

CHAPTER 1

The Right Approach to Building Applications

2 Effective Oracle by Design



In this chapter, we will look at some of the “softer” issues regarding Oracle best practices. These issues pertain to system design, development, testing, deployment, and maintenance. They are not really Oracle-specific, but they are approached from an Oracle perspective. Each of these items would be relevant in virtually every system, regardless of the software or even the goal (what you were trying to build).

This will be the least technical of all of the chapters in this book. Nevertheless, it may be the most important one. Time after time, the mistakes I see being made are more procedural than technical in nature. Here are a few examples:

- The tendency to put up walls between developers and database administrators (DBAs). I’ll examine the potential consequences of this and offer advice to help make the relationship more productive for everyone involved.
- The decision that a testing environment is too costly. On the contrary, not having one is far more expensive in the long run.
- The failure to fully exploit what the database has to offer. This can be due to ignorance; or it can stem from a desire to remain database-independent; or it can be related to fear, uncertainty, and doubt what I call FUD). In any case, it often leads to applications that take longer than necessary to develop and that don’t perform up to the developer’s expectations.

These and related topics are the focus of this chapter.

It’s a Team Effort

Team effort really has nothing to do with technology or software. This is all about human interaction. Let’s face it: Many of the issues we encounter during software development have more to do with politics than they have to do with technology. I cannot tell you how many times I’ve seen development efforts thwarted by policy, procedure, and politics, rather than stymied by some technical challenge.

Too often, the relationship between database development team members and the DBA staff members that support them is characterized by a heavy-duty “us versus them” mentality. In such an atmosphere, DBAs often feel a need to protect the database from the developer. On the other hand, developers feel that they must thwart the DBA in order to implement features in the way they desire. At times, trying to get these two groups to work together, I’ve felt more like a marriage counselor than an onsite database expert!

What we really must remember is that teamwork is a two-way street. Developers often feel that DBAs impose too much of a burden on them to prove why they need a certain privilege or why they need a certain feature enabled. In fact, this is often a reasonable request. The granting of database privileges should be done with care and thought. There is nothing wrong with a DBA requesting that a developer document why he needs, for example, the CREATE VIEW privilege granted directly to his development account (we’ll take a deeper look at this in a moment). On the other hand, it is unreasonable for a DBA to assume *carte blanche* authority to outlaw certain database features, such as database views, stored procedures, or triggers.

Here is just one example of such attitudes in action (taken from my AskTom website):

“Stored procedures—are they evil?”

What are the drawbacks of using stored procedures? Is there any overhead? I am a novice Oracle DBA. I have been asked by one of the SQL programmers to authorize him with the CREATE PROCEDURE system privilege on an ORACLE user...”

In my answer, I explained the overwhelming benefits of stored procedures. I pointed out how they can help the DBA tune the applications (without having to dive into the application). I emphasized that stored procedures are a great security mechanism, a performance enhancement tool, and an awesome programming device. I advised the DBA to permit the use of stored procedures. I suggested that developers could do as much, if not more, damage by putting bad SQL in their client applications. Furthermore, I pointed out that if the SQL were in stored procedures, at least the DBA could reasonably inspect it and help to tune it and manage it.

The follow-up responses were extremely polarized, demonstrating the existence of a chasm between the development and DBA camps. The attitude of the DBAs was, “It’s my job to protect the database.” The attitude of the developers was, “It’s my job to code, and it’s their job to let me do that.” One developer expressed the opinion that it’s the DBA’s job to review every line of code that goes into the database. A DBA rejected this, saying that it was, in fact, the developer’s job to do that. And so the argument went on.

In reality, it would be impossible for the DBA to inspect and review each line of code going into the database, forgetting for the moment that a DBA isn’t a developer in general and wouldn’t necessarily understand what they were even looking at if they did review the source code.

Clearly, the DBA staff and the development staff often feel that they have totally different objectives, but this is not the case. The DBA doesn’t work just to protect the database, but neither does the DBA work solely to serve the developer.

Much like an overprotective parent, a DBA who takes the “protect the database from the developers” mindset will not be productive. On the other hand, the developer isn’t programmed to thwart the DBA (contrary to popular DBA lore). Developers have a job to do and are just trying to do it. Their shared objective is to build a functioning, functional database application that meets the end users’ requirements, performs well, scales as required, is maintainable, and was developed in as cost-effective manner as possible. Unless the two teams work together toward this common goal, the chances for success are severely curtailed. If there is a virtual wall, or even worse, a physical wall between the two teams, many of these goals cannot be realized.

