Developing Successful Oracle Applications

I spend the bulk of my time working with Oracle database software and, more to the point, with people who use this software. Over the last twelve years, I've worked on many projects – successful ones as well as failures, and if I were to encapsulate my experiences into a few broad statements, they would be:

- An application built around the database – dependent on the database – will succeed or fail based on how it uses the database.
- A development team needs at its heart a core of 'database savvy' coders who are responsible for ensuring the database logic is sound and the system is tuned.

These may seem like surprisingly obvious statements, but in my experience, I have found that too many people approach the database as if it were a 'black box' – something that they don't need to know about. Maybe they have a SQL generator that will save them from the hardship of having to learn SQL. Maybe they figure they will just use it like a flat file and do 'keyed reads'. Whatever they figure, I can tell you that thinking along these lines is most certainly misguided; you simply cannot get away with not understanding the database. This chapter will discuss *why* you need to know about the database, specifically why you need to understand:

- The database architecture, how it works, and what it looks like.
- What concurrency controls are, and what they mean to you.
- How to tune your application from day one.
- How some things are implemented in the database, which is not necessary the same as how you think they should be implemented.
- What features your database already provides for you and why it is generally better to use a provided feature then to build your own.
- Why you might want more than a cursory knowledge of SQL.

Now this may seem like a long list of things to learn before you start, but consider this analogy for a second: if you were developing a highly scalable, enterprise application on a brand new operating system (OS), what would be the first thing you would do? Hopefully, you answered, 'find out how this new OS works, how things will run on it, and so on'. If you did not, you would fail.

Consider, for example, one of the early versions of Windows (Windows 3.x, say). Now this, like UNIX, was a 'multi-tasking' operating system. However, it certainly didn't multi-task like UNIX did – it used a non-preemptive multi-tasking model (meaning that if the running application didn't give up control, nothing else could run – including the operating system). In fact, compared to UNIX, Windows 3.x was not really a multi-tasking OS at all. Developers had to understand exactly how the Windows 'multi-tasking' feature was implemented in order to develop effectively. If you sit down to develop an application that will run natively on an OS, then understanding that OS is very important.

What is true of applications running natively on operating systems is true of applications that will run on a database: understanding that database is crucial to your success. If you do not understand what your particular database does or how it does it, your application will fail. If you assume that because your application ran fine on SQL Server, it will necessarily run fine on Oracle then, again, your application is likely to fail.

My Approach

Before we begin, I feel it is only fair that you understand my approach to development. I tend to take a database-centric approach to problems. If I can do it in the database, I will. There are a couple of reasons for this – the first and foremost being that I know that if I build functionality in the database, I can *deploy* it anywhere. I am not aware of a server operating system on which Oracle is not available – from Windows to dozens of UNIX systems to the OS/390 mainframe, the same exact Oracle software and options are available. I frequently build and test solutions on my laptop, running Oracle8*i* on Windows NT. I deploy them on a variety of UNIX servers running the same database software. When I have to implement a feature outside of the database, I find it extremely hard to deploy that feature anywhere I want. One of the main features that makes Java appealing to many people – the fact that their programs are always compiled in the same virtual environment, the **J**ava **V**irtual **M**achine (**JVM**), and so are highly portable – is the exact same feature that make the database appealing to me. The database is *my* Virtual Machine. It is *my* 'virtual operating system'.

My approach is to do everything I can in the database. If my requirements go beyond what the database environment can offer, I do it in Java outside of the database. In this way, almost every operating system intricacy will be hidden from me. I still have to understand how *my* 'virtual machines' work (Oracle, and occasionally a JVM) – you need to know the tools you are using – but they, in turn, worry about how best to do things on a given OS for me.

Thus, simply knowing the intricacies of this one 'virtual OS' allows you to build applications that will perform and scale well on many operating systems. I do not intend to imply that you can be totally ignorant of your underlying OS – just that as a software developer building database applications you can be fairly well insulated from it, and you will not have to deal with many of its nuances. Your DBA, responsible for running the Oracle software, will be infinitely more in tune with the OS (if he or she is not, please get a new DBA!). If you develop client-server software and the bulk of your code is outside of the database and outside of a VM (Java Virtual Machines perhaps being the most popular VM), you will have to be concerned about your OS once again.

I have a pretty simple mantra when it comes to developing database software:

- You should do it in a single SQL statement if at all possible.
- If you cannot do it in a single SQL Statement, then do it in PL/SQL.
- If you cannot do it in PL/SQL, try a Java Stored Procedure.
- If you cannot do it in Java, do it in a C external procedure.
- If you cannot do it in a C external routine, you might want to seriously think about why it is you need to do it...

Throughout this book, you will see the above philosophy implemented. We'll use PL/SQL and Object Types in PL/SQL to do things that SQL itself cannot do. PL/SQL has been around for a very long time, over thirteen years of tuning has gone into it, and you will find no other language so tightly coupled with SQL, nor any as optimized to interact with SQL. When PL/SQL runs out of steam – for example, when we want to access the network, send e-mails' and so on – we'll use Java. Occasionally, we'll do something in C, but typically only when C is the only choice, or when the raw speed offered by C is required. In many cases today this last reason goes away with native compilation of Java – the ability to convert your Java bytecode into operating system specific object code on your platform. This lets Java run just as fast as C.

The Black Box Approach

I have an idea, borne out by first-hand experience, as to why database-backed software development efforts so frequently fail. Let me be clear that I'm including here those projects that may not be documented as failures, but take much longer to roll out and deploy than originally planned because of the need to perform a major 're-write', 're-architecture', or 'tuning' effort. Personally, I call these delayed projects 'failures': more often than not they could have been completed on schedule (or even faster).

The single most common reason for failure is a lack of practical knowledge of the database – a basic lack of understanding of the fundamental tool that is being used. The 'blackbox' approach involves a conscious decision to protect the developers from the database. They are actually encouraged not to learn anything about it! In many cases, they are prevented from exploiting it. The reasons for this approach appear to be FUD-related (**F**ear, **U**ncertainty, and **D**oubt). They have heard that databases are 'hard', that SQL, transactions and data integrity are 'hard'. The solution – don't make anyone do anything 'hard'. They treat the database as a black box and have some software tool generate all of the code. They try to insulate themselves with many layers of protection so that they do not have to touch this 'hard' database.

This is an approach to database development that I've never been able to understand. One of the reasons I have difficulty understanding this approach is that, for me, learning Java and C was a lot harder then learning the concepts behind the database. I'm now pretty good at Java and C but it took a lot more hands-on experience for me to become competent using them than it did to become competent using the database. With the database, you need to be aware of how it works but you don't have to know everything inside and out. When programming in C or Java, you do need to know everything inside and out and these are *huge* languages.

Another reason is that if you are building a database application, then *the most important piece of software is the database*. A successful development team will appreciate this and will want its people to know about it, to concentrate on it. Many times I've walked into a project where almost the opposite was true.

A typical scenario would be as follows:

- The developers were fully trained in the GUI tool or the language they were using to build the front end (such as Java). In many cases, they had had weeks if not months of training in it.
- The team had zero hours of Oracle training and zero hours of Oracle experience. Most had no database experience whatsoever.
- They had massive performance problems, data integrity problems, hanging issues and the like (but very pretty screens).

As a result of the inevitable performance problems, I would be called in to help solve the difficulties. I can recall one particular occasion when I could not fully remember the syntax of a new command that we needed to use. I asked for the *SQL Reference* manual, and I was handed an Oracle 6.0 document. The development was taking place on version 7.3, five years after the release of version.6.0! It was all they had to work with, but this did not seem to concern them at all. Never mind the fact that the tool they really needed to know about for tracing and tuning didn't really exist back then. Never mind the fact that features such as triggers, stored procedures, and many hundreds of others, had been added in the five years since the documentation to which they had access was written. It was very easy to determine why they needed help– fixing their problems was another issue all together.

The idea that developers building a **database application** should be shielded from the database is amazing to me but still the attitude persists. Many people still take the attitude that developers should be shielded from the database, they cannot take the time to get trained in the database – basically, they should not have to know anything about the database. Why? Well, more than once I've heard '... but Oracle is the most scalable database in the world, my people don't have to learn about it, it'll just do that'. It is true; Oracle is the most scalable database in the world. However, I can write bad code that does not scale in Oracle easier then I can write good, scaleable code in Oracle. You can replace Oracle with any technology and the same will be true. This is a fact – it is easier to write applications that perform poorly than it is to write applications that perform well. It is sometimes too easy to build a single-user system in the world's most scalable database if you don't know what you are doing. The database is a tool and the improper use of any tool can lead to disaster. Would you take a nutcracker and smash walnuts with it as if it were a hammer? You could but it would not be a proper use of that tool and the result would be a mess. Similar effects can be achieved by remaining ignorant of your database.

I was recently working on a project where the system architects had designed a very elegant architecture. A web browser client would talk over HTTP to an application server running Java Server Pages (JSP). The application logic would be 100 percent generated by a tool and implemented as EJBs (using container managed persistence) and would be physically located on another application server. The database would hold tables and indexes and nothing else.

So, we start with a technically complex architecture: we have four entities that must talk to each other in order to get the job done: web browser to a JSP in the Application Server to an EJB to the database. It would take technically competent people to develop, test, tune, and deploy this application. I was asked to help benchmark this application post-development. The first thing I wanted to know about was their approach to the database:

- What did they feel would be the major choke points, areas of contention?
- What did they view as the major obstacles to overcome?

They had no idea. When asked, 'OK, when we need to tune a generated query, who can help me rewrite the code in the EJB?' The answer was, 'Oh, you cannot tune that code, you have to do it all in the database'. The application was to remain untouched. At that point, I was ready to walk away from the project – it was already clear to me that there was no way this application would work:

- The application was built without a single consideration for scaling the database level.
- The application itself could not be tuned or touched.
- Experience shows that 80 to 90 percent of *all* tuning is done at the application level, not at the database level.
- The developers had no idea what the beans did in the database or where to look for potential problems.

