CHAPTER 1

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 eighteen years, I've worked on many projects – successful ones as well as complete 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. As a corollary to this – all applications are built around databases, I cannot think of a single useful application that doesn't store data persistently somewhere.
* Applications come, applications go. The **data** however lives forever. It is not about building applications, it really is about the data underneath these applications.
* 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 built to perform from day one. Tuning after the fact – tuning after deployment means you did not

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 the database 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 necessarily 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 than to build your own.
* Why you might want more than a cursory knowledge of SQL.
* That the DBA and Developer staff are team members together, not enemy camps trying to outsmart each other at every turn.

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, Windows versus Linux. Both are operating systems. Each provides largely the same set of services to developers, such as file management, memory management, process management, security and so on. However, they are very different architecturally and if you are a long time Windows programmer asked to develop a new application on the Linux platform then you would have to relearn a couple of things. Memory management is done differently. Building a server process is considerably different. Under Windows, you would develop a single process, a single executable, with many threads. Under Linux, you would not develop a single standalone executable instead you would have many processes working together. . In summary, much of what you learned in the Windows environment won't apply to Linux (and vice versa to be fair). You have to unlearn to be successful. 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, then 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. And to be fair, the opposite is true – a scalable, well developed Oracle application will not necessarily run on SQL Server "as is" without major architectural changes. Just as Windows and Linux are both operating systems, but fundamentally different, Oracle and SQL Server (pretty much any database could be noted there) are both databases, but fundamentally different.

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/Linux 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 Oracle9*i*, Oracle10*g* under Linux or Windows XP under vmware. I am able to then deploy them on a variety of servers running the same database software but different operating systems. 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 the language 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, one that has been consistent for many years:

* 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 (as little PL/SQL as possible!).
* If you cannot do it in PL/SQL, try a Java Stored Procedure. The number of times this is necessary is extremely rare today with Oracle9i and above.
* If you cannot do it in Java, do it in a C external procedure. This is most frequently the approach when raw speed, or the use of a 3rd party API written in C is needed.
* 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 eighteen years of tuning has gone into it – in fact in Oracle10*g* the compiler itself was rewritten to be an optimizing compiler for the first time. You will find no other language so tightly coupled with SQL, nor any as optimized to interact with SQL. Working with SQL in PL/SQL is a very natural thing – where as in virtually every other language from Visual Basic to Java, using SQL can feel cumbersome – it never quite feels "natural", it is not an extension of the language itself. When PL/SQL runs out of steam – which is exceedingly rare today with current database releases – 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 in many cases.

The Black Box Approach

I have an idea, borne out by first-hand personal experience (meaning, I made the mistake myself), 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 than 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 completely 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 would also have a mandate to be 'database independent'. A mandate they could not hope to follow for many reasons, the most obvious one is they didn't know enough about what databases are, what they do to even find the lowest common denominator amongst them.
* 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 am now called in to help solve the difficulties (in the past, I was the cause of such issues). 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.

Even today in 2005, I find many times the developers of database applications have not spent any time reading the documentation. On my web site asktom.oracle.com – I frequently get questions along the lines of "what is the syntax for …" coupled with "we don't have the documentation so please just tell us". I refuse to directly answer many of those questions, but rather point them to the online documentation – freely available to anyone, anywhere in the world. In the last 10 years – the excuse of "we don't have documentation", "we don't have access to resources" has virtually disappeared. The introduction of the 'web' and sites like otn.oracle.com (the Oracle Technology Network) and groups.google.com (the internet usenet discussion forums) makes it inexcusable to not have a full set of documentation at your finger tips!

The very 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 as easily as 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 (and probably some seriously hurt fingers). Similar effects can be achieved by remaining ignorant of your database.

I was called into a project recently. The developers where experiencing massive performance issues – it seemed their system was serializing many transactions, that is – instead of many people working concurrently, everyone was getting into a really long line and waiting for everyone in front of them to complete. The application architects walked me through the architecture of their system – the classic 3 tier approach. They would have a web browser talk to a middle tier application server running Java Server Pages (JSP). The JSP's would in turn utilize another layer – Enterprise Java Beans (EJB's) that did all of the SQL. The SQL in the EJB's was generated by some 3rd party tool and was done in a database independent fashion.

Now, in this system – it was very hard to diagnose anything, for none of the code was instrumented or traceable. Instrumented code is the fine art of making every other line of developed code be debug code of some sort – so when you are faced with performance or capacity or even logic issues, you can track down exactly where the problem is. In this case – we could only limit the space of the problem to be "somewhere in between the browser and the database" – in other words, the entire system was suspect. Fortunately the Oracle database is heavily instrumented, unfortunately the application needs to be able to turn the instrumentation on and off at appropriate points – something it was not designed to do.

