. Thus, if the original SQL requires the optimizer to join certain tables, then obviously, the simplified SQL must do the same. Otherwise, the subsequent analysis will not be helpful, and the proposed solution will fail when tried with the original SQL.

Redundant Columns

Sometimes SQL simplification requires that certain columns be temporarily removed from the `SELECT` list. These are cases where a large number of columns in the `SELECT` clause have no bearing on the execution plan – they just clutter up the analysis. These columns optionally may be removed from the `SELECT` clause as long as the table access method does not change.

For example, if the original execution plan required an index lookup followed by a table access, then the revised SQL should use the same method. Consider the following demonstration; the SQL code shown is based on an actual production problem. The original SQL was as follows:

```
SELECT Account_Name, Account_id, Address,
       Custom_1, Custom_2, Custom_3, Custom_4, Custom_5,
       Custom_6, Custom_7, Custom_8, Custom_9, Custom_10,
       Custom_11, Custom_12, Custom_13, Custom_14, Custom_15,
       Custom_16, Custom_17, Custom_18, Custom_19, Custom_20
FROM Account_Security
WHERE Account_id = 894123;
```

After simplification, it looked like this:

```
SELECT Account_Name, Account_id, Address
FROM Account_Security
WHERE Account_id = 894123;
```

This rewrite at first seems completely incorrect! How can all those columns be eliminated and have no effect on the query? Surely the new SQL will be much easier for the optimizer to execute.

In actuality, the two SQL statements will run in very much the same fashion. This is true because the Oracle access method (and therefore the execution plan) is identical in both cases. The presence of the extra columns has a negligible effect on the actual work that the Oracle engine has to perform.

Let's pause for a moment to discuss a commonly used term. When we discuss the way in which the optimizer executes SQL, we use the term execution plan. Just as it sounds, an execution plan describes how Oracle tackles a particular SQL statement. It addresses questions such as: which table is read? Which indexes are used? If there is a join, how are the tables joined? Are there any parallel operations?

In practice, many DBAs refer to this information as the explain plan; but technically, that really means the step we take to get the execution plan, not the plan itself. We provide extensive coverage of generating and interpreting execution plans in Chapter 8.

Getting back to our example; in the original version, none of the `Custom` columns are accessible via index, so that a table access to get those fields will always be required. In the second version, a table access is still required, because the `Address` column is likewise not available via an index. In both cases, a table access will be required.

The execution plan for both SQL statements is the same:

ID	PARENT	OPERATION	OBJECT_NAME
0		SELECT STATEMENT	
1	0	TABLE ACCESS BY INDEX ROWID	ACCOUNT_SECURITY
2	1	INDEX UNIQUE SCAN	ACCT_ID_IDX

We see that an index (based on `Account_Id`) is used first. This is indicated by the operation `INDEX UNIQUE SCAN`. The index tells the optimizer which row number in the `ACCOUNT_SECURITY` table has an entry for the specified `Account_Id`.

After this index lookup, the optimizer proceeds to perform the operation `TABLE ACCESS` of the `ACCOUNT_SECURITY` table. This access retrieves the columns that are needed; in other words, those listed in the `SELECT` clause.

The database will have to perform the same work, using the same execution plan for both SQL statements.

Large Transactions

When a multiple-transaction set is the problem under analysis, one easy way to simplify the performance problem is to break one large transaction into smaller mini-transactions. What remains will be just a few (frequently only *one*) SQL statements. In this way, the analyst can eliminate the portions that execute quickly, and not waste time on irrelevant code.

This principle is easily stated, but how is the performance analyst to accomplish it? How is one to separate the good SQL from the bad? One easy way to distinguish among the various SQL statements in a transaction is to trace the application while it is running. Recall that `SQL_TRACE` can be activated for the entire database if necessary, by setting the parameter, `SQL_TRACE = True`, or for one particular session by calling the stored procedure, `Sys.dbms_system.set_sql_trace_in_session`.

Then, by using TKPROF to format the trace file, it will be clear which SQL statements are really holding things up. This step should be pretty easy, because the elapsed times for each SQL statement are clearly listed in each TKPROF report.

Occasionally, the application itself might also have run logs that define the time spent on various transactions. This will also aid the analyst in his effort to home-in on the actual SQL code responsible for the bottleneck. With that said, let's now move on to look at the second method for isolating the root cause of a problem.

Divide and Conquer

Closely related to simplifying is breaking the problem into pieces, or the divide and conquer method. Actually, this approach is really just another form of simplifying. The principle is the same – we want to try to remove distractions and irrelevant information, so that what remains is the *essence* of the problem

Identify and remove irrelevant elements that don't contribute to the problem.