That was shown to be the case in the first hour of testing. As it turns out, the first thing the application did was a:

```
select * from t for update;
```

What this did was to force a serialization of *all* work. The model implemented in the database was such that before any significant work could proceed, you had to lock an extremely scarce resource. That immediately turned this application into a very large single user system. The developers did not believe me (in another database, employing a shared read lock, the observed behavior was different). After spending ten minutes with a tool called `TKPROF` (you'll hear a lot more about this in *Tuning Strategies and Tools*, Chapter 10) I was able to show them that, yes, in fact this was the SQL executed by the application (they had no idea – they had never seen the SQL). Not only was it the SQL executed by the application but by using two SQL*PLUS sessions I was able to show them that session two will wait for session one to completely finish its work before proceeding.

So, instead of spending a week benchmarking the application, I spent the time teaching them about tuning, database locking, concurrency control mechanisms, how it worked in Oracle versus Informix versus SQL Server versus DB2 and so on (it is different in each case). What I had to understand first, though, was the *reason* for the `SELECT FOR UPDATE`. It turned out the developers wanted a repeatable read.

> *Repeatable read is a database term that says if I read a row once in my transaction and I read the row again later in the same transaction, the row will not have changed – the read is repeatable.*

Why did they want this? They had heard it was a 'good thing'. OK, fair enough, you want repeatable read. The way to do that in Oracle is to set the isolation level to **serializable** (which not only gives you a repeatable read for any row of data, it gives you a repeatable read for a query – if you execute the same query two times in a transaction, you'll get the same results). To get a repeatable read in Oracle, you do not want to use `SELECT FOR UPDATE`, which you only do when you want to physically serialize access to data. Unfortunately, the tool they utilized did not know about that – it was developed primarily for use with another database where this *was* the way to get a repeatable read.

So, what we had to do in this case, in order to achieve serializable transactions, was to create a logon trigger in the database that altered the session for these applications and set the isolation level to `serializable`. We went back to the tool they were using and turned off the switches for repeatable reads and re-ran the application. Now, with the `FOR UPDATE` clause removed, we got some actual concurrent work done in the database.

That was hardly the end of the problems on this project. We had to figure out:

- How to tune SQL without changing the SQL (that's hard, we'll look at some methods in Chapter 11 on *Optimizer Plan Stability*).
- How to measure performance.
- How to see where the bottlenecks were.
- How and what to index. And so on.

At the end of the week the developers, who had been insulated from the database, were amazed at what the database could actually provide for them, how easy it was to get that information and, most importantly, how big a difference it could make to the performance of their application. We didn't do the benchmark that week (they had some reworking to do!) but in the end they were successful – just behind schedule by a couple of weeks.

This is not a criticism of tools or technologies like EJBs and container managed persistence. This is a criticism of purposely remaining ignorant of the database and how it works and how to use it. The technologies used in this case worked well – after the developers got some insight into the database itself.

The bottom line is that the database is typically the cornerstone of your application. If it does not work well, nothing else really matters. If you have a black box and it does not work well – what are you going to do about it? About the only thing you can do is look at it and wonder why it is not doing so well. You cannot fix it, you cannot tune it, you quite simply do not understand how it works – and you made the decision to be in this position. The alternative is the approach that I advocate: understand your database, know how it works, know what it can do for you, and use it to its fullest potential.

How (and how not) to Develop Database Applications

That's enough hypothesizing, for now at least. In the remainder of this chapter, I will take a more empirical approach, discussing why knowledge of the database and its workings will definitely go a long way towards a successful implementation (without having to write the application twice!). Some problems are simple to fix as long as you understand how to find them. Others require drastic rewrites. One of the goals of this book is to help you avoid the problems in the first place.

> *In the following sections, I discuss certain core Oracle features without delving into exactly what these features are and all of the ramifications of using them. For example, I discuss just one of the implications of using Multi-Threaded Server (MTS) architecture– a mode in which you can (and sometimes have to) configure Oracle in order to support multiple database connections. I will not, however, go fully into what MTS is, how it works and so on. Those facts are covered in detail in the Oracle Server Concepts Manual (with more information to be found in the Net8 Administrators Guide).*

Understanding Oracle Architecture

I was working on a project recently where they decided to use only the latest, greatest technologies: everything was coded in Java with EJBs. The client application would talk to the database server using beans – no Net8. They would not be passing SQL back and forth between client and server, just EJB calls using Remote Method Invocation (RMI) over Internet Inter-Orb Protocol (IIOP).

> *If you are interested in the details of RMI over IIOP you can refer to* http://java.sun.com/products/rmi-iiop/.

This is a perfectly valid approach. This functionality works and can be extremely scalable. The people responsible for the architecture understood Java, EJBs, the protocols involved – all of that stuff. They felt they were in a strong position to successfully build such a project. When their application would not scale beyond a couple of users they decided that the database was at fault and severely doubted Oracle's claim to be the 'most scaleable database ever'.

The problem was not the database but a lack of knowledge of how the database worked – a lack of knowledge that meant that certain key decisions were made at design time that doomed this particular application to failure. In order to deploy EJBs in the database Oracle must be configured to run in MTS mode rather than dedicated server mode. What the team did not understand, crucially, was how using MTS with EJBs in the database would affect them. Without this understanding, and without a general knowledge of how Oracle worked, two key decisions were made:

- We will run some stored procedures that take 45 seconds or longer (much longer at times) in our beans.
- We will not support the use of bind variables. All of our queries will hard code the constant values in the predicate. All inputs to stored procedures will use strings. This is 'easier' than coding bind variables.

These two seemingly minor decisions guaranteed that the project would fail – utterly guaranteed it. They made it so that a highly scalable database would literally fall over and fail with a very small user load. A lack of knowledge of how the database worked more then overwhelmed their intricate knowledge of Java beans and distributed processing. If they had taken time to learn a bit more about the way Oracle worked, and consequently followed the following two simple guidelines, then their project would have had a much better chance of success the first time out.

Do not run Long Transactions Under MTS

The decision to run 45+ second transactions under MTS betrayed a lack of understanding of what MTS was designed to do and how it works in Oracle. Briefly, MTS works by having a shared pool of server processes that service a larger pool of end users. It is very much like connection pooling – since process creation and management are some of the most expensive operations you can ask an operating system to perform, MTS is very beneficial in a large-scale system. So, I might have 100 users but only five or ten shared servers.

When a shared server gets a request to run an update, or execute a stored procedure, then that shared server is dedicated to that task until completion. No one else will use that shared server until that update completes or that stored procedure finishes execution. Thus, when using MTS your goal must be to have very short statements. MTS is designed to scale up **O**n-**L**ine **T**ransaction **P**rocessing (**OLTP**) systems – a system characterized by statements that execute with sub-second response times. You'll have a single row update, insert a couple of line items, and query records by primary key. You won't (or shouldn't) run a batch process that takes many seconds or minutes to complete.

If all of our statements execute very rapidly, then MTS works well. We can effectively share a number of processes amongst a larger community of users. If, on the other hand, we have sessions that monopolize a shared server for extended periods of time then we will see apparent database 'hangs'. Say we configured ten shared servers for 100 people. If, at some point, ten people simultaneously execute the process that takes 45 seconds or longer then every other transaction (including new connections) will have to wait. If some of the queued sessions want to run that same long process, then we have a *big* problem – the apparent 'hang' won't last 45 seconds, it will appear to last much longer for most people. Even if we only have a few people wanting to execute this process simultaneously rather than ten, we will still observe what appears to be a large degradation in performance from the server. We are taking away, for an extended period of time, a shared resource and this is not a good thing. Instead of having ten shared servers processing quick requests on a queue, we now have five or six (or less). Eventually the system will be running at some fraction of its capability, solely due to this resource being consumed.

The 'quick and dirty' solution was to start up more shared servers, but the logical conclusion to this is that you need a shared server per user and this is not a reasonable conclusion for a system with thousands of users (as this system was). Not only would that introduce bottlenecks into the system itself (the more servers you have to manage – the more processing time spent managing), but also it is simply not the way MTS was designed to work.

The real solution to this problem was simple: do not execute long running transactions under MTS. Implementing this solution was not. There was more then one way to implement this and they all required fundamental architectural changes. The most appropriate way to fix this issue, requiring the least amount of change, was to use **A**dvanced **Q**ueues (**AQ**).

> *AQ is a message-oriented middleware hosted in the Oracle database. It provides the ability for a client session to enqueue a message into a database queue table. This message is later, typically immediately after committing, 'dequeued' by another session and the content of the message is inspected. This message contains information for the other session to process. It can be used to give the appearance of lightening fast response times by decoupling the long running process from the interactive client.*

So, rather than execute a 45-second process, the bean would place the request, along with all its inputs, on a queue and execute it in a loosely coupled (asynchronous) rather than tightly coupled (synchronous) fashion. In this way, the end user would not have to wait 45 seconds for a response – the system would apparently be much more responsive

While this approach sounds easy – just drop in 'AQ' and the problem is fixed – there was more to it than that. This 45-second process generated a transaction ID that was required by the next step in the interface in order to join to other tables – as designed, the interface would not work without it. By implementing AQ, we were not waiting for this transaction ID to be generated here – we were just asking the system to do it for us at some point. So, the application was stuck. On the one hand, we could not wait 45 seconds for the process to complete, but on the other hand, we needed the generated ID in order to proceed to the next screen and we could only get that after waiting 45 seconds. To solve this problem, what we had to do was to synthesize a pseudo-transaction ID, modify the long running process to accept this generated pseudo ID and have it update a table when it was done, by which mechanism the real transaction ID was associated with the pseudo id. That is, instead of the transaction ID being an output of the long running process, it would be an input to it. Further, all 'downstream' tables would have to use this pseudo-transaction ID – not the real one (since the real one would not be generated for a while). We also had to review the usage of this transaction ID in order to see what impact this change might have on other modules and so on.

Another consideration was the fact that, while we were running synchronously, if the 45-second process failed then the end user was alerted right away. They would fix the error condition (fix the inputs, typically) and resubmit the request. Now, with the transaction being processed asynchronously under AQ, we don't have that ability. New functionality had to be added in order to support this delayed response. Specifically, we needed some workflow mechanism to route any failed transaction to the appropriate person.