DBA and Developer Roles

Typically, the DBA knows more than the developer about the database and how it works, and the developer knows more than the DBA about software development. This is a natural outcome of their job descriptions.

4 Effective Oracle by Design

DBAs are generally responsible for understanding the database architecture, how to patch the database, and how it works. They would not be able to craft a successful backup and recovery procedure (for example, if they did not have this knowledge) as an intimate understanding of the Oracle architecture is needed to accomplish that particular task. If the DBA didn't understand the architectural relationship between database control files, datafiles, and redo logs, they would invariably make mistakes during the backup and recovery process. The DBA lives to work with the database.

Developers are generally programmers/analysts who see the database as just another tool—something that must be used to achieve an end. In many cases, they spend much of their time doing “nondatabase” work, such as interface design.

What we need here is some cross-pollination. If the two teams can work together, then gradually, the developers will know more about how the database works, and the DBAs will be in a much better position to help facilitate their development processes.

I've drawn up two lists of do and don't advice—one list for DBAs and one for developers—that will be helpful in closing out this section. This advice addresses some of the more damaging attitudes that exist.

DBA Do's and Don'ts

DBAs, do not consider that your primary job is to protect the database from the evil developers. The database is their tool, and you can and should counsel them on how best to use it, rather than attempt to protect it from them. Do not consider the developer your enemy.

Also, do not outlaw a feature or function without proper justification. Too often, these decisions are based on fear and uncertainty, or on that one bad experience. The following are some common restrictions:

- No views allowed. The reasoning behind this is usually that a DBA once experienced bad performance with a view and, therefore, views must be evil. If you applied such reasoning consistently, you would end up outlawing SQL when you came across a poorly performing query.
- No stored procedures allowed. This one really confuses me, since a DBA should optimally want everything in a stored procedure. If this were the case, they would know exactly what module depended on what database object. They would be able to easily tune a poorly performing module if the problem was SQL-based (just read the code from the data dictionary and play with the SQL). Try having a DBA tune a Java 2 Enterprise Edition (J2EE) application some day, since they are *not* developers, they do not program using J2EE, they would not know where to begin. If the database portion of the application is in the database, tuning the database access becomes easy.
- No features added after version 6 allowed. This is common among old-time DBAs, who are leery of anything new—PL/SQL, triggers, application contexts, fine-grained auditing, fine-grained access control, and so on. All they want is keyed reads into the database (if they could only outlaw joins!). My suspicion here is this is a DBA who wants as little responsibility for anything at all. If all the developers do are simple Data Manipulation Language (DML) statements, the DBA's job is trivial. But the company as a whole loses, since it paid a lot of money for this database stuff but cannot use it.

- No adoption of new features for *N* releases. The idea behind this is to let everyone else deal with any problems with this feature; we'll use it in three or four years. The problem is that for those three or four years, you could be missing out on the greatest thing since sliced bread—a feature that could save you hours or days of effort (hence, money). Locally managed tablespaces (introduced in Oracle8i, Release 1) come to mind as the classic example. This good feature had many positive benefits, yet many DBAs outlawed it. The reasons were never technical; they were all FUD (fear, uncertainty, doubt)-based.

"I read about the advantages of locally managed tablespaces and asked my senior DBA if, for the new data warehouse initiative, we could change tablespaces to locally managed. He says that there are performance issues with locally managed tablespaces..."

The DBA was right, there are performance issues, but they are all *positive*. The tablespaces the DBA was using (dictionary-managed) are the ones that have negative performance issues. Lack of correct information leads to rejection of new features.

On the other hand, don't get sucked into "feature obsession," when a feature or function is deemed so unique and so cool that you feel compelled to use it. For example, consider Extensible Markup Language (XML) functionality in the database. Just because it is there doesn't mean everything should be stored in XML. It is just as easy to get carried away with a feature as it is to ignore it.

Here are some do's for DBAs:

- Do consider the developer as someone you can teach and to whom you can pass on your database knowledge. You will find that, over time, that pesky developer actually wants to do the right thing. If you teach people the right way—better yet, teach them how to discover for themselves the right way—they will use it. People want to do the right thing, but they are starved for information. Spread your knowledge around.
- Do evaluate and test new features as they come out. Do not dismiss them out of hand. Do not let one bad experience with a feature cause you to dismiss it entirely. Everything has a time and place; every feature is there for a reason. Perhaps (most likely) the bad experience was due to an inappropriate application of the technology. A hammer is a really good tool for nailing down things down, but it is horrible tool to use to screw in things.
- Do back up your policy and procedures with factual evidence. Never be in the position of saying "I've heard they were slow," or "I've heard there were bugs." Hearsay isn't admissible in court, and it shouldn't be admissible here. This is just like believing all of those performance-tuning myths that abound, such as "If you have a 99.9% cache hit ratio, your job is done," or "A table should be in one extent." In this book, I will present factual evidence to back up my assertions. You should expect no less from anyone else (yourself included!)