The division approach works very well in SQL tuning because most SQL statement difficulties are due to a very few causes. This observation also explains why a true expert in SQL tuning is usually so successful; in many cases, all that needs to be done is to resolve a small number of problems – perhaps just one!

Of course, having just one core issue to solve cannot be guaranteed in every case, but this observation should be encouraging news to Oracle professionals – especially those interested in SQL tuning:

Many performance problems can be improved substantially by making just one or two corrections.

Breaking up SQL Code

The divide and conquer approach is especially useful when tackling complicated SQL code. Often, what begins as a very complex SQL problem can be separated into several parts, each of which is much more manageable. In some cases, the separation of SQL code into parts is very obvious; in other cases, more work will be required.

To illustrate this approach, consider the following SQL statement. This code retrieves a list of employees from three separate tables, meeting three separate conditions:

```
SQL> SELECT Emp_name
  2  FROM Emp
  3  WHERE Zip_Code LIKE '92%'
  4  UNION
  5  SELECT Emp_name FROM Term_Employees
  6  WHERE Term_Date < Sysdate - 365
  7  UNION
  8  SELECT Emp_name FROM Managers
  9  WHERE dept_id = 1000;
```

If this SQL statement were to have a performance problem, the separation tactic would be very obvious – simply divide the SQL before each UNION statement. Then, attention could be individually directed at each of the three SQL statements; each statement could be checked for good run-time. Of course, most SQL statements will not be so trivial; nevertheless, the concept will be the same.

Consider this slightly more complicated example:

```
SQL> SELECT Emp_name
  2  FROM Emp
  3  WHERE Dept_id IN
  4  (SELECT Dept_id FROM Dept
  5   WHERE Dept_Name LIKE 'ENGINEER%');
```

This SQL statement can easily be divided into an outer query and an inner query. The inner query runs first, and then it feeds the rows into the outer query. Naturally, both pieces must run efficiently if the entire statement is to run well.

Firstly, then, we must determine which of the two parts is the problem. Is it the outer query, or the inner query? Let's separate them, and find out.

First, list the inner query:

```
SELECT Dept_id
FROM Dept
WHERE Dept_Name LIKE 'ENGINEER%';
```

Then, the outer query could be modeled as:

```
SELECT Emp_name
FROM Emp
WHERE Dept_id IN 'N';
```

Now that the SQL is broken into its parts, simple timing tests will easily show which piece is the culprit. To check the timing of the outer query, one approach would be simply to substitute a typical value for N and check the performance for this one small SQL statement in SQL*Plus.

Simplification of Table Joins

A more sophisticated use of the divide and conquer approach can be applied to table joins. Simplification of table joins is important, because of the frequency of this form of SQL. It is very common to have SQL statements with a variety of table joins, in which only one or two of the tables joined are causing the performance problem.

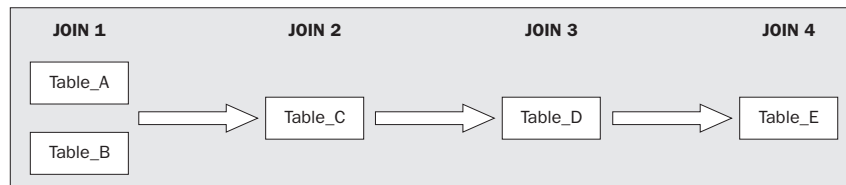
We cover table joins very extensively in Chapter 9, but let's briefly define our terms. A join is a SQL query that uses information from two or more tables. The tables can be joined because they have corresponding columns. For example, one table might contain the fields `Name`, `Address`, and `Phone_Number`; a second table might have the fields `Name` and `Occupation`. These two tables could be part of a SQL join based on the column that they have in common, namely the `Name` field. We call the table that we use first the **driving** table; the second table is called the **driven** table. We will see in Chapter 9 the various ways that the Oracle optimizer can handle these forms of queries.

In complicated SQL queries involving table joins, one useful tactic is to remove a few of the tables in the join in order to simplify it, and thereby distinguish which tables really are performance drivers for the SQL in question. In other words, we try to find out which tables are on the critical path – which ones are really the performance drivers.

Let's look at an example:

Example: One Join at a Time

The following figure illustrates a simple query with five tables involved in joins. In this example, we are not concerned with the exact method of joining:



The five tables used in the join are: `Table_A`, `Table_B`, `Table_C`, `Table_D`, and `Table_E`. Let's assume they are listed in the join order, left to right, in the figure. `Table_A` is the driving table in the first join, and `Table_B` is the driven table. The result from the first join then becomes the driver for the second join, and so on.