The upshot of all this is that we had to undertake major changes in the database structure. New software had to be added (AQ). New processes had to be developed (workflows and such). On the plus side, the removal of 45 seconds of lag time from an interactive process not only solved the MTS architecture issue, it enhanced the user experience – it meant that the end user got much faster 'apparent' response times. On the down side, all of this delayed the project considerably because none of it was detected until immediately before deployment, during scalability testing. It is just too bad that it was not designed the right way from the beginning. With knowledge of how MTS worked physically, it would have been clear that the original design would not scale very well.

Use Bind Variables

If I were to write a book about how to build *non-scalable* Oracle applications, then *Don't use Bind Variables* would be the first and last chapter. This is a major cause of performance issues and a major inhibitor of scalability. The way the Oracle shared pool (a very important shared memory data structure) operates is predicated on developers using bind variables. If you want to make Oracle run slowly, even grind to a total halt – just refuse to use them.

Bind variable is a placeholder in a query. For example, to retrieve the record for employee `123`, I can query:

```
select * from emp where empno = 123;
```

Alternatively, I can query:

```
select * from emp where empno = :empno;
```

In a typical system, you would query up employee `123` maybe once and then never again. Later, you would query up employee `456`, then `789`, and so on. If you use literals (constants) in the query then each and every query is a brand new query, never before seen by the database. It will have to be parsed, qualified (names resolved), security checked, optimized, and so on – in short, each and every unique statement you execute will have to be compiled every time it is executed.

The second query uses a bind variable, `:empno`, the value of which is supplied at query execution time. This query is compiled once and then the query plan is stored in a shared pool (the library cache), from which it can be retrieved and reused. The difference between the two in terms of performance and scalability is huge, dramatic even.

From the above description it should be fairly obvious that parsing a statement with hard-coded variables (called a **hard** parse) will take longer and consume many more resources than reusing an already parsed query plan (called a **soft** parse). What may not be so obvious is the extent to which the former will reduce the number of users your system can support. Obviously, this is due in part to the increased resource consumption, but an even larger factor arises due to the latching mechanisms for the library cache. When you hard parse a query, the database will spend more time holding certain low-level serialization devices called **latches** (see Chapter 3, *Locking and Concurrency*, for more details). These latches protect the data structures in the shared memory of Oracle from concurrent modifications by

two sessions (else Oracle would end up with corrupt data structures) and from someone reading a data structure while it is being modified. The longer and more frequently we have to latch these data structures, the longer the queue to get these latches will become. In a similar fashion to having long transactions running under MTS, we will start to monopolize scarce resources. Your machine may appear to be under-utilized at times – and yet everything in the database is running very slowly. The likelihood is that someone is holding one of these serialization mechanisms and a line is forming – you are not able to run at top speed.It only takes one ill behaved application in your database to dramatically affect the performance of every other application. A single, small application that does not use bind variable will cause the relevant SQL of other well tuned applications to get discarded from the shared pool over time. You only need one bad apple to spoil the entire barrel.

If you use bind variables, then everyone who submits the same exact query that references the same object will use the compiled plan from the pool. You will compile your subroutine once and use it over and over again. This is very efficient and is the way the database intends you to work. Not only will you use fewer resources (a soft parse is much less resource intensive), but also you will hold latches for less time and need them less frequently. This increases your performance and greatly increases your scalability.

Just to give you a tiny idea of how huge a difference this can make performance-wise, you only need to run a very small test:

```
tkyte@TKYTE816> alter system flush shared_pool;

System altered.
```

Here I am starting with an 'empty' shared pool. If I was to run this test more than one time, I would need to flush the shared pool every time, or else the non-bind variable SQL below would, in fact, be cached and appear to run very fast.

```
tkyte@TKYTE816> set timing on
tkyte@TKYTE816> declare
  2      type rc is ref cursor;
  3      l_rc rc;
  4      l_dummy all_objects.object_name%type;
  5      l_start number default dbms_utility.get_time;
  6  begin
  7      for i in 1 .. 1000
  8      loop
  9          open l_rc for
 10          'select object_name
 11             from all_objects
 12            where object_id = ' || i;
 13          fetch l_rc into l_dummy;
 14          close l_rc;
 15      end loop;
 16      dbms_output.put_line
 17      ( round( (dbms_utility.get_time-l_start)/100, 2 ) ||
 18        ' seconds...' );
 19  end;
 20  /
14.86 seconds...

PL/SQL procedure successfully completed.
```

The above code uses dynamic SQL to query out a single row from the ALL_OBJECTS table. It generates 1000 unique queries with the values 1, 2, 3, ... and so on 'hard-coded' into the WHERE clause. On my 350MHz Pentium laptop, this took about 15 seconds (the speed may vary on different machines).

Next, we do it using bind variables:

```
tkyte@TKYTE816> declare
  2      type rc is ref cursor;
  3      l_rc rc;
  4      l_dummy all_objects.object_name%type;
  5      l_start number default dbms_utility.get_time;
  6  begin
  7      for i in 1 .. 1000
  8      loop
  9          open l_rc for
 10          'select object_name
 11             from all_objects
 12            where object_id = :x'
 13          using i;
 14          fetch l_rc into l_dummy;
 15          close l_rc;
 16      end loop;
 17      dbms_output.put_line
 18      ( round( (dbms_utility.get_time-l_start)/100, 2 ) ||
 19        ' seconds...' );
 20  end;
 21  /
1.27 seconds...

PL/SQL procedure successfully completed.
```

We use the same logic here – the only thing that has changed is the fact that we are not hard coding the values 1, 2, 3… and so on into the query – we are using bind variables instead. The results are pretty dramatic. The fact is that not only does this execute much faster (we spent more time *parsing* our queries then actually *executing* them!) it will let more users use your system simultaneously.

Executing SQL statements without bind variables is very much like compiling a subroutine before each and every method call. Imagine shipping Java source code to your customers where, before calling a method in a class, they had to invoke the Java compiler, compile the class, run the method, and then throw away the byte code. Next time they wanted to execute the exact same method, they would do the same thing; compile it, run it, and throw it away. You would never consider doing this in your application – you should never consider doing this in your database either.

> *In Chapter10, Tuning Strategies and Tools, we will look at ways to identify whether or not you are using bind variables, different ways to use them, an 'auto binder' feature in the database and so on. We will also discuss a specialized case where you don't want to use bind variables.*

As it was, on this particular project, rewriting the existing code to use bind variables was the only possible course of action. The resulting code ran orders of magnitude faster and increased many times the number of simultaneous users that the system could support. However, it came at a high price in terms of time and effort. It is not that using bind variables is hard, or error prone, it's just that they did not do it initially and thus were forced to go back and revisit virtually *all* of the code and change it. They would not have paid this price if they had understood that it was vital to use bind variables in their application from day one.

Understanding Concurrency Control

Concurrency control is one area where databases differentiate themselves. It is an area that sets a database apart from a file system and that sets databases apart from each other. As a programmer, it is vital that your database application works correctly under concurrent access conditions, and yet this is something people fail to test time and time again. Techniques that work well if everything happens consecutively do not work so well when everyone does them simultaneously. If you don't have a good grasp of how your particular database implements concurrency control mechanisms, then you will:

- Corrupt the integrity of your data.
- Run slower than you should with a small number of users.
- Decrease your ability to scale to a large number of users.

Notice I don't say, 'you might...' or 'you run the risk of...' but rather that invariably you *will* do these things. You will do these things without even realizing it. Without correct concurrency control, you will corrupt the integrity of your database because something that works in isolation will not work as you expect in a multi-user situation. You will run slower than you should because you'll end up waiting for data. You'll lose your ability to scale because of locking and contention issues. As the queues to access a resource get longer, the wait gets longer and longer. An analogy here would be a backup at a tollbooth. If cars arrive in an orderly, predictable fashion, one after the other, we never have a backup. If many cars arrive simultaneously, queues start to form. Furthermore, the waiting time does not increase in line with the number of cars at the booth. After a certain point we are spending considerable additional time 'managing' the people that are waiting in line, as well as servicing them (in the database, we would talk about context switching).

Concurrency issues are the hardest to track down – the problem is similar to debugging a multi-threaded program. The program may work fine in the controlled, artificial environment of the debugger but crashes horribly in the 'real world'. For example, under 'race conditions' you find that two threads can end up modifying the same data structure simultaneously. These kinds of bugs are terribly hard to track down and fix. If you only test your application in isolation and then deploy it to dozens of concurrent users, you are likely to be (painfully) exposed to an undetected concurrency issue.

Over the next two sections, I'll relate two small examples of how the lack of understanding concurrency control can ruin your data or inhibit performance and scalability.

Implementing Locking

The database uses locks to ensure that, at most, one transaction is modifying a given piece of data at any given time. Basically, they are the mechanism that allows for concurrency – without some locking model to prevent concurrent updates to the same row, for example, multi-user access would not be possible in a database. However, if overused or used improperly, locks can actually inhibit concurrency. If you or the database itself locks data unnecessarily, then fewer people will be able to concurrently perform operations. Thus, understanding what locking is and how it works in your database is vital if you are to develop a scalable, correct application.

What is also vital is that you understand that each database implements locking differently. Some have page-level locking, others row level; some implementations escalate locks from row-level to page-level, some do not; some use read locks, others do not; some implement serializable transactions via locking and others via read-consistent views of data (no locks). These small differences can balloon into huge performance issues or downright bugs in your application if you do not understand how they work.

The following points sum up Oracle's locking policy:

- Oracle locks data at the row level on modification only. There is no lock escalation to a block or table level, ever.
- Oracle never locks data just to read it. There are no locks placed on rows of data by simple reads.
- A writer of data does not block a reader of data. Let me repeat – *reads* are not blocked by *writes*. This is fundamentally different from almost every other database, where reads are blocked by writes.
- A writer of data is blocked only when another writer of data has already locked the row it was going after. A reader of data never blocks a writer of data.