Developer Do's and Don'ts

To the developers, I say this:

- Do not try to work around the DBAs; work with them. If you present a factually correct and compelling argument, odds are they will listen. It is when you try to do whatever you want, without involving them, that they start locking you out entirely. You become that loose cannon—something to rightly fear.
- Do not assume that the DBAs are working against you. In many cases, the policies and procedures in place are there for a reason. Work to change the rules you don't agree with, but don't try to subvert them (loose cannon syndrome again).
- Do ask the DBAs to tell you why. If you suggested the use of locally managed tablespaces, but the senior DBA said no because of “performance issues,” ask that senior DBA for those performance-related references. Explain that you want to learn why locally managed tablespaces have these bad performance characteristics (they don't, by the way).
- Do make sure you know what you are talking about. Otherwise, you'll instantly lose credibility. Use the scientific method: Start with a hypothesis. Construct a test to prove (or equally as often disprove) the hypothesis. Make sure the test is general enough (broad enough) to prove the point and is reproducible. Have others reproduce it. Prepare the results and have others critique it. In that fashion, you'll have a very solid leg to stand on when describing why your approach is correct.

In short, I believe the DBA and development team need to work together as a whole. They should sit together and converse, meeting on a regular basis. They should not be considered two independent divisions that just throw work to each other over a wall. The “them versus us” mentality will prohibit a fast, reliable, available system. Rather than have development teams and DBA teams in separate areas, it makes more sense to have a unified team of developers and DBAs who can take advantage of each other's specialized knowledge.

Read the Documentation

The Oracle database comes with more than 100 manuals (108 with Oracle9i Release 2). There are over 46,000 pages of text. It can be quite intimidating trying to figure out where to begin. In fact, many people are so intimidated by so much documentation that they pretend it doesn't exist. In reality, it is really quite simple to figure out where to begin once you are told, and that is what this section is all about. First, we will go over the key pieces of documentation, and then I'll suggest just what should be considered mandatory reading.

“I have learned a lot from your site. One point that you constantly bring up that wasn't getting through my thick head though was READ THE CONCEPTS GUIDE. So, over the past couple of weeks, I put a copy of the PDF version on my laptop and worked my way through it. I am very glad that I did (a reread is coming soon)! Many things I didn't understand are clear now, and being a consultant who works with a lot of other developers and DBAs, I now know most all of you have not read this either.”

That is the best testimonial I could ever have hoped for. I frequently get questions that start with, “Please don’t point me to the documentation.” This is a request that I almost always ignore, because often their question is elegantly answered in the documentation. I frequently begin an answer with, “Well, the handy-dandy *Concepts Guide* has this to say on the topic....” I have been known to say more than once that if you simply read the *Concepts Guide* from cover to cover and retain just 10% of what it contains, you’ll already know 90% more than most people do about Oracle and how it works. And, in the future, when a problem comes up, you can say, “I remember something about that. Let’s look in the *Concepts Guide* for a solution.”

A Guide to the Guides

Given that there are more than 100 pieces of documentation, the road maps I suggest after this guide to the guides are surprisingly terse. The guides summarized here are the ones you should hit from cover to cover (precisely which ones depends on whether you are a developer or a DBA). The remaining documentation can serve as reference material, depending on your needs and interests.

The Concepts Guide

The *Concepts Guide* is the one piece of Oracle documentation that I make a point to read with each major release of the database. It contains a wealth of information on topics such as:

- **What is Oracle?** An introduction to the database, memory structures, distributed database, concurrency controls, data consistency, security, administration, and other topics.
- **Database structures** An in-depth look at how things are stored. The who, what, where, when, and why of database blocks, extents, segments, tablespaces, and more.
- **The Oracle instance** What an instance of Oracle is, startup, shutdown processing, how applications interact with an Oracle instance, what the memory and process architecture look like, and how database resources are managed.
- **Data** An overview of the entire set of schema objects (such as tables, views, indexes, and so on), all the options available with each type of object, and all the datatypes—both native and user-defined.
- **Data access** SQL, PL/SQL, and Java interactions with the database, how dependencies are managed and maintained between schema objects, and transaction management and triggers.
- **Parallel operations** The hows and whens of parallel operations including parallel query, parallel DML, parallel operations for administration, and so on.
- **Data protection** Perhaps one of the most important sections, it covers topics such as how concurrency and consistency work in Oracle, how and when is data integrity enforced, how security fits into all of this, and what is available to protect data. It also explains how to use privileges and roles, as well as how to audit what is happening.