The SQL used for this query is:

```

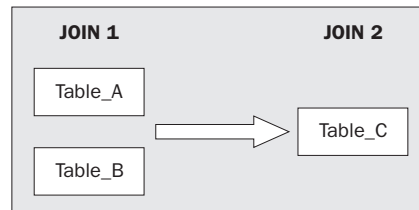
SQL> SELECT A. Emp_id
       2 FROM Table_A A, Table_B B, Table_C C, Table_D D,
       3 Table_E E
       4 WHERE A. Deptno = B.Deptno
       5 AND B.Site = C.Site
       6 AND C.Division = D.Division
       7 AND D.Office = E.Office;
  
```

In this example, note that none of the columns from tables B through E are even listed in the `SELECT` clause. This means that we don't really want to *see* the values from these tables; we only want to use their information in the join conditions.

Assuming that there is a performance problem with this query, we are faced with the question, "which join or which table is causing the bottleneck?"

One good way to answer that question is to systematically divide the SQL statement, thus greatly simplifying the root cause analysis. By splitting the SQL into pieces, the core of the performance problem can be quickly determined.

There are several good ways to start. One idea that immediately comes to mind is to eliminate `Table_D` and `Table_E` from the SQL entirely. This would then leave us with the very simple join shown here:



Notice that we have roughly divided the SQL statement in half. The next step would be to run the simplified SQL that persists in only `Table_A` through `Table_C`, and check performance. The objective would be to see if the problem stays with `Table_A` through `Table_C`, or whether the problem has disappeared.

Note that this simplification process requires that the `SELECT` clause be modified each time a table is removed from consideration. Thus, when `Table_D` and `Table_E` are removed, fields that originate from these tables will need to be eliminated from the `SELECT` clause. This leaves us with this SQL:

```

SQL> SELECT A. Emp_id
2 FROM Table_A A, Table_B B, Table_C C
3 WHERE A. Deptno = B.Deptno
4 AND B.Site = C.Site;
  
```

When simplifying table join code as a method of understanding a more complex expression, it is important that the simplified SQL retains the same join order and join method as the original. Otherwise, we are not comparing apples to apples, and our analysis will be misleading. Thus, in our example, we should verify that the simplified SQL has an execution plan that joins the tables in order `Table_A`, `Table_B`, `Table_C`. Later, once the root cause has been identified and we are ready to change the SQL, we can use whatever join order we deem best.

Suppose that our simplified query with just a few joins still runs poorly. Since there are only three tables (and two joins) involved, the troubleshooting process has been greatly simplified. There are not too many choices left to investigate.

On the other hand, if the simplified query runs well, the next logical step would be to add back `Table_D`, then `Table_E`, and determine at which point the performance problem returns.

The overall goal of the simplification approach is to choose a method that reduces the complexity of the performance tuning process. This may involve the DBA choosing to break a given problem into two, employing a temporary table in the process.

Temporary tables are often used to make a problem more manageable – not because they have to be used. Admittedly, creating a temporary table might incur a slight performance decrease; the tradeoff is the increased maintainability of the program.

Be wary of SQL performance improvements that make it difficult to understand what is happening.

Now let's move onto the final method of determining the root cause of a performance problem.

The Timeline Method

Another useful way to isolate a root cause is to find out what changed. Some performance difficulties occur on systems that had previously been performing fine. All functions had been performing normally, and suddenly the users report that the system is nearly unusable.

In some cases, it will be helpful to create a timeline of events leading up to the performance problem. For example, suppose that the performance of a customer service application suddenly went bad. The timeline might look like this:

Monday	Tuesday	Wednesday	Thursday	Friday
All OK	CPU #1 replacement	Code update	Rebuild indexes	New stats

The performance expert will naturally want to investigate the changes shown on Tuesday through Thursday. On Tuesday, for instance, we note that CPU #1 was replaced. This seems to indicate that there was a major hardware problem. What happened? Was the CPU satisfactorily changed? If so, how did the load on the system change?

The analyst will also want to look into the application and database modifications on Wednesday, Thursday, and Friday. Why were the indexes rebuilt – were there any database problems leading up to that decision? Why were new statistics gathered – and was the same sample size used?

If anything, people are sometimes too suspicious of changes made just prior to the discovery of a performance problem. It is very convenient having an immediate scapegoat to blame. Remember, however, that the exact start time of a performance problem is frequently not precisely known. In fact, many issues reported as new performance problems are actually old problems that were never reported. Believing that the performance problem has just begun, the Oracle analyst could easily jump to wrong conclusions, blaming some harmless change made a day earlier:

Performance problems are frequently reported inaccurately. Gather facts, not conjecture.