These facts must be taken into consideration when developing your application and you must also realize that this policy is unique to Oracle. A developer who does not understand how his or her database handles concurrency will certainly encounter data integrity issues (this is particularly common when a developer moves from another database to Oracle, or vice versa, and neglects to take the differing concurrency mechanisms into account in their application.

One of the side-effects of Oracle's 'non-blocking' approach is that if you actually want to ensure that no more than one user has access to a row at once, then you, the developer, need to do a little work yourself. Consider the following example. A developer was demonstrating to me a resource-scheduling program (for conference rooms, projectors, etc.) that he had just developed and was in the process of deploying. The application implemented a business rule to prevent the allocation of a resource to more than one person, for any given period of time. That is, the application contained code that specifically checked that no other user had previously allocated the time slot (as least the developer thought it did). This code queried the `schedules` table and, if no rows existed that overlapped that time slot, inserted the new row. So, the developer was basically concerned with two tables:

```
create table resources ( resource_name varchar2(25) primary key, ... );
create table schedules( resource_name varchar2(25) references resources,
                        start_time    date,
                        end_time      date );
```

And, before making, say, a room reservation, the application would query:

```
select count(*)
  from schedules
 where resource_name = :room_name
   and (start_time between :new_start_time and :new_end_time
        rr
        end_time between :new_start_time and :new_end_time)
```

It looked simple and bullet-proof (to the developer anyway); if the count came back zero, the room was yours. If it came back non-zero, you could not reserve it for that period. Once I knew what his logic was, I set up a very simple test to show him the error that would occur when the application went live. An error that would be incredibly hard to track down and diagnose after the fact – one would be convinced it *must* be a database bug.

All I did was get someone else to use the terminal next to him. They both navigated to the same screen and, on the count of three, each hit the Go button and tried to reserve the same room for the exact same time. Both people got the reservation – the logic, which worked perfectly in isolation, failed in a multi-user environment. The problem in this case was caused by Oracle's non-blocking reads. Neither session ever blocked the other session. Both sessions simply ran the above query and then performed the logic to schedule the room. They could both run the query to look for a reservation, even if the other session had already started to modify the `schedules` table (the change wouldn't be visible to the other session until commit, by which time it was too late). Since they were never attempting to modify the same row in the `schedules` table, they would never block each other and, thus, the business rule could not enforce what it was intended to enforce.

The developer needed a method of enforcing the business rule in a multi-user environment, a way to ensure that exactly one person at a time made a reservation on a given resource. In this case, the solution was to impose a little serialization of his own – in addition to performing the `count(*)` above, the developer must first:

```
select * from resources where resource_name = :room_name FOR UPDATE;
```

A little earlier in the chapter, we discussed an example where use of the `FOR UPDATE` clause caused problems, but here it is what makes this business rule work in the way intended. What we did here was to lock the resource (the room) to be scheduled immediately *before* scheduling it, in other words before we query the `Schedules` table for that resource. By locking the resource we are trying to schedule, we have ensured that no one else is modifying the schedule for this resource simultaneously. They must wait until we commit our transaction – at which point, they would be able to see our schedule. The chance of overlapping schedules is removed. The developer must understand that, in the multi-user environment, they must at times employ techniques similar to those used in multi-threaded programming. The `FOR UPDATE` clause is working like a semaphore in this case. It serializes access to the `resources` tables for that particular row – ensuring no two people can schedule it simultaneously.

This is still highly concurrent as there are potentially thousands of resources to be reserved – what we have done is ensure that only one person modifies a resource at any time. This is a rare case where the manual locking of data you are not going to actually update is called for. You need to be able to recognize where you need to do this and, perhaps as importantly, where not to (I have an example of when not to below). Additionally, this does not lock the resource from other people reading the data as it might in other databases, hence this will scale very well.

Issues such as the above have massive implications when attempting to port an application from database to database (I return to this theme a little later in the chapter), and this trips people up time and time again. For example, if you are experienced in other databases, where writers block readers and vice versa then you may have grown reliant on that fact to protect you from data integrity issues. The *lack* of concurrency is one way to protect yourself from this – that is how it works in many non-Oracle databases. In Oracle, concurrency rules supreme and you must be aware that, as a result, things will happen differently (or suffer the consequences).

For 99 percent of the time, locking is totally transparent and you need not concern yourself with it. It is that other 1 percent that you must be trained to recognize. There is no simple checklist of 'if you do this, you need to do this' for this issue. This is a matter of understanding how your application will behave in a multi-user environment and how it will behave in your database.

Multi-Versioning

This is a topic very closely related to concurrency control, as it forms the foundation for Oracle's concurrency control mechanism – Oracle operates a multi-version read-consistent concurrency model. In Chapter 3, *Locking and Concurrency*, we'll cover the technical aspects of this in more detail but, essentially, it is the mechanism by which Oracle provides for:

- **Read-consistent queries**: Queries that produce consistent results with respect to a point in time.
- **Non-blocking queries**: Queries are never blocked by writers of data, as they would be in other databases.

These are two very important concepts in the Oracle database. The term multi-versioning basically comes from the fact that Oracle is able to simultaneously maintain multiple versions of the data in the database. If you understand how multi-versioning works, you will always understand the answers you get from the database. Before we explore in a little more detail how Oracle does this, here is the simplest way I know to *demonstrate* multi-versioning in Oracle:

```
tkyte@TKYTE816> create table t
  2  as
  3  select * from all_users;
Table created.

tkyte@TKYTE816> variable x refcursor

tkyte@TKYTE816> begin
  2          open :x for select * from t;
  3  end;
  4  /

PL/SQL procedure successfully completed.

tkyte@TKYTE816> delete from t;

18 rows deleted.

tkyte@TKYTE816> commit;

Commit complete.

tkyte@TKYTE816> print x

USERNAME                          USER_ID CREATED
------------------------------ ---------- ---------
SYS                                     0 04-NOV-00
SYSTEM                                  5 04-NOV-00
DBSNMP                                 16 04-NOV-00
AURORA$ORB$UNAUTHENTICATED             24 04-NOV-00
ORDSYS                                 25 04-NOV-00
ORDPLUGINS                             26 04-NOV-00
MDSYS                                  27 04-NOV-00
CTXSYS                                 30 04-NOV-00
...
DEMO                                   57 07-FEB-01

18 rows selected.
```

In the above example, we created a test table, T, and loaded it with some data from the ALL_USERS table. We opened a cursor on that table. We fetched *no data* from that cursor: we just opened it.

> *Bear in mind that Oracle and does not 'answer' the query, does not copy the data anywhere when you open a cursor – imagine how long it would take to open a cursor on a one billion row table if it did. The cursor opens instantly and it answers the query as it goes along. In other words, it would just read data from the table as you fetched from it.*

In the same session (or maybe another session would do this), we then proceeded to delete all data from that table. We even went as far as to COMMIT work on that delete. The rows are gone – but are they? In fact, they are retrievable via the cursor. The fact is that the resultset returned to us by the OPEN command was pre-ordained at the point in time we opened it. We had touched not a single block of data in that table during the open, but the answer was already fixed in stone. We have no way of knowing what the answer will be until we fetch the data – however the result is immutable from our cursor's perspective. It is not that Oracle copied all of the data above to some other location when we opened the cursor; it was actually the delete command that preserved our data for us by placing it into a data area called a **rollback segment**.

This is what read-consistency is all about and if you do not understand how Oracle's multi-versioning scheme works and what it implies, you will not be able to take full advantage of Oracle nor will you be able to write correct applications in Oracle (ones that will ensure data integrity).

Let's look at the implications of multi-versioning, read-consistent queries and non-blocking reads. If you are not familiar with multi-versioning, what you see below might be surprising. For the sake of simplicity, we will assume that the table we are reading stores one row per database block (the smallest unit of storage in the database), and that we are fullscanning the table in this example.

The table we will query is a simple accounts table. It holds balances in accounts for a bank. It has a very simple structure:

```
create table accounts
( account_number number primary key,
  account_balance number
);
```

In reality the accounts table would have hundreds of thousands of rows in it, but for simplicity we're just going to consider a table with four rows (we will visit this example in more detail in Chapter 3, *Locking and Concurrency*):

Row	Account Number	Account Balance
1	123	$500.00
2	234	$250.00
3	345	$400.00
4	456	$100.00

What we would like to do is to run the end-of-day report that tells us how much money is in the bank. That is an extremely simple query:

```
select sum(account_balance) from accounts;
```

And, of course, in this example the answer is obvious: $1250. However, what happens if we read row 1, and while we're reading rows 2 and 3, an Automated Teller Machine (ATM) generates transactions against this table, and moves $400 from account 123 to account 456? Our query counts $500 in row 4 and comes up with the answer of $1650, doesn't it? Well, of course, this is to be avoided, as it would be an error – at no time did this sum of money exist in the account balance column. It is the way in which Oracle avoids such occurrences, and how Oracle's methods differ from every other database, that you need to understand.

In practically every other database, if you wanted to get a 'consistent' and 'correct' answer to this query, you would either have to lock the whole table while the sum was calculated *or* you would have to lock the rows as you read them. This would prevent people from changing the answer as you are getting it. If you lock the table up-front, you'll get the answer that was in the database at the time the query began. If you lock the data as you read it (commonly referred to as a shared read lock, which prevents updates but not other readers from accessing the data), you'll get the answer that was in the database at the point the query finished. Both of these methods inhibit concurrency a great deal. The table lock would prevent any updates from taking place against the entire table for the duration of your query (for a table of four rows, this would only be a very short period – but for tables with hundred of thousands of rows, this could be several minutes). The 'lock as you go' method would prevent updates on data you have read and already processed and could actually cause deadlocks between your query and other updates.

Now, I said earlier that you would not be able to take full advantage of Oracle if you did not understand the concept of multi-versioning. Here is one reason why that is true. Oracle uses multi-versioning to get the answer, as it existed at the point in time the query began, and the query will take place *without locking a single thing* (while our account transfer transaction updates rows 1 and 4, these rows will be locked to other writers – but not locked to other readers, such as our `SELECT SUM...` query). In fact, Oracle doesn't have a 'shared read' lock common in other databases – it does not need it. Everything inhibiting concurrency that can be removed, has been removed.