8 Effective Oracle by Design

One of the nice attributes of the *Concepts Guide* (besides being totally free) is that it acts as a readable “index” to the rest of the documentation. Frequently, topics end with “See Also” links that point to the other pieces of Oracle documentation for all of the gory details. The *Concepts Guide* is a high-level overview, the metadata document for the rest of the documentation. As you read through it, you will be naturally guided to the other relevant pieces of Oracle documentation. You will follow your interests and drill down over time.

New Features Guide

If you don’t know something exists, you’ll never be able to use it right? The *New Features Guide* is a comprehensive guide to the new features, by release, for the past couple of releases. For each feature, the guide includes a terse explanation, along with a pointer to where you can find more details. It also includes a list of “what’s included” in each of the flavors of Oracle (Personal, Standard, and Enterprise). This matrix will save you much frustration if you are designing a system to run on the Standard Edition of Oracle, because the features available to you are clearly listed. In Oracle8i and before, this guide was titled *Getting to Know*.

Additionally, the beginning of most documents now contain a “What’s New In” section as well. The *New Features Guide* presents most new features including administrative, development, performance, scalability, availability or whatever. The individual documents present a more focused list of “What’s New.” For example, the Administrators Guide will have a “What’s New in Administration” section whereas the Application Developers Guide will have a “What’s New in Application Development” section.

Application Developers Guide

There are actually quite a few guides that start with *Application Developers Guide* for various Oracle features, such as Advanced Queuing (messaging software), LOBs (Large Objects), Object Relational Features, and Workspace Management. Recommended reading for all developers is the *Fundamentals* guide. This comprehensive guide takes you through topics such as the following:

- Understanding the various programmatic environments you can access
- Designing your database schemas
- Maintaining data integrity via constraints
- How to begin indexing your data—what to consider, how SQL statements are processed by the engine, using dynamic SQL, using PL/SQL, implementing security, and so on

Whereas the *Concepts Guide* explains the existence of these features, the *Application Developers Guide* explains how to exploit them.

PL/SQL Users Guide and Reference

PL/SQL is one of the single most important languages developers have access to when developing against Oracle. A good understanding of what it can and cannot do for you is vital. The *PL/SQL Users Guide and Reference* covers the fundamentals of PL/SQL, error handling, syntax, packages/procedures, and many other PL/SQL-related topics.

Performance Tuning Guide and Reference

One of my favorite and most frequently referenced guides on a day-to-day basis is the *Performance Tuning Guide and Reference* (called the *Designing and Tuning for Performance Guide* in Oracle8i and earlier). The first half is geared toward developers. It describes how the optimizer actually works, how to gather statistics correctly, how the different physical structures work and when they are best used (for example, when an index-organized table is a good thing). Most important, it fully documents the basic tools you need to use: Explain Plan, SQL_TRACE, TKPROF, Autotrace, and even Statspack!

While the first half of this book is relevant to DBAs as well as to developers, the second half of the book is all about the work of DBAs. It covers topics such as building a database for performance, memory configuration, understanding operating system interactions and resource usage, configuring shared servers and dedicated servers, how to gather statistics, how and when to use performance views, and how to use other performance tools. This is a must read—before you begin tuning and before you begin implementing. Even if you have been working with Oracle for 100 years, you will find something new and useful in here.

Backup and Recovery Concepts

If there is one thing a DBA is not allowed to mess up *ever*, it is backup and recovery. Even if you are using a tool to automate backups, and even if you think you know it all, read the *Backup and Recovery Concepts Guide*. A solid, core understanding of how backup and recovery works will never be something you regret having.

Reading the RMAN guide isn't enough. I know this based on the number of questions I get from people who just read that and don't understand why they cannot fully recover after restoring last week's control files. They are missing that basic understanding of how the files work with each other and what needs to be restored given the current situation. Unless you want to cost your company thousands of dollars (or probably more), read the *Backup and Recovery Concepts Guide*. And if you don't understand something in it, read it again and again until you do. Then test it out; try out your knowledge. You don't want to discover the day of recovery that your knowledge isn't up to par with reality.

Recovery Manager Reference

The *Recovery Manager Reference* talks about the tool you want to be using to back up your database. Forget the old scripts and lose `tar`, `cpio`, `dd`, and `ocopy`. RMAN is the bookkeeper you always wanted. Its features—block-level, in-place recovery, backup retention policies, hot backups, and more—make learning this tool 100% worth your while.