Even when a performance problem is accurately reported, it should be remembered that performance problems that occur after a program or database change are not necessarily caused by the change:

Changes made prior to performance problems are not necessarily the cause of the bottleneck.

Sometimes, it really *is* just a coincidence. It is very easy to become so fixated on finding what changed that the analyst mistakenly finds a correlation where none exists.

Common Causes of Performance Problems

For the analyst who is just beginning his or her career in Oracle performance tuning, it will be helpful to list some of the most common root causes of performance issues. Admittedly, this sample is not truly scientific in the statistical sense, but it should nonetheless prove useful in getting a feel for what kinds of things often are the culprits.

In my experience, root causes frequently fall into the following categories:

- Application design and interaction with database
- Database design
- Indexing
- `init.ora` parameters
- Problems from competing batch jobs
- Hardware (including networks)

Application and database design issues are almost always involved in correcting performance problems. At the other end, rarely have I seen problems that are truly caused by inadequate hardware sizing – it happens, but not too often.

Different DBAs will certainly have their own way of grouping root causes, but most DBAs and performance analysts recognize that application design is often at the heart of a performance problem. By application design, we must specifically consider *how the application talks to the database*. This usually leads the analyst to examine the exact construction of the SQL statement that originates at the application:

The number 1 performance issue is how the application interacts with the database.

More than any other cause, the method in which the application retrieves data from the database is the key to resolving many performance issues. In fact, this principle applies even if the database is not Oracle!

In general, relational databases, whether Oracle, SQL Sever, DB2, Sybase, or Informix, follow similar steps in executing queries. When a query is received for processing, the database engine accesses many of the same type of database objects.

For instance, each database flavor uses indexes to speed access to information in tables. Each database must also deal efficiently with table joins (albeit with slightly different versions of join algorithms). Each database must protect the integrity of data through some sort of locking mechanisms. This similarity means that the skills learned in tuning Oracle performance problems will probably transfer to other database brands.

Application Design

The application design is very often a critical piece of the performance puzzle. Good application design permits (but does not guarantee) excellent response time. An especially poor application design may make good performance nearly impossible to achieve.

For instance, an application that encourages users to retrieve a gigantic result set (say, 50 Mb) will certainly run poorly, no matter how well the Oracle database operates. This large data transfer will cause a long delay – no matter what the performance analyst does:

Good database design cannot counteract poor application design.

This principle explains why *every* subsystem must be considered. It only takes one weak link to wreck the overall performance. Since there are hundreds of potential problems with application design, let's have a quick look at some very frequent design flaws that occur time and again.

First, the application logic may be designed to handle results from very small tables only. This is another way of saying that the application will not scale. Usually, this also means that there had been no load/scalability testing.

Unfortunately, it is extremely common for a vendor-supplied application only to have been tested with tables of trivial size. Incredible as it seems, many applications are only tested with tables filled with fewer than 100 rows! The minute that these tables are populated with a realistic set of data, the application grinds to a halt:

Even the poorest of applications may run well with only a few rows in the database tables.

Second, the application uses an algorithm that processes rows one by one, as opposed to an entire set. As in the previous problem, this only becomes an issue when a realistic set of data is used in testing. This problem mainly applies to data warehouse loading algorithms in which it is critical to process many thousands of rows as a group.

Third, the SQL defined by the application might preclude index usage. Developers will often mistakenly put functions in the `WHERE` clause, which often prevents index usage. The following SQL code will block index usage for the column `Cust_Name`:

```
SQL> SELECT Cust_Name
2 FROM Customers C, Accounts A
3 WHERE C.Account_id = A.Account_id
4 AND UPPER(Cust_Name) = 'SMITH';
```

The Oracle engine cannot use an index on `Cust_Name` to address this query, because the query doesn't care about `Cust_Name`; it cares about `UPPER(Cust_Name)`. Recent versions of Oracle, however, allow the DBA to create function-based indexes. Just as it sounds, we create an index on the result of a function, not the raw value itself. In this case, we would create an index on `UPPER(Cust_Name)`.

The real question, however, is why the database is not storing the customer names in a consistent format; that is, why does the SQL query have to reformat the data? These questions are the issue in our next case study:

Case Study: Trouble with Trouble Tickets

One of the most popular help desk applications relies on an Oracle database to store all the trouble ticket information. In one particular company, the group that supported this application complained to the DBA that the nightly update was beginning to run very slowly. Each night, the application had to refresh various database tables with updated customer information.