So, how does Oracle get the correct, consistent answer ($1250) during a read without locking any data– in other words, without decreasing concurrency? The secret lies in the transactional mechanisms that Oracle uses. Whenever you modify data, Oracle creates entries in two different locations. One entry goes to the redo logs where Oracle stores enough information to **redo** or 'roll forward' the transaction. For an insert this would be the row inserted. For a delete, it is a message to delete the row in file X, block Y, row slot Z. And so on. The other entry is an **undo** entry, written to a rollback segment. If your transaction fails and needs to be undone, Oracle will read the 'before' image from the rollback segment and restore the data. In addition to using this rollback segment data to undo transactions, Oracle uses it to undo changes to blocks as it is reading them – to restore the block to the point in time your query began. This gives you the ability to read right through a lock and to get consistent, correct answers without locking any data yourself.

So, as far as our example is concerned, Oracle arrives at its answer as follows:

Time	Query	Account transfer transaction
T1	Reads row 1, sum = $500 so far	
T2		Updates row 1, puts an exclusive lock on row 1 preventing other updates. Row 1 now has $100
T3	Reads row 2, sum = $750 so far	
T4	Reads row 3, sum = $1150 so far	
T5		Updates row 4, puts an exclusive lock on block 4 preventing other updates (but not reads). Row 4 now has $500.
T6	Reads row 4, discovers that row 4 has been modified. It will actually rollback the block to make it appear as it did at time = T1. The query will read the value $100 from this block	
T7		Commits transaction
T8	Presents $1250 as the answer	

At time T6, Oracle is effectively 'reading through' the lock placed on row 4 by our transaction. This is how non-blocking reads are implemented – Oracle only looks to see if the data changed, it does not care if the data is currently locked (which implies that it has changed). It will simply retrieve the old value from the rollback segment and proceed onto the next block of data.

This is another clear demonstration of multi-versioning – there are multiple versions of the same piece of information, all at different points in time, available in the database. Oracle is able to make use of these 'snapshots' of data at different points in time to provide us with read-consistent queries and non-blocking queries.

This read-consistent view of data is always performed at the SQL statement level, the results of any single SQL statement are consistent with respect to the point in time they began. This quality is what makes a statement like the following insert a predictable set of data:

```
for x in (select * from t)
loop
    insert into t values (x.username, x.user_id, x.created);
end loop;
```

The result of the `SELECT * FROM T` is preordained when the query begins execution. The `SELECT` will not see any of the new data generated by the `INSERT`. Imagine if it did – this statement might be a never-ending loop. If, as the `INSERT` generated more rows in `CUSTOMER`, the `SELECT` could 'see' those newly inserted rows – the above piece of code would create some unknown number of rows. If the table

T started out with 10 rows, we might end up with 20, 21, 23, or an infinite number of rows in T when we finished. It would be totally unpredictable. This consistent read is provided to all statements so that an INSERT such as the following is predicable as well:

```
insert into t select * from t;
```

The INSERT statement will with be provided a read-consistent view of T – it will not see the rows that it itself just inserted, it will only insert the rows that existed at the time the INSERT began. Many databases won't even permit recursive statements such as the above due to the fact that they cannot tell how many rows might actually be inserted.

So, if you are used to the way other databases work with respect to query consistency and concurrency, or you have never had to grapple with such concepts (no real database experience), you can now see how understanding how this works will be important to you. In order to maximize Oracle's potential, you need to understand these issues as they pertain to Oracle – not how they are implemented in other databases.

Database Independence?

By now, you might be able to see where I'm going in this section. I have made references above to other databases and how features are implemented differently in each. With the exception of some read-only applications, it is my contention that building a wholly database-independent application that is highly scalable is extremely hard – and is in fact quite impossible unless you know exactly how each database works in great detail.

For example, let's revisit our initial resource scheduler example (prior to adding the FOR UPDATE clause). Let's say this application had been developed on a database with an entirely different locking/concurrency model from Oracle. What I'll show here is that if you migrate your application from one database to another database you will have to verify that it still works correctly in these different environments.

Let's assume that we had deployed the initial resource scheduler application in a database that employed page-level locking with blocking reads (reads are blocked by writes) and there was an index on the SCHEDULES table:

```
create index schedules_idx on schedules( resource_name, start_time );
```

Also consider that the business rule was implemented via a database trigger (*after* the INSERT had occurred but before the transaction committed we would verify that only our row existed in the table for that time slot). In a page-locking system, due to the update of the index page by RESOURCE_NAME and START_TIME it is very likely that we would have serialized these transactions. The system would have processed these inserts sequentially due to the index page being locked (all of the RESOURCE_NAMEs with START_TIMEs near each other would be on the same page). In that page level locking database our application would be apparently well behaved – our checks on overlapping resource allocations would have happened one after the other, not concurrently.

If we migrated this application to Oracle and simply assumed that it would behave in the same way, we would be in for a shock. On Oracle, which does row level locking and supplies non-blocking reads, it appears to be ill behaved. As we saw previously, we had to use the FOR UPDATE clause to serialize access. Without this clause, two users could schedule the same resource for the same times. This is a direct consequence of not understanding how the database we have works in a multi-user environment.

I have encountered issues such as this many times when an application is being moved from database A to database B. When an application that worked flawlessly in database A does not work, or works in an apparently bizarre fashion, on database B, the first thought is that database B is a 'bad database'. The simple truth is that database B just does it *differently* – neither database is wrong or 'bad', they are just different. Knowing and understanding how they work will help you immensely in dealing with these issues.

For example, very recently I was helping to convert some Transact SQL (the stored procedure language for SQL Server) into PL/SQL. The developer doing the conversion was complaining that the SQL queries in Oracle returned the 'wrong' answer. The queries looked like this:

```
declare
    l_some_variable   varchar2(25);
begin
   if ( some_condition )
   then
       l_some_variable := f( … );
   end if;

   for x in ( select * from T where x = l_some_variable )
   loop
        ...
```

The goal here was to find all of the rows in `T` where `X` was Null if some condition was not met or where `x` equaled a specific value if some condition was met.

The complaint was that, in Oracle, this query would return no data when `L_SOME_VARIABLE` was not set to a specific value (when it was left as Null). In Sybase or SQL Server, this was not the case – the query would find the rows where `X` was set to a Null value. I see this on almost every conversion from Sybase or SQL Server to Oracle. SQL is supposed to operate under tri-valued logic and Oracle implements Null values the way ANSI SQL requires them to be implemented. Under those rules, comparing `X` to a Null is neither True or False – it is, in fact, *unknown*. The following snippet shows what I mean:

```
ops$tkyte@ORA8I.WORLD> select * from dual;

D
-
X

ops$tkyte@ORA8I.WORLD> select * from dual where null=null;

no rows selected

ops$tkyte@ORA8I.WORLD> select * from dual where null<>null;

no rows selected
```

This can be confusing the first time you see it – it proves that, in Oracle, Null is neither equal to nor not equal to Null. SQL Server, by default, does not do it that way: in SQL Server and sybase, Null is equal to Null. Neither Oracle's, sybase nor SQL Server's SQL processing is *wrong* – they are just *different*. Both databases are in fact ANSI compliant databases but they still work differently. There are ambiguities, backward compatibility issues, and so on, to be overcome. For example, SQL Server supports the ANSI method of

Null comparison, just not by default (it would break thousands of existing legacy applications built on that database).

In this case, one solution to the problem was to write the query like this instead:

```
select *
  from t
  where ( x = l_some_variable OR (x is null and l_some_variable is NULL ))
```

However, this leads to another problem. In SQL Server, this query would have used an index on `x`. This is not the case in Oracle since a B*Tree index (more on indexing techniques in Chapter 7) will not index an entirely Null entry. Hence, if you need to find Null values, B*Tree indexes are not very useful.

What we did in this case, in order to minimize impact on the code, was to assign `X` some value that it could never in reality assume. Here, `X`, by definition, was a positive number – so we chose the number `-1`. Thus, the query became:

```
select * from t where nvl(x,-1) = nvl(l_some_variable,-1)
```

And we created a function-based index:

```
create index t_idx on t( nvl(x,-1) );
```

With minimal change, we achieved the same end result. The important points to recognize from this are that:

- Databases are different. Experience in one will in part carry over to another but you must be ready for some *fundamental* differences as well as some very minor differences.
- Minor differences (such as treatment of Nulls) can have as big an impact as fundamental differences (such as concurrency control mechanism).
- Being aware of the database and how it works and how its features are implemented is the only way to overcome these issues.

Developers frequently ask me (usually more than once a day) how to do something specific in the database. For example, they will ask the question 'How do I create a temporary table in a stored procedure?' I do not answer such questions with a direct answer – I always respond with a question: 'Why do you want to do that?. Many times, the answer will come back: 'In SQL Server we created temporary tables in our stored procedures and we need to do this in Oracle.' That is what I expected to hear. My response, then, is easy – 'you do not want to create temporary tables in a stored procedure in Oracle (you only think you do).' That would, in fact, be a very bad thing to do in Oracle. If you created the tables in a stored procedure in Oracle you would find that:

- Doing DDL is a scalability inhibitor.
- Doing DDL constantly is not fast.
- Doing DDL commits your transaction.
- You would have to use Dynamic SQL in all of your stored procedures in order to access this table – no static SQL.
- Dynamic SQL in PL/SQL is not as fast or as optimized as static SQL.

The bottom line is that you don't want to do it exactly as you did it in SQL Server (if you even need the temporary table in Oracle at all). You want to do things as they are best done in Oracle. Just as if you were going the other way from Oracle to SQL Server, you would not want to create a single table for all users to share for temporary data (that is how Oracle does it). That would limit scalability and concurrency in those other databases. All databases are not created equal – they are all very different.

The Impact of Standards

If all databases are SQL92-compliant, then they must be the same. At least that is the assumption made many times. In this section I would like to dispel that myth.