Administrators Guide

Yes, I know DBAs are already administrators, but you will still learn something you didn't know by reading the *Administrators Guide*. This is where you might discover for the first time that there is a resource manager inside the database (new in 8i; enhanced in 9i). Or, you might learn about fine-grained auditing, new in Oracle9i. I bet that if you read this guide, you will learn something new.

Some new features of Oracle with regard to administering the database are to be found only in this document. These are the features deemed “not general enough” to make it into the *New Features Guide*. Additionally, all of the new features relevant to database administration are listed here, without being intermingled and obscured by other new features.

Road Maps to Reading

Here, I present road maps—suggested sets of mandatory reading. I offer three sets: one for everyone, one for people who consider themselves DBAs, and one for people who consider themselves developers.

Required Reading for Both Developers and DBAs

For both groups, to be read with each and every new release, we have:

- *Concepts Guide*
- *New Features Guide*

Required Reading for Developers

For developers, continue your reading with:

- *Application Developers Guide (Fundamentals)*
- *PL/SQL Users Guide and Reference*
- *Performance Tuning Guide and Reference (Designing and Tuning for Performance Guide in 8i and earlier)*

As noted earlier, developers should be sure to read the first half of the *Performance Tuning Guide and Reference*, and look over the other sections at your leisure. As a prequel to this guide in Oracle9i and above, consider looking at the *Performance Method* (9i Release 1) or *Performance Planning* (9i Release 2) reference. This guide, weighing in at a very light 60 pages, describes the topics you need to understand in order to be successful, such as scalability, system architecture, application design principles, and so on.

Required Reading for DBAs

After the *Concepts Guide* and *New Features Guide*, DBAs should continue with:

- *Backup and Recovery Concepts*
- *Recovery Manager Reference*
- *Backup and Recovery Concepts* (No, this isn't a typo. Yes, I put it in here twice. It really is that crucial. It is the one thing you are not allowed to get wrong. It is also the thing that DBAs get wrong the most. Just read it and understand it. You won't regret it.)
- *Administrators Guide*
- *Performance Tuning Guide and Reference* (with special attention to the second half)

Recommended Reading

After reading the guides listed in the previous sections, it is a matter of what interests you. Are you a developer with a need to do XML processing? Well, there are no fewer than three XML guides available. Are you interested in the Java capabilities? There are guides on this topic as well. Are you

a DBA who needs to understand how to set up and configure a failover environment? The manuals have that covered as well.

As you dive into and actually read the documentation, you'll find it is really pretty good. I'm not saying that just because I work for Oracle, rather I'm saying that because most of what I know about Oracle today comes from these very documents. I really do read that *Concepts Guide*. The way I put together things like a "new features seminar" is to dig into the *New Features Guide*. I encourage you to go to <http://otn.oracle.com> and click the documentation link. It's all there, just waiting to be read.

Avoid the Black Box Syndrome

Without a fundamental knowledge of the Oracle database and how it works, it is very easy and common for developers to adopt an approach that is flawed or just plain wrong. Many times, people approach the database as if it were a black box—a commodity as interchangeable as a battery in a radio. According to this approach, you use the database, but you avoid, at all costs, doing anything that would make you database-dependent, as if that were a bad thing. In the name of database independence, you will refuse to use the features of the database, to exploit its capabilities. You actually choose to write off, to not use, most of the functionality that you or your customers have paid for. It means you have chosen to take the path of do-it-yourself—to write more code than you need to maintain and to take longer to get to market.

For a company that has decided to use a particular database, the decision to develop more code that must be maintained and to take longer to get to market must not be taken lightly. It costs money, both in real, measurable terms (you are taking longer to develop the same functionality, and they must pay you) and in softer terms of lost opportunity.

Database Independence versus Database Dependence

Here is something that you might find controversial: Database dependence (regardless of the database you are using; this does not apply to just Oracle) should be your goal, not something you avoid. You want to maximize your investment in your database. You want to develop the best software in the least amount of time against that database. The only way to do that is to fully exploit what the database has to offer.

The truth is that, with the exception of trivial applications, achieving database independence is not only extremely hard, but it is also extremely costly and consumes a huge amount of resources. Yes, perhaps a simple report could be database-independent. But could this be the case with a scalable transaction system? No, not unless you take the approach of companies like PeopleSoft or SAP, and do it on the same scale as they do. Like the products of those companies, your application would not use SQL beyond simple "keyed reads" (sort of like a VSAM file on a mainframe would be used, a file read by a key only—no joins, no analysis, just keyed reads). You would not use any vendor extensions, nor most ANSI SQL functions, since not all vendors have implemented them. You would not be using the database for concurrency controls (since they all do it differently). You would not be using the database for analytics (again, because they all do it differently). In practice, you would end up writing your own database. In fact, SAP did just that: wrote its own database!