The root cause of the performance bottleneck was easy to discover; the nightly batch job was being degraded by a single SQL statement. This SQL statement had the `UPPER` function hard coded into the program:

```
SQL> INSERT INTO Cust
2 SELECT Cust_Name, Cust_Id
3 FROM Old_Cust
4 WHERE UPPER (Cust_Name) LIKE 'A%';
```

As shown above, the program first looked for, then inserted all new trouble tickets for the A customers. Next, it would proceed to the B customers, and so on. Since these customer names were not stored in a consistent format, the nightly batch job had to correct all the names dynamically; hence the use of the `UPPER` function, which turns the letters into upper case. This same correction was applied every night.

The DBA documented the design flaw and recommended two steps:
Note to production: this list starts with 1.

Perform a one-time update of the customer names to use a consistent format (in this case, all upper case);

Request that the vendor modify their program so that all subsequent customer entries be saved in upper case.

The vendor in this case agreed to the modification suggested, and provided a patch release of the program. Next, the DBA performed a one-time update to all customer names to convert them to the consistent case.

The nightly batch job was then able to complete in a fraction of the previous time. The modified application stored all names in a consistent format, making any sort of function unnecessary.

Note that the application in the above case study was a major application used throughout the nation. There were probably dozens, if not hundreds of companies experiencing the same problem! Thus, problems with scalability are not limited to just "mom and pop" (small, home-grown) applications. Many well-known applications make similar mistakes, leading to performance degradation as the database grows.

Fourth, the SQL might cause tables to be joined in an inefficient order. Deciding the order of table joins is almost an art in itself. Although including a large number of tables in one SQL statement does make the job a little trickier, performance should not necessarily be bad as more tables are added to the join. We reserve an entire chapter for this important subject: Chapter 9, *Analyzing Table Joins*.

Next, the application may encourage large transfers of data across the network from the server to the client. Transferring many megabytes of data from the database to the user is a sure way to degrade performance. Besides the "hop" over the network to the end user, there are frequently additional delays due to transfers to the application server, report server, web server, and so on. In cases like this, no matter how perfectly the database runs, the best it can do is not make the problem any worse.

Frequently, the end user will not be aware of the large data transfer. For example, the application may not actually display to the user the entire result set at one time. Some reporting applications, for instance, accept the entire result set from the database, then feed it to the client a few pages at a time. The end user may only see one page of data, but behind the scenes, the total amount of work and network transfers can be huge.

Finally, the application could allow empty searches, in which the `WHERE` clause is empty. Many applications do a poor job of restricting user queries to what is reasonable. Very often, the user can mistakenly forget to enter some restrictive criteria, and the application will execute the query anyway. The lack of search criteria means that all data will be retrieved.

Consider the following SQL; it fetches the name and grade for an Oracle "OCP" (Oracle Certified Professional) course for all students who live in Glasgow, who graduated in the past month:

```
SQL> SELECT Student_Name, Grade
  2  FROM Student_Tab S, Grade_Tab G
  3  WHERE S.Student_id = G.Student_id
  4  AND S.Home_Town = 'Glasgow'
  5  AND S.Grad_Date > (Sysdate - 30)
  6  AND G.Course = 'ORACLE OCP';
```

Student_Name	Grade
-----	-----
Herman Smith	A-
Bob Hernandez	B+
Susan Johnson	C+

This query is well written, and the performance should be fine. What would happen if a user ran the same query, but with no restrictions? The following query would return the grade of every student in the college, for every course:

```
SQL> SELECT Student_Name, Grade
  2  FROM Student_Tab S, Grade_Tab G
  3  WHERE S.Student_id = G.Student_id;
```

Student_Name	Grade
-----	-----
Robert Aagor	D
Robert Aagor	A
Robert Aagor	B+
...	**
Herbert Abbot	C-
Herbert Abbot	A-
Herbert Abbot	A
...	**

The results of this query would be completely unusable; the database would have performed unnecessary work, and the network delay would have been substantial. Additionally, other users would possibly have suffered performance degradation due to the unavailability of resources.

Database Design

Besides problems that may occur with the application design, the database design may itself be faulty. The DBA may have forgotten to perform some critical task (such as gathering statistics), or there might have been miscommunication between the design team and the DBAs, leading to wrong or missing indexes.

Here are some common errors on the database side:

- ❑ Unnecessary database feature wrongly activated
- ❑ Old statistics fool the optimizer
- ❑ Skewed data but no histograms built
- ❑ Index with excessive columns
- ❑ Stagnant indexes (need rebuilding)
- ❑ Missing/wrong indexes

In this list, some of the items are arguably really application issues. Index problems could probably be classified as either a database problem or an application problem. The DBA might assist with indexing suggestions, but indexes are really closely related to the application. Indexes exist in order to assist access to application objects, so one could argue that they are really part of application design.