So, we were faced with trying to diagnose a performance issue with not too many details – just what we could glean from the database itself. Fortunately in this case – it was fairly easy. When someone who knew the Oracle V$ tables (the V$ tables are one way Oracle exposes its instrumentation, it's statistics to us) reviewed them, we could see the major contention was around a single table – a queue table of sorts. The application would place records into this table and another set of processes would pull the records out of this table and process them. Digging deeper, we found a bitmap index on a column in this table (see the later chapter on Indexing for more information on bitmapped indexes). The reasoning was that this column, the processed flag column, had only two values – Y and N. As records were inserted – they would have a value of N for not processed. As the other processes read and processed the record, they would update the N to Y to indicate it was done. They needed to find the N records rapidly and hence knew they wanted to index that column. They had read somewhere that bitmap indexes are for low cardinality columns – when the column has but a few distinct values so it seemed a natural fit.

But – that bitmap index was the cause of all of their problems. In a bitmap index, a single key entry points to many rows – hundreds or more of them. If you update a bitmap index key – the hundreds of records that key points to are locked as well. So, someone inserting the new record with 'N' would lock the 'N' record in the bitmap index, effectively locking hundreds of other 'N' records as well. Meanwhile the process trying to read this table and process the records would be prevented from modifying some 'N' record to be a 'Y' (processed) record – because in order for it to update this column from N to Y, it would need to lock that same bitmap index key. In fact other sessions just trying to insert a new record into this table would be blocked as well, as they would be attempting to lock this same bitmap key entry. In short, the developers had created a table that at most one person would be able to insert or update against at a time! We can see this easily using a simple scenario. Here, I will use an autonomous transaction in the database to have two concurrent transactions in a single session:

Tony – would like to point the reader to the old material on Apress that will not be in this edition, how do you want to do that?

```
ops$tkyte@ORA10G> create table t ( processed_flag varchar2(1) );
Table created.
ops$tkyte@ORA10G> create bitmap index t_idx on t(processed_flag);
Index created.
ops$tkyte@ORA10G> insert into t values ( 'N' );
1 row created.
ops$tkyte@ORA10G> declare
  2      pragma autonomous_transaction;
  3  begin
  4      insert into t values ( 'N' );
  5  end;
  6  /
declare
*
ERROR at line 1:
ORA-00060: deadlock detected while waiting for resource
ORA-06512: at line 4
```

Since I used an autonomous transaction, I received a deadlock – meaning my second insert was blocked by my first insert. Had I used two separate sessions, this would not "deadlocked", the second insert would have just blocked and waited for the first transaction to commit or rollback.

So here we had an issue whereby not understanding the database feature (bitmap indexes) and how it worked, what it did – the database was doomed to poor scalability from the start. Fortunately, once this issue was discovered, correcting it was easy. We needed an index on the processed flag column, but not a bitmap index. We needed a conventional B*Tree index here. This took a bit of convincing – no one wanted to believe that conventionally indexing a column with two distinct values was a "good idea". But after setting up a simulation (something I am very much into – simulations, testing, experimenting) – we were able to prove it was not only the correct approach but would work very nicely. There were two ways to approach the indexing of this particular column:

* Just create an index on the processed flag column
* Create an index only on the processed flag column when the processed flag is N, that is – only index the values of interest. We typically do not want to use an index when processed flag = 'Y' since that is the vast majority of the records in the table have the value 'Y'. Notice that I did not say "We never want to use", you might want to very frequently count the number of processed records for some reason – there an index on the processed records might well come in very handy.

In the chapter on Indexing we'll go into more detail on both. In the end, we ended up creating a very small index on just the records where processed flag = N. Access to those records was extremely fast and the vast majority of Y records did not contribute to this index at all (we used a function based index to return either N or NULL – since an entirely NULL key is not placed into a conventional B*Tree index, we ended up only indexing the N records).

Was that the end of the story? No, not at all. They still had a less than optimal solution on their hands. We fixed their major problem, caused by not fully understand the tools they were using – and found only after lots of looking and study since the system was not nicely instrumented. What we didn't fix yet was the fact that

* The application was built without a single consideration for scaling the database level.
* The application itself could not be tuned or touched.
* The application was performing functionality (the queue table) that the database *already supplied in a highly concurrent and scalable fashion.* I'm referring to the Advance Queue (AQ) software that is burned into the database, functionality they were trying to re-invent.
* 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 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 the chapter on *Tuning Strategies and Tools*). Oracle10*g* actually does permit us to accomplish this magic feat for the first time to a large degree.
* 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. 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. I will refer you either to a subsequent chapter in this book or to the relevant Oracle documentation for more information.

Understanding Oracle Architecture

Recently, I was working with a customer running a large production application – an application that had been 'ported' from SQL Server to Oracle. I quote 'ported' simply because most 'ports' I see are 'what is the minimal change we can make to have our SQL Server code compile and execute on Oracle'. The applications that result from that are frankly the ones I see most often because they are the ones that need the most help. I want to make clear however, this is not a bash against SQL Server in this respect – the opposite is true, taking an Oracle application and just plopping it down on top of SQL Server with as few changes as possible results in the same poorly performing code in reverse – this goes both ways.

In this particular case however, the SQL Server architecture and how you use SQL Server really impacted the Oracle implementation. Their stated goal was to scale up – to go larger, but they did not want to really 'port' to another database. They wanted to do it with as little work as humanly possible. So, they kept the architecture basically the same in the client and database layers. The decision had two important ramifications for us as we'll see. They were:

* Keeping the connection architecture to the database the same in Oracle as it was in SQL Server
* Using literal (non-bound) SQL

These two decisions resulted in a system that could not support the required user load (the database server simply ran out of available memory) and even for the set of users that could log in and use the application – abysmal performance.