So, unless you are making a product that will actually ship on many different databases as off-the-shelf software, database independence isn't a goal you want to achieve. In reality, most software is built to run in-house against a corporate, standard database. The need for database

independence is questionable at best in these circumstances. Getting applications built, getting them built quickly, and getting them built with as few lines of code as possible for maintenance—those are the goals you want to satisfy. Working to stay database-independent or, even worse, just ignoring (either purposely or due to lack of knowledge) the capabilities of your database, is not a laudable goal.

The best way to achieve some level of application portability across multiple databases would be to code all of the database components of your applications in stored procedures. Now that may sound counterintuitive. If we code in stored procedures, and every vendor has its own language, won't we be tied to that vendor? Yes and no. The visual component of your application is safe. The application logic—the logic that falls outside the data logic—is safe. The data logic is encoded in the way that is best for the database on which you are executing. Since it is hidden in a stored procedure, you can make use of—in fact, would almost be forced to make use of—every vendor extension and feature you could in order to have the best data layer. (We'll go over an example of this soon.)

Once developed and deployed, the application stays deployed on that database forever. If it is moved to another database, that move is typically done in combination with a rework of the application itself, as a major upgrade with new features and functions. It is not a simple port.

Dangers of Block Box Syndrome

Here are the reasons why you should not treat the database as a generic block box:

- **Inability to get the correct answer** Concurrency controls are major differentiators between the databases. Applications will get different results given the same exact inputs, in the same exact order, depending on the database it was run against.
- **Inability to perform** Your performance will be a fraction of what it could and should be.
- **Inability to quickly deliver software** You spend large amounts of time “doing it yourself.”
- **Inability to maximize your investment** You spent large amounts of money on this database software; use it. Ironically, the reason people sometimes change vendors is because one does not appear to perform well enough, but the reason for this poor performance is that they didn't use the vendor-specific features in the first place!

These are not listed in any order. Depending on who you are, different points will mean different things. Let's take a look at some examples.

Inability to Perform

Suppose you were developing an application that had a requirement to show an employee hierarchy or a BOM (bill of material) hierarchy. In Oracle, showing almost any hierarchy is achieved via the CONNECT BY statement in SQL. The most efficient, effective way to accomplish this goal would be something similar to the following (using the standard Oracle SCOTT.EMP table):

```
scott@ORA920.US.ORACLE.COM> select rpad('*',2*level,'*') || ename ename
2  from emp
3  start with mgr is null
4  connect by prior empno = mgr
5  /
```

```

ENAME
-----
**KING
****JONES
*****SCOTT
*****ADAMS
*****FORD
*****SMITH
****BLAKE
*****ALLEN
*****WARD
*****MARTIN
*****TURNER
*****JAMES
****CLARK
*****MILLER

```

14 rows selected.

CONNECT BY is not ubiquitous. Many databases do not support this syntax. With other databases, you might need to write some procedural logic and populate a temporary table with the results. You could do that in Oracle as well, but why? It would be slower, it would consume more resources, and it would be the wrong way to do it. Rather, you should do things the right way. In this case, use the CONNECT BY construct and “hide” it in a stored procedure. In this way, you can implement it differently in another database, such as Microsoft SQL Server, if and when the need ever arises (this is rare, in my experience).

Let’s take this a step further. Suppose that you need to return a result set that displays employee information, including the employee name, department, and salary. You also want a running total of the salary by department and the percentage the employee salary represents by department and in total (for example, employee X in department Y makes 10% of the total salary in her department and 1% of the total salaries in the company). The right way to do this in Oracle is to use analytic functions, as follows:

```

scott@ORA920.US.ORACLE.COM> column pct_dept format 99.9
scott@ORA920.US.ORACLE.COM> column pct_overall format 99.9
scott@ORA920.US.ORACLE.COM> break on deptno skip 1

scott@ORA920.US.ORACLE.COM> select deptno,
 2      ename,
 3      sal,
 4      sum(sal) over (partition by deptno order by sal,ename) cum_sal,
 5      round(100*ratio_to_report(sal)
 6          over (partition by deptno), 1 ) pct_dept,
 7      round(100*ratio_to_report(sal) over () , 1 ) pct_overall
 8  from emp
 9  order by deptno, sal
10 /