The DBA might have implemented an overly complex design, such as using Oracle Advanced Replication, where a much simpler design could have worked equally well. Or, perhaps the MTS (Multi-Threaded Server) option is being used for no real reason.

init.ora Parameters

Oracle databases are tremendously resilient when it comes to operating well in spite of incorrect initialization parameters. Occasionally, however, the database has been set up so poorly that the database engine struggles to accomplish even simple queries.

Some typical errors with these parameters include:

- ❑ Greatly undersized SGA, for example, `Db_block_buffers` set too low (or so high that the memory on the server is exhausted).
- ❑ Greatly oversized shared pool.
- ❑ Unusual or exotic features used unnecessarily, for example MTS.
- ❑ Wrong setting for `Db_File_Multiblock_Read_Count`.

Nowadays, many new DBAs are unreasonably biased in favor of bigger is better for database caches; thus, undersized database caches are becoming rarer. In their place are oversized shared pools. A huge shared pool is often created when the application uses many unique SQL statements (for example, bind variables are not used). The large pool often makes things worse, because for each SQL statement issued, the Oracle engine has to search through the pool looking for matches. (The solution, of course, is to reduce the massive number of distinct SQL statements, not to create a huge shared pool.)

Setting the parameter `Db_File_Multiblock_Read_Count` is a bit trickier. It is normally set to the maximum number of Oracle blocks that can be read from disk at one time (traditionally, a total of 64 KB – but new servers often have a far greater capacity, sometimes even 1 MB). A low value means that multi-block reads will be unnecessarily restricted; a high value (beyond the actual capability of the server) means that the optimizer incorrectly believes that multi-block reads are more efficient than they really are. This in turn leads the optimizer to incorrectly favor full table scans instead of index lookups.

Interference from Batch Jobs

Sometimes, neither the database nor the application is at fault. Instead, some other process on the server is interfering with efficient database operation by consuming resources. For instance, in one of our previous case studies involving a data warehouse, one batch job interfered with another by locking access to some rows. In other situations, there might be regular maintenance that is mistakenly scheduled at the wrong time.

Our next case study illustrates how other batch jobs can lead to performance problems:

Case Study: Backed-up Backups

A DBA supporting a data warehouse application was trying to determine why the run-time for the nightly data extract was inconsistent. The nightly data extraction began just after midnight, and normally ran until 2:30 A.M. Sometimes, however, the time stretched out about 20 minutes longer.

The logs for the data extraction showed that one step in the data extraction (pull of data from a Sybase system) usually took about one minute. Occasionally, however, the time would grow. The logs showed that the number of rows retrieved from this one table in the Sybase database was nearly identical every night; even when the time was much greater, the number of rows varied by only a few percent.

The DBA started keeping a logbook to try to understand what was happening. All the other steps in the nightly data load ran very consistently – perhaps just a few percent or so variance each night. The steps both prior to and following the odd one ran fine.

The DBA mentioned this phenomenon to one of the Systems Administrators. The Sys Admin explained that the network bandwidth between the various production servers was so great, that the network could not possibly be responsible for the delay of a relatively small data transfer.

Another Systems Administrator, the one responsible for data backups, explained to the DBA that all backups had been shifted outside the midnight to 3 A.M. time window, thus avoiding any conflicts.

The DBA continued to log the runtime of the various data loads. All the facts indicated that there was some process that activated at exactly the same time every night. Finally, the DBA prepared to activate a special monitor to determine what processes were running on the server at that time. At this point, the backup Systems Administrator revealed that one of the backup jobs had been missed, and was indeed running at 1:30 every morning. Depending on the exact runtime of the backup job, there would sometimes be resource competition between the database server and the backup processes. When they both ran at the same time, database performance was severely degraded.

In this case study, it would have been understandable if the performance analyst had focused on either the Sybase data source or the application that pulled the data from Sybase. In this case, however, there was nothing wrong with either the application or the database; the root cause was interference from a batch job that had been incorrectly scheduled.

Hardware

On some occasions, there may actually be a resource problem with a hardware component; for instance, there could be so few disk drives installed, that disk I/O over the few spindles drags down performance. This scenario appears to occur less frequently, now that disk drives are so inexpensive. Also, most firms use striping to distribute files across multiple disks.

Rather than having too few disks, the more common difficulty is that the busy, or "hot", files are lumped together, instead of being more evenly distributed across the entire set of drives.