Use A Single Connection in Oracle

Now, in SQL Server it is a very common practice to open a connection to the database for each concurrent statement you want to execute. If you are going to do five queries, you might well see five connections in SQL Server. In Oracle – if you want to do five queries, or five hundred, the maximum number of connections you want to open is one. So, a common practice in SQL Server is something that is not only not encouraged in Oracle but actively discouraged – having multiple connections to the database is just something you don't want to do.

But did it they did. A simple web based application would open 5, 10, 15 or more connections per web page – meaning that their server could support only 1/5th, 1/10th or 1/15th the number of concurrent users that it should have been able to. Additionally – they were attempting to run the database on the Windows platform itself – just a plain Windows XP server without access to the 'data center' version of Windows. This meant that the Windows architecture of a single process limited the Oracle database server to about 1.75gig of RAM in total. Since each Oracle connection took at least a fixed amount of RAM, their ability to scale was severely limited on this hardware. They had 8gig of RAM on the server, but could only use about 2gig of it.

There are ways to get much more RAM used in a Windows environment – such as with the /AWE switch, but they required versions of the operating system that were not in use in this situation.

There were three approaches to correcting this problem – and all three entailed quite a bit of work, and this was after the 'port' was complete! The options were:

* Re-architect the application, to allow it to take advantage of the fact it was running "on" Oracle and use a single connection to generate a page, not somewhere between 5 to 15 connections. This is the only solution that would actually solve the problem.

* Upgrade the operating system (no small chore) and utilize the larger memory model of the Windows Data Center version – in itself not a small chore either as it involves a rather involved database setup with indirect data buffers and other non-standard settings.
* Migrate the database from a Windows based OS to some other OS where multiple processes are used, effectively allowing the database to utilize all installed RAM.

As you can see – none of these were "ok, we'll do that this afternoon" sort of solutions. Each was a complex solution to a problem that would have most easily been corrected during the database 'port' phase – while you were in the code poking around and changing things in the first place. Additionally, a simple test to "scale" prior to rolling out a production

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 in most cases. 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 the chapter *Locking and Latching*, 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. 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. In this test, we'll just be inserting some rows into a table, the simple table we will use is:

```
ops$tkyte@ORA9IR2> drop table t;
Table dropped.

ops$tkyte@ORA9IR2> create table t ( x int );
Table created.
```

Now, we'll create two very simple stored procedures. They both will insert the numbers 1 through 10,000 into this table however the first procedure will use a single SQL statement with a bind variable:

```
ops$tkyte@ORA9IR2> create or replace procedure proc1
  2  as
  3  begin
  4      for i in 1 .. 10000
  5      loop
  6          execute immediate
  7          'insert into t values ( :x )' using i;
  8      end loop;
  9  end;
 10  /
Procedure created.
```

And the second procedure will construct a unique SQL statement for each and every row to be inserted:

```
ops$tkyte@ORA9IR2> create or replace procedure proc2
  2  as
  3  begin
  4      for i in 1 .. 10000
  5      loop
  6          execute immediate
  7          'insert into t values ( '||i||')';
  8      end loop;
  9  end;
 10  /
Procedure created.
```

Now, the only difference arguably between the two is one uses a bind variable and the other does not. Both are utilizing dynamic SQL, the logic is identical, the only change is the use or not of bind variables. We are ready to compare the two approaches and we'll use 'runstats', a simple tool I've developed, to compare the two in detail:

For details on runstats and other utilities please see the chapter on *Tuning Strategies and Tools*

```
ops$tkyte@ORA9IR2> exec runstats_pkg.rs_start
PL/SQL procedure successfully completed.

ops$tkyte@ORA9IR2> exec proc1
PL/SQL procedure successfully completed.

ops$tkyte@ORA9IR2> exec runstats_pkg.rs_middle
PL/SQL procedure successfully completed.

ops$tkyte@ORA9IR2> exec proc2
PL/SQL procedure successfully completed.

ops$tkyte@ORA9IR2> exec runstats_pkg.rs_stop(1000)
Run1 ran in 159 hsecs
Run2 ran in 516 hsecs
run 1 ran in 30.81% of the time
```

Now, that clearly shows that by the "wall clock", it took significantly longer to insert 10,000 rows without bind variables than it did with them. In fact, it took 3 times longer, meaning for every insert without bind variables in this case, we spent 2/3^{rds} of the time to execute the statement simply *parsing* the statement! But it gets worse. When we look at other information, we can see a significant difference in the resources utilized by each approach:

```
Name                                  Run1        Run2        Diff
STAT...parse count (hard)                4      10,003       9,999
LATCH.library cache pin             80,222     110,221      29,999
LATCH.library cache pin alloca      40,161      80,153      39,992
LATCH.row cache enqueue latch           78      40,082      40,004
LATCH.row cache objects                 98      40,102      40,004
LATCH.child cursor hash table           35      80,023      79,988
LATCH.shared pool                   50,455     162,577     112,122
LATCH.library cache                110,524     250,510     139,986

Run1 latches total versus runs -- difference and pct
Run1        Run2        Diff       Pct
407,973     889,287     481,314     45.88%

PL/SQL procedure successfully completed.
```