```

14 Effective Oracle by Design

DEPTNO	ENAME	SAL	CUM_SAL	PCT_DEPT	PCT_OVERALL
10	MILLER	1300	1300	14.9	4.5
	CLARK	2450	3750	28.0	8.4
	KING	5000	8750	57.1	17.2
20	SMITH	800	800	7.4	2.8
	ADAMS	1100	1900	10.1	3.8
	JONES	2975	4875	27.4	10.2
	FORD	3000	7875	27.6	10.3
	SCOTT	3000	10875	27.6	10.3
30	JAMES	950	950	10.1	3.3
	MARTIN	1250	2200	13.3	4.3
	WARD	1250	3450	13.3	4.3
	TURNER	1500	4950	16.0	5.2
	ALLEN	1600	6550	17.0	5.5
	BLAKE	2850	9400	30.3	9.8

14 rows selected.

However, these analytic functions are a feature that many relational databases do not have, so this is essentially a database-dependent technique. There is another way to do this that would work in most databases. This other approach involves the use of self-joins, views, and the like.

```
scott@ORA920.US.ORACLE.COM> select emp.deptno,
2      emp.ename,
3      emp.sal,
4      sum(emp4.sal) cum_sal,
5      round(100*emp.sal/emp2.sal_by_dept,1) pct_dept,
6      round(100*emp.sal/emp3.sal_overall,1) pct_overall
7  from emp,
8      (select deptno, sum(sal) sal_by_dept
9        from emp
10       group by deptno ) emp2,
11      (select sum(sal) sal_overall
12        from emp ) emp3,
13      emp emp4
14  where emp.deptno = emp2.deptno
15        and emp.deptno = emp4.deptno
16        and (emp.sal > emp4.sal or
17              (emp.sal = emp4.sal and emp.ename >= emp4.ename))
18  group by emp.deptno, emp.ename, emp.sal,
19         round(100*emp.sal/emp2.sal_by_dept,1),
20         round(100*emp.sal/emp3.sal_overall,1)
21  order by deptno, sal
22  /
```

DEPTNO	ENAME	SAL	CUM_SAL	PCT_DEPT	PCT_OVERALL
--------	-------	-----	---------	----------	-------------

10	MILLER	1300	1300	14.9	4.5
	CLARK	2450	3750	28.0	8.4
	KING	5000	8750	57.1	17.2
20	SMITH	800	800	7.4	2.8
	ADAMS	1100	1900	10.1	3.8
	JONES	2975	4875	27.4	10.2
	FORD	3000	7875	27.6	10.3
	SCOTT	3000	10875	27.6	10.3
30	JAMES	950	950	10.1	3.3
	MARTIN	1250	2200	13.3	4.3
	WARD	1250	3450	13.3	4.3
	TURNER	1500	4950	16.0	5.2
	ALLEN	1600	6550	17.0	5.5
	BLAKE	2850	9400	30.3	9.8

14 rows selected.

This works and is a more database-independent. However, not many companies that need this functionality have only 14 people in them; most have hundreds or thousands. Let's scale the example up and see what happens in terms of performance. When we run this against data sets of various sizes, we see this sort of performance:

Rows in table	CPU/analytics	CPU/generic	Difference
2000	0.05	2.13	42 times
4000	0.09	8.57	95 times
8000	0.19	35.88	188 times

As the data scales up, the generic implementation gets progressively worse in an exponential fashion. The correct implementation in an Oracle environment scales linearly as the data doubles, and so does the amount of time required to process the analysis. As we doubled the number of rows of data, the generic implementation fell apart.

Now, if I were the end user, I know which implementation I would prefer you to use, and it is not the generic one! This also illustrates why the analysis tools that feature on their packaging "we exploit your native database!" are much more desirable than those that boast "we run generically on 15 databases!" Tools and applications that exploit the database fully will perform better than generic solutions. About the only person who would be happy with the generic solution would be your hardware vendor. You obviously need to upgrade those CPUs.

There are other possible solutions to this problem. A common one would be to use temporary tables, placing the data in them a bit at a time. However, they all suffer from the same fundamental problems as described earlier: They are the wrong way to do it in Oracle (they might be the right way to do it in some other database), and they involve writing a lot more code, which can also be much harder to write.

As another example, I worked with a customer who refused to use a bitmap index. His thinking was, “Well, not everyone has them, so I cannot put it into my system, since that index created would be different in Oracle than everywhere else. I want a generic script that can run on all databases.” Using a bitmap index, the query would go from running in hours to executing in less than a minute. But this customer chose to penalize all implementations because a few might not have a specific feature. I don’t know about you, but I would rather have the bitmap index in Oracle than to have no index in all.