For applications requiring large transfers of data over a network, the network bandwidth may become an issue. Our next case study illustrates this problem.

Case Study: Local or Long-Distance?

A financial firm was re-evaluating its software development process. They wanted the various designers to be able to model the new application using a design tool, and be able to share the model within the group. The company decided to use the Erwin modeling tool, along with the ModelMart repository. With this setup, each designer checks out the up-to-date model, performs their work, and then checks-in the changes to the repository.

The DBA on the job was familiar with the Erwin product, and quickly had a sample repository for the designers to use. Testing showed that everything was working well; the DBA was able to create a small model, check it out, and so on. There seemed to be no problems. The DBA told the designers to give it a try.

When the designers tried to use the ModelMart repository, they experienced massive delays. In fact, even the act of logging in required about 30 seconds! The supervisor called the DBA and complained about the problem.

The DBA went to the office down the street where the designers worked. He first tried connecting over SQL*Plus to the database. The connection was established without a hitch, not immediately, but it seemed ok nonetheless.

Tests with the application, however, did not go so well. The DBA confirmed that using the application to connect to the database did elicit a long delay, about 15 to 30 seconds, just as the designers said.

The DBA put a trace on the application (activated `SQL_TRACE`) to see what was happening. The trace file showed that after login, the application transferred a moderate amount of data (about 500 rows) to the client. This did not seem a large amount of data, but it appeared to be the only difference between the SQL*Plus login, and the ModelMart login.

Armed with this new information, the DBA began to get suspicious of the network speed between the two buildings. (The designers were in one building; the database server was in another.) He ran a simple ping test to see how fast the network responded. To his astonishment, a simple ping sometimes took one second to accomplish.

While network response times can vary greatly, a reasonable estimate for a ping might be roughly 25 ms; thus, a one-second delay represents nearly 2 orders of magnitude degradation. A delay of 1 second on average is unbelievably bad.

He then tried transferring (via ftp) a few small files. The file transfer test confirmed that the network bandwidth between the two buildings was severely limited. Even the smallest data transfers would take a few seconds.

Discussions with the network group revealed that the network between the two buildings had been set up as a temporary measure. The network technicians were well aware of this bottleneck, but there was nothing they could do in the short run.

Unfortunately, the users who wanted to use the ModelMart application were all located in one building, and the server (and the DBA) in another. There was simply no way that a model could possibly be transferred over this severely limited network. Since a simple login required 15 seconds, a transfer of the entire application model would probably take an entire hour! Obviously, this would make the system completely unusable.

The DBA explained the problem to the project manager. She understood, and reluctantly agreed to postpone the ModelMart project. The entire project had to be delayed for months, until a more suitable network architecture could be implemented.

Of course, there were other potential options, for example, moving people, moving servers; but the real core of the problem was the network. All other solutions would be dodging the fundamental issue that needed to be addressed. Fortunately, the network administrators readily accepted responsibility for correcting these network issues.

Another potential hardware problem is related to CPUs. On some servers, the number of CPUs installed may be too few in number (or too slow). This situation is sometimes due not only to a desire to minimize hardware costs, but also as a way to reduce licensing fees. This phenomenon occurs because the price of some software packages is proportional to the number of CPUs installed. For some sophisticated software packages, this means that the licensing cost per CPU can be very high – even hundreds of thousands of dollars. Naturally, this pricing scheme encourages IT managers to try fewer CPUs, at least at first.

Note that the database server is supposed to be CPU-limited; that is, in a very well-tuned system, the other bottlenecks, such as disk I/O and network limitations, should have already been eliminated. In this ideal case, adding more (or faster) CPUs will naturally improve performance. Of course, the important question is, have the other bottlenecks really been addressed?

Recall our analogy of the clerk in the supermarket, servicing a long list of customers waiting in the checkout line. Once all the supplies and equipment have been provided to the clerk, there is only one thing left to do; employ more (or faster) workers. Prior to hiring more staff, however, the supermarket management will be very careful to make sure that all the clerks are performing at peak efficiency:

Well-tuned database systems are normally CPU-limited.

Although there are undoubtedly some cases where all other bottlenecks have been addressed, and more CPUs are appropriate; this option is not usually the first alternative that the performance specialist should consider. Although some performance gain is normally achieved by adding more CPUs, it is better first to find out why the CPU load is very high. If an underlying problem is never investigated, the addition of more CPUs may buy a brief respite, but the real problem will still exist. For instance, an excessive number of logical reads will tend to drive the CPU load very high. It might be tempting to bolt-on more CPUs immediately, but the correct approach is to tune the SQL that is causing the excessive logical reads.