The runstats utility I wrote produces a report that shows differences in latch utilization as well as differences in statistics. Here I asked runstats to print out anything with a difference greater than 1000 (in the call the rs_stop, that is the meaning of the number 1000). You can see we “hard” parsed 4 times in the first approach using bind variables but we hard parsed over 10,000 times without bind variables (once for each of the inserts). But that is just the tip of the iceberg – you can see here that we used more than twice as many “latches” in the non-bind variable approach than we did with bind variables. The might beg the question “what is a latch” so we’ll start with that briefly. A latch is a type of lock – it is used to serialize access to shared data structures used by Oracle. The shared pool is a big shared data structure found in the System Global Area (SGA) for example. This is where Oracle stores parsed, compiled SQL. In order to modify this shared structure, we must take care to allow only one process in at a time (it is very bad if two processes or threads would attempt to update the same in memory data structure simultaneously, corruption would abound). So, Oracle employs a ‘latching’ mechanism, a lightweight locking method to serialize access. Don’t be fooled by the word lightweight – these are serialization devices, one at a time. The latches overused by the hard parsing implementation are some of the most used latches out there – the latch into the shared pool, the latch for the library cache. Those are “big time” latches, latches that people compete for frequently. What that means is – as we increase the number of users attempting to hard parse statements simultaneously – our performance problem will get progressively worse and worse over time. The more people parsing, the more people waiting in line to latch the shared pool, the longer the queues, the longer the wait.

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 the chapter 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. While this sounds like an extremely positive attribute (it generally is), if you do not understand this thoroughly, and you attempt to enforce integrity constraints in your application via application logic – *you are most likely doing it incorrectly.*

* 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; every database has subtle differences in its approach to locking. Even if you go with "lowest common denominator SQL" in your applications, the locking and concurrency control models employed by each vendor assure something will be different. 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 <= :new_end_time)
  AND (end_time >= :new_start_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 in part 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;
```

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.

When we get the chapters on Locking and Concurrency control we'll get into this topic in much more depth. There we will learn that integrity constraint enforcement of the type we did above – where we must enforce a rule that crosses multiple rows in a single table, or is between two or more tables (like a referential integrity constraint) are cases where we must always pay special attention and will most likely have to restore to manual locking or some other technique to ensure integrity in a multi-user environment.

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 the chapter on *Concurrency Control*, 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:

```
ops$tkyte@ORA10G> create table t
  2  as
  3  select *
  4    from all_users;
Table created.

ops$tkyte@ORA10G> variable x refcursor

ops$tkyte@ORA10G> begin
  2     open :x for select * from t;
  3  end;
  4  /
PL/SQL procedure successfully completed.

ops$tkyte@ORA10G> delete from t;
28 rows deleted.

ops$tkyte@ORA10G> commit;
Commit complete.

ops$tkyte@ORA10G> print x

USERNAME                          USER_ID CREATED
------------------------------ ---------- ---------
BIG_TABLE                             411 14-NOV-04
OPS$TKYTE                             410 14-NOV-04
DIY                                    69 26-SEP-04
…
OUTLN                                  11 21-JAN-04
SYSTEM                                  5 21-JAN-04
```

```
SYS                              0 21-JAN-04

28 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 – it would work as well), 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*.

In the past, Oracle always decided when our queries would be consistent as of for us. That is – Oracle made it such that any result set we opened would be current as of one of two points in time:

* The point in time the query was opened. This is the default behavior in read committed isolation (we'll be covering the differences between read committed, read only, serializable transaction levels in the chapter on Concurrency Control)
* The point in time the transaction the query is part of began. This is the default behavior in read only and serialiable transaction levels.

Starting with Oracle9i however, we have the ability to tell Oracle when to execute a query "as of" (with certain reasonable limitations on the length of time you can go back in to the past of course). So, for example you can 'see' read consistency and multi-versioning even more directly. Consider this following example. We start by getting an SCN (System Change or System Commit number – the terms are interchangeable). This SCN is Oracle's internal clock – every time a commit occurs, this clock ticks upwards, it increments. We could use a date or timestamp as well, but here the SCN is readily available and very precise:

```
scott@ORA10G> column scn format 99999999999999999999 new_val s
scott@ORA10G> select dbms_flashback.get_system_change_number scn from dual;

                  SCN
---------------------
```

```
         42641429

scott@ORA10G> variable SCN number
scott@ORA10G> exec :scn := &s;

PL/SQL procedure successfully completed.
```

We got the SCN simply so we have a "point in time" we'd like to query as of. We want to be able to query Oracle later and see what was in this table at this precise moment in time. Continuing on, we will see what is in the EMP table right now:

```
scott@ORA10G> select count(*) from emp;

  COUNT(*)
----------
        14
```