SQL92 is an ANSI/ISO standard for databases. It is the successor to the SQL89 ANSI/ISO standard. It defines a language (SQL) and behavior (transactions, isolation levels, and so on) that tell you how a database will behave. Did you know that many commercially available databases are SQL92-compliant? Did you know that it means very little as far as query and application portability goes?

Starting with the standard, we will find that the SQL92 standard has four levels:

- **Entry-level** – This is the level to which most vendors have complied. This level is a minor enhancement of the predecessor standard, SQL89. No database vendors have been certified higher and in fact the National Institute of Standards and Technology (NIST), the agency that used to certify for SQL-compliance, does not even certify anymore. I was part of the team that got Oracle 7.0 NIST-certified for SQL92 entry-level compliance in 1993. An entry level compliant database has the feature set of Oracle 7.0.
- **Transitional** – This is approximately 'halfway' between entry-level and intermediate-level as far as a feature set goes.
- **Intermediate** – this adds many features including (not by any means an exhaustive list):

  Dynamic SQL

  Cascade `DELETE` for referential integrity

  `DATE` and `TIME` data types

  Domains

  Variable length character strings

  A `CASE` expression

  `CAST` functions between data types
- **Full** – Adds provisions for (again, not exhaustive):

  Connection management

  A `BIT` string data type

  Deferrable integrity constraints

  Derived tables in the `FROM` clause

  Subqueries in `CHECK` clauses

  Temporary tables

The entry-level standard does not include features such as outer joins, the new inner join syntax, and so on. Transitional does specify outer join syntax and inner join syntax. Intermediate adds more, and Full is, of course all of SQL92. Most books on SQL92 do not differentiate between the various levels leading to confusion on the subject. They demonstrate what a theoretical database implementing SQL92 FULL would look like. It makes it impossible to pick up a SQL92 book, and apply what you see in the book to just any SQL92 database. For example, in SQL Server the 'inner join' syntax is supported in SQL statements, whereas in Oracle it is not. But, they are both SQL92-compliant databases. You can do inner joins and outer joins in Oracle, you will just do it differently than in SQL Server. The bottom line is that SQL92 will not go very far at the entry-level and, if you use any of the features of intermediate or higher, you risk not being able to 'port' your application.

You should not be afraid to make use of vendor-specific features – after all, you are paying a lot of money for them. Every database has its own bag of tricks, and we can always find a way to perform the operation in each database. Use what is best for your current database, and re-implement components as you go to other databases. Use good programming techniques to isolate yourself from these changes. The same techniques are employed by people writing OS-portable applications. The goal is to fully utilize the facilities available to you, but ensure you can change the implementation on a case-by-case basis.

For example, a common function of many database applications is the generation of a unique key for each row. When you insert the row, the system should automatically generate a key for you. Oracle has implemented the database object called a `SEQUENCE` for this. Informix has a `SERIAL` data type. Sybase and SQL Server have an `IDENTITY` type. Each database has a way to do this. However, the methods are different, both in how you do it, and the possible outcomes. So, to the knowledgeable developer, there are two paths that can be pursued:

- Develop a totally database-independent method of generating a unique key.
- Accommodate the different implementations and use different techniques when implementing keys in each database.

The theoretical advantage of the first approach is that to move from database to database you need not change anything. I call it a 'theoretical' advantage because the 'con' side of this implementation is so huge that it makes this solution totally infeasible. What you would have to do to develop a totally database-independent process is to create a table such as:

```
create table id_table ( id_name varchar(30), id_value number );
insert into id_table values ( 'MY_KEY', 0 );
```

Then, in order to get a new key, you would have to execute the following code:

```
update id_table set id_value = id_value + 1 where id_name = 'MY_KEY';
select id_value from id_table where id_name = 'MY_KEY';
```

Looks simple enough, but the outcome is that only one user at a time may process a transaction now. We need to update that row to increment a counter, and this will cause our program to serialize on that operation. At best, one person at a time will generate a new value for this key. This issue is compounded by the fact that our transaction is much larger then we have outlined above. The `UPDATE` and `SELECT` we have in the example are only two statements of potentially many other statements that make up our transaction. We have yet to insert the row into the table with this key we just generated, and do whatever other work it takes to complete this transaction. This serialization will be a huge limiting factor

in scaling. Think of the ramifications if this technique was used on web sites that processed orders, and this was how we generated order numbers. There would be no multi-user concurrency, so we would be forced to do everything sequentially.

The correct approach to this problem would be to use the best code for each database. In Oracle this would be (assuming the table that needs the generated primary key is `T`):

```
create table t ( pk number primary key, ... );
create sequence t_seq;
create trigger t_trigger before insert on t for each row
begin
   select t_seq.nextval into :new.pk from dual;
end;
```

This will have the effect of automatically, and transparently, assigning a unique key to each row inserted. The same effect can be achieved in the other databases using their types – the `create tables` syntax will be different, the net results will be the same. Here, we have gone out of our way to use each databases feature to generate a *non-blocking*, highly concurrent unique key, and have introduced no real changes to the application code – all of the logic is contained in this case in the DDL.

Another example of defensive programming to allow for portability is, once you understand that each database *will implement features in a different way*, to layer your access to the database when necessary. Let's say you are programming using JDBC. If all you use is straight SQL `SELECT`s, `INSERT`s, `UPDATE`s, and `DELETE`s, you probably do not need a layer of abstraction. You may very well be able to code the SQL directly in your application, as long as you limit the constructs you use to those constructs supported by each of the databases you intend to support. Another approach that is both more portable and offers better performance, would be to use stored procedures to return resultsets. You will discover that every vendor's database can return resultsets from stored procedures but how they are returned is different. The actual source code you must write is different for different databases.

Your two choices here would be to either not use stored procedures to return resultsets, or to implement different code for different databases. I would definitely follow the 'different code for different vendors' method, and use stored procedures heavily. This apparently seems to increase the amount of time it would take to implement on a different database. However, you will find it is actually easier to implement on multiple databases with this approach. Instead of having to find the perfect SQL that works on *all* databases (perhaps better on some than on others), you will implement the SQL that works best on that database. You can do this outside of the application itself, giving you more flexibility in tuning the application. We can fix a poorly performing query in the database itself, and deploy that fix immediately, without having to patch the application. Additionally, you can take advantage of vendor extensions to SQL using this method freely. For example, Oracle supports hierarchical queries via the `CONNECT BY` operation in its SQL. This unique feature is great for resolving recursive queries. In Oracle you are free to utilize this extension to SQL since it is 'outside' of the application (hidden in the database). In other databases, you would use a temporary table and procedural code in a stored procedure to achieve the same results, perhaps. You paid for these features so you might as well use them.

These are the same techniques developers who implement multi-platform code utilize. Oracle Corporation for example uses this technique in the development of its own database. There is a large amount of code (a small percentage of the database code overall) called **OSD** (**O**perating **S**ystem **D**ependent) code that is implemented specifically for each platform. Using this layer of abstraction, Oracle is able to make use of many native OS features for performance and integration, without having to rewrite the large majority of the database itself. The fact that Oracle can run as a multi-threaded

application on Windows and a multi-process application on UNIX attests to this feature. The mechanisms for inter-process communication are abstracted to such a level that they can be re-implemented on an OS-by-OS basis, allowing for radically different implementations that perform as well as an application written directly, and specifically, for that platform.

In addition to SQL syntactic differences, implementation differences, and differences in performance of the same query in different databases outlined above, there are the issues of concurrency controls, isolation levels, query consistency, and so on. We cover these items in some detail in Chapter 3, *Locking and Concurrency*, and Chapter 4, *Transactions* of this book, and see how their differences may affect you. SQL92 attempted to give a straightforward definition of how a transaction should work, how isolation levels are to be implemented, but in the end, you'll get different results from different databases. It is all due to the implementation. In one database an application will deadlock and block all over the place. In another database, the same exact application will not – it will run smoothly. In one database, the fact that you did block (physically serialize) was used to your advantage and when you go to deploy on another database, and it does not block, you get the wrong answer. Picking an application up and dropping it on another database takes a lot of hard work and effort, even if you followed the standard 100 percent.

Features and Functions

A natural extension of the argument that you shouldn't necessarily strive for 'database independence' is the idea that you should understand exactly what your specific database has to offer and make full use of it. This is not a section on all of the features that Oracle 8i has to offer. That would be an extremely large book in itself. The new features of Oracle 8i themselves fill a book in the Oracle documentation set. With about 10,000 pages of documentation provided by Oracle, covering each and every feature and function would be quite an undertaking. Rather, this is a section on why it would benefit you to get at least a cursory knowledge of what is provided.

As I've said before, I answer questions about Oracle on the web. I'd say that 80 percent of my answers are simply URLs to the documentation. People are asking how they might go about writing some complex piece of functionality in the database (or outside of it). I just point them to the place in the documentation that tells them how Oracle has already implemented it, and how to use it. Replication comes up this way frequently. I'll receive the question 'I would like to keep a copy of my data elsewhere. I would like this to be a read-only copy. I need it to update only once a day at midnight. How can I write the code to do that?' The answer is as simple as a `CREATE SNAPSHOT` command. This is what built-in functionality in the database.

It is true you can write your own replication, it might even be fun to do so, but at the end of the day, it would not be the smartest thing to do. The database does a lot of stuff. In general, it can do it better then we can ourselves. Replication for example is internalized in the kernel, written in C. It's fast, it's fairly easy, and it is robust. It works across versions, across platforms. It is supported, so if you hit a problem, Oracle's support team will be glad to help. If you upgrade, replication will be supported there as well, probably with some new features. Now, consider if you had developed your own. You would have to provide support for all of the versions you wanted to support. Inter-operability between 7.3 and 8.0 and 8.1 and 9.0, and so on – this would be your job. If it 'broke', you won't be calling support. At least, not until you can get a test case that is small enough to demonstrate your basic issue. When the new release of Oracle comes out, it will be up to you to migrate your replication code to that release.