Other Practical Suggestions for Finding the Root Cause

There are some easy ways to make the job of the Oracle Pathologist a little easier. Consider what the Oracle Pathologist is trying to do; he is trying to reduce the overall elapsed time of SQL processing. Thus, it is important to have an easy way to determine exactly what the runtime is.

Another practical issue is how to handle large result sets. How can the analyst find the runtime of a SQL statement that returns 100,000 rows? How can the display time be separated from the actual run-time?

Simple Timing Tests

When testing SQL statements for elapsed time, it is convenient simply to be able to run the query in SQL*Plus, with timing set to ON. (The command in SQL*Plus is simply `Set Timing ON`). Although not as sophisticated as running `SQL_TRACE` (followed by `TKPROF`), this simple test is actually good enough for reviewing many problems.

Simple test runs using tools such as SQL*Plus are often more than adequate to confirm a problem and get a feel for the magnitude of the bottleneck. Of course, the analyst is still free to choose a more sophisticated approach if necessary; in practice, however, a quick check in SQL*Plus is often a good first step:

Use simple methods, such as SQL*Plus timing, to obtain approximate run-times.

For instance, a customer with a one-minute performance bottleneck really doesn't care whether the performance problem is technically 59.3 seconds, 65.2 seconds, or any other similar number. All the performance analyst needs to know is that the delay is about one minute. Admittedly, using SQL*Plus is not exactly the same as running the same SQL from the application – but it is usually good enough to allow the Oracle expert to diagnose the problem.

Large Result Sets

In some cases, a query returns so many rows that the time to display the rows becomes the driving factor for the total execution time. This makes it difficult to distinguish between actual database-related time and the time to display the result set. When running the query in SQL*Plus, the long display time will probably mask the underlying problem, or exaggerate the actual run-time.

There are several ways around this difficulty. One simple way is to reword the SQL code to insert the results into a temporary table, rather than actually displaying the results. For example, consider a large query that returns 10000 rows. (Assume for this example that indexes are not being used). Note how the SQL could be rewritten to facilitate a simple timing test:

The original SQL:

```
SELECT Emp_Number
FROM Dept
WHERE Emp_Number > 100;
```

can be re-written as:

```
INSERT INTO Temp_Tab
  (SELECT Emp_Number
   FROM Dept
   WHERE Emp_Number > 100);
```

A possible disadvantage of this approach is that when SQL*Plus Autotrace is being used to display the execution plan, the `INSERT` method will prevent the execution plan from being shown. The `Timing` function in SQL*Plus will still function, however.

An alternative way to achieve the same result is to modify the `SELECT` clause to specify a function rather than the raw data. Instead of returning each row, we simply force the optimizer to fetch each row as though it is going to be displayed. The execution plan will normally remain the same, and SQL*Plus `AUTOTRACE` will work fine.

Continuing with our same example, this is how the SQL text could be reworded. The original SQL:

```
SELECT Emp_Number
FROM Dept
WHERE Emp_Number > 100;
```

can be rewritten as:

```
SELECT MAX (Emp_Number)
FROM Dept
WHERE Emp_Number > 100;
```

By using the function `MAX`, there is little change in the work that the Oracle engine will have to perform. The Oracle engine will still have to retrieve exactly the same rows as before; the only difference is that they will not be displayed. Admittedly, there will be a very slight impact due to the new function, but this effect is usually negligible – especially when compared to the magnitude of the performance problem being investigated.

When simplifying a SQL query to use the function approach, be careful not to disturb the execution plan. For instance, if certain fields are removed from the `SELECT` clause, the optimizer may modify the execution plan to take advantage of the lessened requirement. For instance, with fields removed from the `SELECT` clause, an index access might retrieve all the information needed, where a table access was required before. When in doubt, compare the execution plans for the two SQL statements.

Summary

In this chapter, we have presented a general overview of exactly what the Oracle Pathologist is trying to accomplish, as well as a procedure that he can follow. We learned that the very definition of root cause suggests a few courses of action that we might follow. They follow, namely:

- ❑ The simplify technique to remove unnecessary distractions and complications.
- ❑ The divide and conquer technique to break complicated problems into manageable pieces.
- ❑ The timeline technique to discover what changed.

We also saw that, amongst other things, performance problems frequently arise because the SQL code executed on the server is poorly optimized. This means that the Oracle Pathologist must become an expert in optimizing SQL code.

Chapters 8 through 10 continue our discussion about the Oracle Pathologist, and his critical job of finding a root cause. These follow-on chapters deal in detail with topics that a competent performance analyst will need to understand. The critical topic of SQL optimization is discussed in the next chapter.