And then we'll delete all of this information and verify it is "gone":

```
scott@ORA10G> delete from emp;
14 rows deleted.

scott@ORA10G> select count(*) from emp;

  COUNT(*)
----------
         0
```

However, using the flashback query – the "AS OF SCN" or "AS OF TIMESTAMP" clause, we can ask Oracle to reveal to us what was in the table as of that point in time:

```
scott@ORA10G> select count(*) from emp AS OF SCN :scn;

  COUNT(*)
----------
        14
```

Further, this capability works across transactional boundaries – and we can even query the same object "as of two points in time" in the same query! That opens some interesting opportunities indeed:

```
scott@ORA10G> commit;
Commit complete.

scott@ORA10G> select *
  2  from (select count(*) from emp),
  3       (select count(*) from emp as of scn :scn)
  4  /
```

```
 COUNT(*)  COUNT(*)
---------- ----------
         0         14
```

Lastly, if you are using Oracle10g and above, you have a command called "flashback" that uses this underlying multi-versioning technology to allow you to put back objects the way they were at some prior point in time – so, in this case, we can put EMP back the way it was before we deleted all of the information:

```
scott@ORA10G> flashback table emp to scn :scn;
Flashback complete.

scott@ORA10G> select *
  2   from (select count(*) from emp),
  3        (select count(*) from emp as of scn :scn)
  4  /

 COUNT(*)  COUNT(*)
---------- ----------
        14         14
```

This is what read-consistency and multi-versioning 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 full scanning 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 the chapter on *Concurrency Controls*):

Table 1-1. Accounts table contents

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. Read consistency is the way in which Oracle avoids such occurrences, and how Oracle's methods differ from most 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. I have seen actual cases where a report written by a developer who did not understand Oracle's multi-versioning capabilities would lock an entire system up as tight as could be. The reason – they wanted to have read consistent (eg: correct) results from their queries. In every other database they worked in this required locking the tables, or using a "select … with holdlock" (a SQL Server mechanism for locking rows in a shared mode as you go along). So they would either lock the tables prior to running the report or use "select …. for update" (the closest they could find to with holdlock). This would cause their system to basically stop processing transactions – needlessly.

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:

I guess I need a lesson in tables?? This should be a 3 column table with a time, session 1, session 2 dimension. I don’t get it?

Table 1-2. Insert table caption here.

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, and to implement correct code, 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. And, if you knew how each database worked in great detail, you would understand that database independence it not something you really want to achieve (a very circular argument!)

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 and substantially change it as you do!

Let's assume that we had deployed the initial resource scheduler application in a database that employed blocking reads (reads are blocked by writes).

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 blocking read system, due to this newly inserted data it would be true that insertions into this table would serialize. The first person would insert their request for "room a" from 2pm to 3pm on Friday – and then run a query looking for overlaps. The next person would try to insert an overlapping request and upon looking for overlaps would become blocked (waiting for the newly inserted data it hit to become available for reading). In that blocking read database our application would be apparently well behaved (well, sort of – we could just as easily DEADLOCK, a concept covered in the chapter on Locking as well if we both inserted our rows and then attempted to read each others data) – our checks on overlapping resource allocations would have happened one after the other, never 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. Taking an application from Oracle to SQL Server exposes SQL Servers blocking reads and deadlock issues – it goes both ways.

For example, I've been ask to help 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@ORA10G> select * from dual where null=null;
no rows selected

ops$tkyte@ORA10G> select * from dual where null <> null;
no rows selected

ops$tkyte@ORA10G> select * from dual where null is null;

D
-
X
```

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 the chapter on indexes) 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 SQL99-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.

SQL99 is an ANSI/ISO standard for databases. It is the successor to the SQL92 ANSI/ISO standard, which in turn superceded 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 SQL99-compliant to at least some degree? Did you also 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 had 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 a feature set that is a subset of Oracle 7.0's capabilities.
* *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. 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.

SQL99 defines only two levels of conformance – Core and Enhanced. SQL99 attempts to go far beyond traditional 'SQL' and introduces object relational constructs (arrays, collections). It covers a SQL MM (multi-media) type, Object Relational types and so on. There is no one certifying databases to be SQL99 Core or Enhanced "compliant" and in fact – I know of no one claiming their product is either Core or Enhanced fully compliant.

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. As an analogy – Oracle is a portable application. It runs on many operating systems. However on Windows it runs in the Windows way – using threads and other Windows specific facilities. On Unix, Oracle runs as a multi-process server, using individual processes to do what threads did on Windows – that is the Unix way. The 'core Oracle' functionality is available on both platforms – but it is implemented in very different ways under the covers. Your database applications that must function on multiple databases – will be the same.

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:

```
ops$tkyte@ORA10G> create table id_table
  2  ( id_name  varchar2(30) primary key,
  3    id_value number );
Table created.
```

```
ops$tkyte@ORA10G> insert into id_table values ( 'MY_KEY', 0 );
1 row created.

ops$tkyte@ORA10G> commit;
Commit complete.
```

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

```
ops$tkyte@ORA10G> update id_table
  2    set id_value = id_value+1
  3   where id_name = 'MY_KEY';

1 row updated.

ops$tkyte@ORA10G> select id_value
  2    from id_table
  3   where id_name = 'MY_KEY';

  ID_VALUE
----------
         1
```

Looks simple enough, but the outcomes (notice plural) are that either:

* 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.
* In Oracle (and the behavior might be different in other databases), all but the first user to attempt to concurrently perform this operation would receive "ORA-08177: can't serialize access for this transaction" in the isolation level of SERIALIABLE.

For example, using a serializable transaction (more common in the J2EE environment, many tools automatically use this as the default mode of isolation, unbeknownst to the developers many times) you would observe the following behavior. Notice that the SQL prompt contains information about which session is active in this example:

```
OPS$TKYTE session(261,2586)> set transaction isolation level serializable;
Transaction set.