Not having a full understanding of what is available to you can come back to haunt you in the long run. I was recently talking with some developers and their management. They were demonstrating a 'very cool' piece of software they had developed. It was a message-based system that solved the database queue problem. You see this normally in a database if you wanted many people to use a table as a 'queue'. You would like many people to be able to lock the next record in the queue,

skipping over any previously locked records (these queue records are being processed already). The problem you encounter is that there is no documented syntax in the database for skipping locked rows. So, if you didn't know anything about Oracle's features, you would assume that if you wanted queuing software on top of the database, you would have to build it (or buy it).

That is what these developers did. They built a series of processes, and developed APIs for doing message queuing on top of the database. They spent quite a bit of time on it, and used quite a few man-hours to achieve it. The developers were quite sure it was unique. Immediately after seeing it, and hearing of its functionality, I had one thing to say – Advanced Queues. This is a native feature of the database. It solves the 'get the first unlocked record in the queue table and lock it for me' problem. It was right there all along. Their developers, not knowing that this feature existed, spent a lot of time and energy writing their own. In addition to spending lots of time in the past on it, they would be spending lots of time maintaining it in the future. Their manager was less than impressed upon discovering the unique piece of software in effect emulated a native database feature.

I have seen people in an Oracle 8i database set up daemon processes that reads messages off of pipes (a database IPC mechanism). These daemon processes execute the SQL contained within the pipe message, and commit the work. They did this so that they could execute auditing in a transaction that would not get rolled back if the bigger transaction did. Usually, if a trigger or something were used to audit an access to some data, but a statement failed later on, all of the work would be rolled back (see Chapter 4 on *Transactions*, we discuss this statement level atomicity in some detail). So, by sending a message to another process, they could have a separate transaction do the work and commit it. The audit record would stay around, even if the parent transaction rolled back. In versions of Oracle before Oracle 8I, this was an appropriate (and pretty much the only) way to implement this functionality. When I told them of the database feature called autonomous transactions (we will take a detailed look at these in Chapter 15), they were quite upset with themselves. Autonomous transactions, implemented with a single line of code, do exactly what they were doing. On the bright side, this meant they could discard a lot of code and not have to maintain it. In addition, the system ran faster overall, and was easier to understand. Still, they were upset at the amount of time they had wasted reinventing the wheel. In particular the developer who wrote the daemon processes was quite upset at having just written a bunch of 'shelf-ware'.

The above list of examples is something I see repeated time, and time again – large complex solutions to problems that are already solved by the database itself. Unless you take the time to learn what is available, you are doomed to do the same thing at some point. In the second section of this book, *Database Structures and Utilities*, we are going to take an in-depth look at a *handful* of functionality provided by the database. I picked and chose the features and functions that I see people using frequently, or in other cases, functionality that should be used more often but is not. It is only the tip of the iceberg however. There is so much more to Oracle than can be presented in a single book.

Solving Problems Simply

There are always two ways to solve everything: the easy way and the hard way. Time and time again, I see people choosing the hard way. It is not always done consciously. More usually, it is done out of ignorance. They never expected the database to be able to do 'that'. I, on the other hand, expect the database to be capable of anything and only do it the 'hard' way (by writing it myself) when I discover it cannot do something.

For example, I am frequently asked 'How can I make sure the end user has only one session in the database?' (There are hundreds of other examples I could have used here). This must be a requirement of many applications but none that I've ever worked on – I've not found a good reason for limiting people in this way. However, people want to do it and when they do, they usually do it the hard way. For example, they will have a batch job run by the operating system that will look at the `V$SESSION`

table and arbitrarily kill sessions of users who have more then one session. Alternatively, they will create their own tables and have the application insert a row when a user logs in, and remove the row when they log out. This implementation invariably leads to lots of calls to the help desk because when the application 'crashes', the row never gets removed. I've seen lots of other 'creative' ways to do this, but none is as easy as:

```
ops$tkyte@ORA8I.WORLD> create profile one_session limit sessions_per_user 1;
Profile created.

ops$tkyte@ORA8I.WORLD> alter user scott profile one_session;
User altered.

ops$tkyte@ORA8I.WORLD> alter system set resource_limit=true;
System altered.
```

That's it – now any user with the `ONE_SESSION` profile can log on only once. When I bring up this solution, I can usually hear the smacking of a hand on the forehead followed by the statement 'I never knew it could do that'. Taking the time to familiarize yourself with what the tools you have to work with are capable of doing can save you lots of time and energy in your development efforts.

The same 'keep in simple' argument applies at the broader architecture level. I would urge people to think carefully before adopting very complex implementations. The more moving parts you have in your system, the more things you have that can go wrong and tracking down exactly where that error is occurring in an overly complex architecture is not easy. It may be really 'cool' to implement using umpteen tiers, but it is not the right choice if a simple stored procedure can do it better, faster and with less resources.

I've worked on a project where the application development had been on going for over a year. This was a web application, to be rolled out to the entire company. The HTML client talked to JSPs in the middle tier, which talked to CORBA objects, which talked to the database. The CORBA objects would maintain 'state' and a connection to the database in order to maintain a session. During the testing of this system we found that they would need many front end application servers and a very large database machine to support the estimated 10,000 concurrent users. Not only that, but stability was an issue at times given the complex nature of the interaction between the various components (just exactly where in that stack is the error coming from and why? – that was a hard question to answer). The system would scale, it would just take a lot of horsepower to do it. Additionally, since the implementation used a lot of complex technologies – it would require experienced developers to not only to develop it but to maintain it. We took a look at that system and what it was trying to do and realized that the architecture was a little more complex than it needed to be in order to do the job. We saw that simply by using the PL/SQL module of Oracle iAS and some stored procedures, we could implement the exact system on a fraction of the hardware, and using less 'experienced' developers. No EJBs, no complex interaction between JSPs and EJBs – just the simple translation of a URL into a stored procedure call. This new system is still up and running today, exceeding the estimated user count and with response times that people do not believe. It uses the most basic of architectures, has the fewest moving pieces, runs on an inexpensive 4-CPU workgroup server and never breaks (well a tablespace filled up once, but that's another issue).

I will always go with the simplest architecture that solves the problem completely over a complex one any day. The payback can be enormous. Every technology has its place – not every problem is a nail, we can use more than a hammer in our toolbox.

Openness

There is another reason that I frequently see people doing things the hard way and again it relates to the idea that one should strive for 'openness' and 'database independence' at all costs. The developers wish to avoid using 'closed', 'proprietary' database features – even something as simple as 'stored procedures' or 'sequences' because that will lock them into a database system. Well, let me put forth the idea that the instant you develop a read/write application you are already somewhat locked in. You will find subtle (and sometimes not so subtle) differences between the databases as soon as you start running queries and modifications. For example, in one database you might find that your `SELECT COUNT(*) FROM T` deadlocks with a simple update of two rows. In Oracle, you'll find that the `SELECT COUNT(*)` never blocks for a writer. We've seen the case where a business rule appears to get enforced on one database, due to side effects of the database's
s locking model, and does not get enforced in another database. You'll find that, given the same exact transaction mix, reports come out with different answers in different databases – all because of fundamental implementation differences. You will find that it is a very rare application that can simply be picked up and moved from one database to another. Differences in the way the SQL is interpreted (for example, the `NULL=NULL` example) and processed will always be there.

On a recent project, the developers were building a web-based product using Visual Basic, ActiveX Controls, IIS Server, and the Oracle 8i database. I was told that the development folks had expressed concern that since the business logic had been written in PL/SQL, the product had become database dependent and was asked: 'How can we correct this?'

I was a little taken aback by this question. In looking at the list of chosen technologies I could not figure out how being database dependent was a 'bad' thing:

- They had chosen a language that locked them into a single operating system and is supplied by a single vendor (they could have opted for Java).
- They had chosen a component technology that locked them into a single operating system and vendor (they could have opted for EJB or CORBA).
- They had chosen a web server that locked them in to a single vendor and single platform (why not Apache?).

Every other technology choice they had made locked them into a very specific configuration – in fact the only technology that offered them any choice as far as operating systems go was in fact the database.

Regardless of this – they must have had good reasons to choose the technologies they did – we still have a group of developers making a conscious decision to not utilize the functionality of a critical component in their architecture, and doing it in the name of 'openness'. It is my belief that you pick your technologies carefully and then you exploit them to the fullest possible extent. You have paid a lot for these technologies – would it not be in your best interest to exploit them fully? I had to assume that they were looking forward to utilizing the full potential of the other technologies – so why was the database an exception? An even harder question to answer in light of the fact that it was crucial to their success.

We can put a slightly different spin on this argument if we consider it from the perspective of 'openness'. You put all of your data into the database. The database is a very open tool. It supports data access via SQL, EJBs, HTTP, FTP, SMB, and many other protocols and access mechanisms. Sounds great so far, the most open thing in the world.

Then, you put all of your application logic and more importantly, your *security* outside of the database. Perhaps in your beans that access the data. Perhaps in the JSPs that access the data. Perhaps in your Visual Basic code running under Microsoft's Transaction Server (MTS). The end result is that you have just closed off your database – you have made it 'non-open'. No longer can people hook in existing technologies to make use of this data – they *must* use your access methods (or bypass security altogether). This sounds all well and fine today, but what you must remember is that the 'whiz bang' technology of today, EJBs for example, yesterday's concept, and tomorrow's old, tired technology. What has persevered for over 20 years in the relational world (and probably most of the object implementations as well) is the database itself. The front ends to the data change almost yearly, and as they do, the applications that have all of the security built inside themselves, not in the database, become obstacles, roadblocks to future progress.

The Oracle database provides a feature called **Fine Grained Access Control** (Chapter 21 is dedicated to it). In a nutshell, this technology allows the developer to embed procedures in the database that can modify queries as they are submitted to the database. This query modification is used to restrict the rows the client will receive or modify. The procedure can look at who is running the query, when they are running the query, what terminal they are running the query from, and so on, and can constrain access to the data as appropriate. With FGAC, we can enforce security such that, for example:

- Any query executed outside of normal business hours by a certain class of users returned zero records.
- Any data could be returned to a terminal in a secure facility but only non-sensitive information to a 'remote' client terminal.