OPS$TKYTE session(261,2586)> update id_table
  2    set id_value = id_value+1
  3   where id_name = 'MY_KEY';
1 row updated.

OPS$TKYTE session(261,2586)> select id_value
  2    from id_table
  3   where id_name = 'MY_KEY';

  ID_VALUE
----------
         1
```

Now, we'll go to another SQL*Plus session and perform the same operation, a concurrent request for a unique id:

```
OPS$TKYTE session(271,1231)> set transaction isolation level serializable;
Transaction set.

OPS$TKYTE session(271,1231)> update id_table
  2     set id_value = id_value+1
  3   where id_name = 'MY_KEY';
```

This will block at this point, only one transaction at a time can update the row. This demonstrates the first possible outcome – you would block and wait for the row. But since we are using serializable in Oracle, we'll observe the following behavior as we commit the first sessions transaction:

```
OPS$TKYTE session(261,2586)> commit;
Commit complete.
```

The second session will immediately now display the following error:

```
OPS$TKYTE session(271,1231)> update id_table
  2     set id_value = id_value+1
  3   where id_name = 'MY_KEY';
update id_table
       *
ERROR at line 1:
ORA-08177: can't serialize access for this transaction
```

So, that database independent piece of logic really isn't database independent – it doesn't even perform reliably in a single database depending on the isolation level! Sometimes we block and wait – sometimes we get an error message. To say the end user would be upset given either case (wait long time, or wait long time to get error) is putting it mildly.

This issue is compounded by the fact that our transaction is much larger than 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. A more performance driven approach would be simply:

```
Insert into t ( pk, …. ) values ( t_seq.NEXTVAL, …. );
```

That is, skip the overhead of the trigger altogether (this would be my preferred approach). 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 – and that you have verified work exactly the same (remember the null = null discussion!). 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.

Another argument for this approach – developing specialized code for the database you will deploy on – is that finding a single developer, let alone a team of developers, who are savvy enough to understand the nuances, the differences between Oracle, SQL Server and DB2 (let's just limit the discussion to three databases in this case) will be virtually impossible. I've worked mostly with Oracle for the last eleven years (mostly, not exclusively). I learn something new about Oracle *every single day I use it.* To suggest that I could be expert in three databases simultaneously and understand what the differences between all three are and how it will affect the 'generic code' layer we would have to build is highly questionable. I doubt I would be able to do that accurately or efficiently. And – there is the fact that we are talking about individuals here, how many of your developers actually fully understand or utilize the database they currently have – let alone three of them? Finding the unique individual that can develop bullet proof, scalable, database independent routines is a holy grail. Building a team of developers that can – impossible. Finding an Oracle expert, a DB2 expert and a SQL Server expert and telling them "we need a transaction to do X, Y and Z" – that is relatively easy. They are told "here are your inputs, these are the outputs we need and this is what this business process entails" and produce transactional API's (stored procedures) that do that. Each will be implemented in the manner best for that particular database, according to that databases unique set of capabilities. These developers are free to utilize the full power (or lack thereof as the case may be) of the underlying database platform.

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 (however, 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 the chapter on *Concurrency Control* of this book, and see how their differences may affect you. SQL92/SQL99 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 10g has to offer. That would be an extremely large book in itself. The new features of Oracle 9iR1, 9iR2 and 10gR1 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 (for every question you see that I've published – many of which are pointers into the documentation, there are two more questions

I choose not to publish, almost all of which are "read this" answers). 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 MATERIALIZED VIEW command. This is what built-in functionality in the database. Actually, there are many ways to implement replication from read only materialized views, to updateable materialized views, to peer to peer replication, to streams based replication.

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 than 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 old and new releases – 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 working with some developers with years of experience developing database applications – on other databases. They built analysis software (trending, reporting, visualization software). It was to work on clinical data (healthcare related). They were not aware of SQL syntactical features like inline views, analytic functions, scalar subqueries. Their major problem was they needed to analyze data from a single parent table to two child tables, an Entity Relation Diagram (ERD) might look like this:

See fig1.doc

Figure 1 – Simple ERD

The needed to be able to report on the parent record with aggregates from each of the children tables. The databases they worked with in the past did not support subquery factoring (WITH clause) nor did they support inline views – the ability to "query a query" instead of query a table. Not knowing these features existed, they wrote their own database of sorts in the middle tier. They would query the parent table and for each row returned run an aggregate query against each of the child tables. This resulted in their running thousands of queries for each single query the end user wanted to run. Or, they would fetch the entire aggregated child tables into their middle tier into hash tables in memory – and do a hash join.

In short, they were re-inventing the database, performing the functional equivalent of a nested loops join or a hash join, without the benefit of temporary tablespaces, sophisticated query optimizers and the like. They were spending their time developing, designing, fine tuning, enhancing software the was trying to do the same thing the database they already bought does! Meanwhile, end users are asking for new features but not getting them, because the bulk of the development time is in this reporting "engine" which really was a database engine in disguise.

When I showed them that they could do things like:

```
select p.id, c1_sum1, c2_sum2
  from p,
       (select id, sum(q1) c1_sum1
          from c1
         group by id) c1,
       (select id, sum(q2) c2_sum2
          from c2
         group by id) c2
 where p.id = c1.id
   and p.id = c2.id
/
```

Inline Views – query from a "query"

```
select p.id,
       (select sum(q1) from c1 where c1.id = p.id) c1_sum1,
       (select sum(q2) from c2 where c2.id = p.id) c2_sum2
  from p
 where p.name = '1234'
/
```

Scalar Subqueries – run another query per row

```
with c1_vw as
(select id, sum(q1) c1_sum1
   from c1
  group by id),
c2_vw as
(select id, sum(q2) c2_sum2
   from c2
  group by id),
c1_c2 as
(select c1.id, c1.c1_sum1, c2.c2_sum2
   from c1_vw c1, c2_vw c2
  where c1.id = c2.id )
select p.id, c1_sum1, c2_sum2
  from p, c1_c2
 where p.id = c1_c2.id
/
```

With Subquery factoring

Not to mention what we can do with analytic functions (which are so important, I've dedicated an entire chapter to just these functions in this book) like LAG, LEAD, ROW_NUMBER, the ranking functions and so much more – well, rather than spending the rest of the day trying to figure out how to tune their middle tier database engine, we spent the day with the SQL Reference Guide projected on the screen (coupled with SQL*Plus to create ad-hoc demonstrations of how things worked). The end goal was no longer tuning the middle tier, now it was "how fast can we turn off the middle tier".

I have seen people in an Oracle 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. 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, 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. *I've been guilty of this myself.* I still remember the day when my Oracle Sales Consultant (I was the customer at the time) walked in and saw me surrounded by a ton of Oracle documentation. I looked up at him and just asked "is this all true?". I spent the next couple of days just digging, reading – I had fallen into the trap that "I knew all about databases" – because I had worked with SQL/DS, DB2, Ingress, Sybase, Informix, SQLBase, Oracle and others. Rather than take the time to see what each had to offer, I would just apply what I knew from the others to whatever I was working on (moving to Sybase/SQL Server was the biggest shock to me – it worked nothing like the others at all). Upon actually discovering what Oracle could do (and the others to be fair), I started taking advantage of it – and was able to move faster, with less code. This was in 1993. Imagine what you can do with the software today – over a decade later.

I learn something new about Oracle pretty much every single day. It requires some keeping up with. I myself read the documentation (still).

Unless you take the time to learn what is available, you are doomed to do the same thing at some point. In this book, 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 than 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@ORA10G> create profile one_session limit sessions_per_user 1;
Profile created.

ops$tkyte@ORA10G> alter user scott profile one_session;
User altered.

ops$tkyte@ORA10G> alter system set resource_limit=true;
System altered.

ops$tkyte@ORA10G> connect scott/tiger
Connected.
scott@ORA10G> host sqlplus scott/tiger

SQL*Plus: Release 10.1.0.2.0 - Production on Sun Nov 28 12:49:49 2004
Copyright (c) 1982, 2004, Oracle.  All rights reserved.
ERROR:
ORA-02391: exceeded simultaneous SESSIONS_PER_USER limit

Enter user-name:
```

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 seen projects where application development has been going on for months – with no end in sight. The developers are using the latest greatest technologies and languages but it is not going very fast. It wasn't that big of an application – and perhaps that was the problem. If you are building a doghouse (a small wood working job) you would not bring in the heavy machinery. You would use a few small power tools – but you wouldn't have use for the "big stuff". On the other hand, if you were building an apartment complex – you would have a cast of hundreds working in the project, you would have the big machines – you would use totally different tools to approach this problem. The same is true of application development. There is not one single "perfect architecture". There is not one single "perfect language". There is not one single "perfect approach". To build my web site – I used HTML/DB. It was a smallish application, there was a single developer (or two) working on it. It has maybe 20 screens. PL/SQL and HTML/DB was the correct choice for this implementation – it did not need a cast of dozens, coding in Java, making EJB's and so on. It was a simple problem, solved simply. There are few complex large scale, huge applications (we buy most of those today – our HR systems, our ERP systems and so on). There are thousands of small applications. We need to use the proper approach and tools for the job.

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 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 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 J2EE).
* 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 a large variety of open systems 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 25 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* . 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 Oracle Net (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 design and implementation level – not the database level. You cannot tune a database until you have tuned the applications that run on the database.