Basically, it allows us to locate access control in the database, *right next to the data*. It no longer matters if the user comes at the data from a Bean, a JSP, a VB application using ODBC, or SQL*PLUS, the same security protocols will be enforced. You are well situated for the next technology that comes along.

Now, I ask you – which implementation is more 'open'? The one that makes all access to the data possible only through calls to the VB code and ActiveX controls (replace VB with Java and ActiveX with EJB if you like – I'm not picking on a particular technology but an implementation here) or the solution that allows access from anything that can talk to the database, over protocols as diverse as SSL, HTTP and Net8 (and others) or using APIs such as ODBC, JDBC, OCI, and so on? I have yet to see an ad-hoc reporting tool that will 'query' your VB code. I know of dozens that can do SQL, though.

The decision to strive for database independence and total 'openness' is one that people are absolutely free to take, and many try, but I believe that it is the wrong decision. No matter what database you are using, you should exploit it fully, squeezing every last bit of functionality you can out of that product. You'll find yourself doing that in the tuning phase (which again always seems to happen right after deployment) anyway. It is amazing how quickly the database independence requirement can be dropped when you can make the application run five times faster just by exploiting the software's capabilities.

How Do I Make it Run Faster?

The question in the heading is one I get asked all the time. Everyone is looking for the fast = true switch, assuming 'database tuning' means that you tune the database. In fact, it is my experience that more than 80 percent (frequently much more, 100 percent) of all performance gains are to be realized at the application level – not the database level. You cannot tune a database until you have tuned the applications that run on the data.

As time goes on there are some switches we can 'throw' at the database level to help lessen the impact of egregious programming blunders. For example, Oracle 8.1.6 adds a new parameter, CURSOR_SHARING=FORCE. This feature implements an 'auto binder' if you will. It will silently take a query written as SELECT * FROM EMP WHERE EMPNO = 1234 and rewrite it for us as SELECT * FROM EMP WHERE EMPNO = :x. This *can* dramatically decrease the number of hard parses, and decrease the library latch waits we discussed in the Architecture sections – *but* (there is always a but) it can have some side effects. You may hit an issue (a.k.a. 'bug') with regards to this feature, for example in the first release:

```
ops$tkyte@ORA8I.WORLD> alter session set cursor_sharing=force;
Session altered.

ops$tkyte@ORA8I.WORLD> select * from dual where dummy='X'and 1=0;
select * from dual where dummy='X'and 1=0
                                     *
ERROR at line 1:
ORA-00933: SQL command not properly ended

ops$tkyte@ORA8I.WORLD> alter session set cursor_sharing=exact;
Session altered.

ops$tkyte@ORA8I.WORLD> select * from dual where dummy='X'and 1=0;
no rows selected
```

The way they rewrote the query (because of the lack of whitespace between 'X' and the word AND) didn't work in 8.1.6. The query ended up being:

```
select * from dual where dummy=:SYS_B_0and :SYS_B_1=:SYS_B_2;
```

The key word AND became part of the bind variable :SYS_B_0. In 8.1.7, however, this query is rewritten as:

```
select * from dual where dummy=:"SYS_B_0"and :"SYS_B_1"=:"SYS_B_2";
```

This works *syntactically* but might negatively affect your program's performance. For example, in the above, notice how 1=0 (also False) is rewritten to be :"SYS_B_1" = :"SYS_B_2". The optimizer no longer has all of the information at parse time, it can no longer see that this query returns zero rows (before it even executes it). While I don't expect you to have lots of queries with 1=0 in them, I would expect you to have some queries that do use literals in them *on purpose*. You may have a column with very skewed values in it, for example 90 percent of the values of the column are greater than 100, 10 percent are less then 100. Further, 1 percent is less then 50. You would want the query:

```
select * from t where x < 50;
```

to use an index, and the query:

```
select * from t where x > 100;
```

to *not* use an index. If you use CURSOR_SHARING = FORCE, the optimizer will not have the 50 or 100 values to consider when optimizing – hence it will come up with a generic plan that probably does not use the index (even if 99.9 percent of your queries are of the type WHERE x < 50).

Additionally, I have found that while CURSOR_SHARING = FORCE runs much faster than parsing and optimizing lots of unique queries, I have also found it to be slower than using queries where the developer did the binding. This arises not from any inefficiency in the cursor sharing code, but rather in inefficiencies in the program itself. In Chapter 10, *Tuning Strategies and Tools*, we'll discover how parsing of SQL queries can affect our performance. In many cases, an application that does not use bind variables is not efficiently parsing and reusing cursors either. Since the application believes each query is unique (it built them as unique statements) it will never use a cursor more than once. The fact is that if the programmer had used bind variables in the first place, they could have parsed a query once and reused it many times. It is this overhead of parsing that decreases the overall potential performance you could see.

Basically, it is important to keep in mind that simply turning on CURSOR_SHARING = FORCE will not necessarily fix your problems. It may very well introduce new ones. CURSOR_SHARING is, in some cases, a very useful tool, but it is not a silver bullet. A well-developed application would never need it. In the long term, using bind variables where appropriate, and constants when needed, is the correct approach.

Even if there are some switches that can be thrown at the database level, and they are truly few and far between, problems relating to concurrency issues and poorly executing queries (due to poorly written queries or poorly structured data) cannot be fixed with a switch. These situations require rewrites (and frequently a re-architecture). Moving datafiles around, changing the multi-block read count, and other 'database' level switches frequently have a minor impact on the overall performance of an application. Definitely not anywhere near the 2, 3, ... N times increase in performance you need to achieve to make the application acceptable. How many times has your application been 10 percent too slow? 10 percent too slow, no one complains about. Five times too slow, people get upset. I repeat: you will not get a 5-times increase in performance by moving datafiles around. You will only achieve this by fixing the application – perhaps by making it do significantly less I/O.

Performance is something you have to design for, to build to, and to test for continuously throughout the development phase. It should never be something to be considered after the fact. I am amazed at how many times people wait until the application has been shipped to their customer, put in place and is actually running before they even start to tune it. I've seen implementations where applications are shipped with nothing more than primary keys – no other indexes whatsoever. The queries have never been tuned or stress tested. The application has never been tried out with more than a handful of users. Tuning is considered to be part of the installation of the product. To me, that is an unacceptable approach. Your end users should be presented with a responsive, fully tuned system from day one. There will be enough 'product issues' to deal with without having poor performance be the first thing they experience. Users are expecting a few 'bugs' from a new application, but at least don't make them wait a painfully long time for them to appear on screen.

The DBA-Developer Relationship

The back cover of this book talks of the importance of a DBA knowing what the developers are trying to accomplish and of developers knowing how to exploit the DBA's data management strategies. It's certainly true that the most successful information systems are based on a symbiotic relationship between the DBA and the application developer. In this section I just want to give a developer's perspective on the division of work between developer and DBA (assuming that every serious development effort has a DBA team).

As a developer, you should not necessarily have to know how to install and configure the software. That should be the role of the DBA and perhaps the SA (System Administrator). Setting up Net8, getting the listener going, configuring MTS, enabling connection pooling, installing the database, creating the database, and so on – these are functions I place in the hands of the DBA/SA.

In general, a developer should not have to know how to tune the operating system. I myself generally leave this task to the SAs for the system. As a software developer for database applications you will need to be competent in use of your operating system of choice, but you shouldn't expect to have to tune it.

Perhaps one of the biggest concerns of the DBA is how to back up and restore a database, and I would say that this is the sole responsibility of the DBA. Understanding how rollback and redo work – yes, that is something a developer has to know. Knowing how to perform a tablespace point in time recovery is something a developer can skip over. Knowing that you can do it might come in handy, but actually having to do it – no.

Tuning at the database instance level, figuring out what the optimum `SORT_AREA_SIZE` should be – that's typically the job of the DBA. There are exceptional cases where a developer might need to change some setting for a session, but at the database level, the DBA is responsible for that. A typical database supports more than just a single developer's application. Only the DBA who supports all of the applications can make the right decision.

Allocating space and managing the files is the job of the DBA. Developers will contribute their estimations for space (how much they feel they will need) but the DBA/SA will take care of the rest.

Basically, developers do not need to know how to run the database. They need to know how to run *in* the database. The developer and the DBA will work together on different pieces of the same puzzle. The DBA will be visiting you, the developer, when your queries are consuming too many resources, and you will be visiting them when you cannot figure out how to make the system go any faster (that's when instance tuning can be done, when the application is fully tuned).

This will all vary by environment, but I would like to think that there is a division. A good developer is usually a very bad DBA, and vice versa. They are two different skillsets, two different mindsets, and two different personalities in my opinion.

Summary

Here we have taken a somewhat anecdotal look at why you need to know the database. The examples I have given are not isolated – they happen every day, day in and day out. I observe a continuous cycle of this happening over and over again and again. Let's quickly recap the key points. If you are developing with Oracle:

- You need to understand the Oracle architecture. You don't have to know it so well that you are able to rewrite the server if you wanted but you should know it well enough that you are aware of the implications of using a particular feature.
- You need to understand locking and concurrency control and that every database implements this differently. If you don't, your database will give 'wrong' answers and you will have large contention issues – leading to poor performance.
- Do not treat the database as a black box, something you need not understand. The database is the most critical piece of most applications. To try to ignore it would be fatal.
- Do not re-invent the wheel. I've seen more then one development team get in trouble, not only technically but on a personal level, due to a lack of awareness what Oracle provides for free. This will happen when it is pointed out that the feature they just spent the last couple of months implementing was actually a core feature of the database all along.

- Solve problems as simply as possible, using as much of Oracle's built-in functionality as possible. You paid a lot for it.
- Software projects come and go, programming languages and frameworks come and go. We developers are expected to have systems up and running in weeks, maybe months, and then move on to the next problem. If we re-invent the wheel over and over, we will never come close to keeping up with the frantic pace of development. Just as you would never build your own hash table class in Java – since it comes with one – you should use the database functionality you have at your disposal. The first step to being able to do that, of course, is to understand what it is you have at your disposal. Read on.