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. A common side effect with cursor sharing is something like this:

```
ops$tkyte@ORA10G> select /* TAG */ substr( username, 1, 1 )
  2    from all_users au1
  3   where rownum = 1;

S
-
B

ops$tkyte@ORA10G> alter session set cursor_sharing=force;
Session altered.

ops$tkyte@ORA10G> select /* TAG */ substr( username, 1, 1 )
  2    from all_users au2
  3   where rownum = 1;

SUBSTR(USERNAME,1,1)
--------------------
B
```

What happened there? Why is the column reported by SQL*Plus suddenly so large for the second query, arguably the same query? If we look at what the cursor sharing setting did for us, it (and something else) will become obvious:

```
ops$tkyte@ORA10G> select sql_text from v$sql where sql_text like 'select /* TAG */ %';

SQL_TEXT
---------------------------------------------------
select /* TAG */ substr( username, 1, 1 )   from all_users au1  where rownum =
1

select /* TAG */ substr( username, :"SYS_B_0", :"SYS_B_1" )   from all_users au
2  where rownum = :"SYS_B_2"
```

The cursor sharing removed information from the query. It found *every* literal – including the substr constants we were using. It removed them from the query and replaced them with bind variables. The SQL engine no longer knows that the column is a substr of length 1 – it is of indeterminate length. Also, you can see that "where rownum = 1" is now bound as well. That seems like a good idea – however, the optimizer has just had some important information removed. It no longer knows that "this query will retrieve a single row", it now believes "this query will return the first N rows and N could be any number at all". In fact – if you run these queries with SQL_TRACE=TRUE (the chapter on *Tuning Tools and Techniques* will cover that in much more detail), you will find the query plans used by each query and the amount of work they perform to be very different! Consider:

```
select /* TAG */ substr( username, 1, 1 )
  from all_users au1
 where rownum = 1

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        2      0.00       0.00          0         77          0           1
------- ------  -------- ---------- ---------- ---------- ----------
total        4      0.00       0.00          0         77          0           1

Misses in library cache during parse: 0
Optimizer mode: ALL_ROWS
Parsing user id: 412

Rows     Row Source Operation
-------  ---------------------------------------------------
      1  COUNT STOPKEY (cr=77 pr=0 pw=0 time=5767 us)
      1   HASH JOIN  (cr=77 pr=0 pw=0 time=5756 us)
   1028    HASH JOIN  (cr=70 pr=0 pw=0 time=8692 us)
      9     TABLE ACCESS FULL TS$ (cr=15 pr=0 pw=0 time=335 us)
   1028     TABLE ACCESS FULL USER$ (cr=55 pr=0 pw=0 time=2140 us)
      4    TABLE ACCESS FULL TS$ (cr=7 pr=0 pw=0 time=56 us)
********************************************************************************
select /* TAG */ substr( username, :"SYS_B_0", :"SYS_B_1" )
  from all_users au2
 where rownum = :"SYS_B_2"

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        2      0.00       0.00          0         85          0           1
------- ------  -------- ---------- ---------- ---------- ----------
total        4      0.00       0.00          0         85          0           1

Misses in library cache during parse: 0
Optimizer mode: ALL_ROWS
Parsing user id: 412
```

```
Rows     Row Source Operation
-------  ---------------------------------------------------
      1  COUNT  (cr=85 pr=0 pw=0 time=3309 us)
      1   FILTER  (cr=85 pr=0 pw=0 time=3301 us)
   1028    HASH JOIN  (cr=85 pr=0 pw=0 time=5343 us)
   1028     HASH JOIN  (cr=70 pr=0 pw=0 time=7398 us)
      9      TABLE ACCESS FULL TS$ (cr=15 pr=0 pw=0 time=148 us)
   1028      TABLE ACCESS FULL USER$ (cr=55 pr=0 pw=0 time=1079 us)
      9     TABLE ACCESS FULL TS$ (cr=15 pr=0 pw=0 time=90 us)
```

The plans were subtly different, they did different amounts of work. So, just turning on cursor sharing is something to do with great trepidation (well, testing really – you need to test this). It will potentially change the behavior of your application (the column widths for example) and because it removes *all* literals from SQL – even those that never really change, it can have a negative impact on your query plans.

Additionally, I have proven 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 the chapter on *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.

There are no silver bullets, *none.* If there were – they would be the default behavior and you would never hear about them!

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 Oracle Net, getting the listener going, configuring shared server, 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.

The single largest DBA responsibility is database recovery. Note, I did not say "backup", I said recovery – 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 `PGA_AGGREGATE_TARGETE` should be – that's typically the job of the DBA (and the database is quite willing and able to assist them in determining the correct fugire). 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 than 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.
* And building on that last point – software projects come and go, programming languages come and go – but the *data,* the data is here forever. We build applications that use data and that data will be used by many applications over time. It is not about the application – it is about the data. Use techniques and implementations that permit the data to be used and reused over and over. If you use the database as a bit bucket – making it so that all access to any data must come through your application – you have missed the point. You cannot "ad hoc query" your application. You cannot build a new application on top of your old application. But if you use the database, you'll find adding new applications, reports, whatever to be much easier over time.