Inability to Get the Correct Answer

As with the problem of being unable to perform, not being able to get the correct answer happens often when you think of the database as a black box. In the previous section, the results were clear, because the raw performance data immediately revealed the right and wrong approach. However, sometimes it is harder to see what you are doing is wrong.

Let’s look at an example that concerns Oracle’s consistency and concurrency controls (multiversioning, read consistency, locking, and so on), because this is where proponents of the black box ideology frequently come unstuck. Consider a simple transaction against a master/detail table. The goal here is to maintain an aggregation of a details table’s rows in the parent table; for example, to maintain the sum of individual employee salaries in the department table. For illustrative purposes, we’ll use these two simple tables:

```
ops$tkyte@ORA920> create table dept
  2  ( deptno          int primary key,
  3    sum_of_salary number
  4  );
Table created.

ops$tkyte@ORA920> create table emp
  2  ( empno           int primary key,
  3    deptno          references dept,
  4    salary           number
  5  );
Table created.

ops$tkyte@ORA920> insert into dept ( deptno ) values ( 1 );
1 row created.

ops$tkyte@ORA920> insert into dept ( deptno ) values ( 2 );
1 row created.
```

Now, when we perform transactions against the child table EMP, we’ll include an UPDATE to the parent table DEPT in order to keep the SUM_OF_SALARY column in sync. For example, our transactions would include the UPDATE statement that appears last in this transaction:

```
ops$tkyte@ORA920> insert into emp ( empno, deptno, salary )
  2  values ( 100, 1, 55 );
1 row created.

ops$tkyte@ORA920> insert into emp ( empno, deptno, salary )
```

```

2 values ( 101, 1, 50 );
1 row created.

```

```

ops$tkyte@ORA920> update dept
2      set sum_of_salary =
3      ( select sum(salary)
4        from emp
5        where emp.deptno = dept.deptno )
6      where dept.deptno = 1;
1 row updated.

```

```

ops$tkyte@ORA920> commit;
Commit complete.

```

This seems straightforward—just insert a child record and update the parent record sum. Nothing so simple could be wrong, or could it? If we query our schema right now, it appears correct:

```
ops$tkyte@ORA920> select * from emp;
```

EMPNO	DEPTNO	SALARY
100	1	55
101	1	50

```
ops$tkyte@ORA920> select * from dept;
```

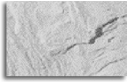
DEPTNO	SUM_OF_SALARY
1	105
2	

If we were to add rows to the child table for DEPTNO 1 or DEPTNO 2 and run that update, everything would be just fine, or so we might think. What we haven't considered is what happens during concurrent access to these tables. For example, let's see what happens when two users work on the child EMP table at the same time. One user will add a new employee to DEPTNO 2. The other user will transfer EMPNO 100 from DEPTNO 1 to DEPTNO 2. Consider what happens when these transactions execute simultaneously. Let's look at a specific sequence of events:

Time	Session 1 Activity	Session 2 Activity
T1	Insert into EMP (EMPNO, DEPTNO, SALARY) values (102, 2, 60); added new employee to DEPTNO 2	
T2		Update EMP (set DEPTNO = 2 where EMPNO = 100); transferred employee across departments

Time	Session 1 Activity	Session 2 Activity
T3		Update DEPT for departments 1 and 2 since we modified a record in both departments
T4		Commit transaction
T5	Update DEPT for department 2, the department we modified	
T6	Commit transaction	

We can simulate this transactional sequence of events easily using two sessions and switching back and forth between them on screen, or we can use a single session with an autonomous transaction.



NOTE

Chapter 16 of the Oracle Server Concepts Manual has full details on the specifics of autonomous transactions. In a nutshell, they are independent transactions within a transaction. They are separate from the current transaction and behave as if they were executed in another session entirely.

Here is the example with an autonomous transaction:

```
ops$tkyte@ORA920> insert into emp ( empno, deptno, salary )
2 values ( 102, 2, 60 );
1 row created.

ops$tkyte@ORA920> declare
2 pragma autonomous_transaction;
3 begin
4
5 update emp set deptno = 2 where empno = 100;
6 update dept
7 set sum_of_salary =
8 ( select sum(salary)
9 from emp
10 where emp.deptno = dept.deptno )
11 where dept.deptno in ( 1, 2 );
12
13 commit;
14 end;
15 /
PL/SQL procedure successfully completed.

ops$tkyte@ORA920> update dept
2 set sum_of_salary =
```

```

3      ( select sum(salary)
4        from emp
5        where emp.deptno = dept.deptno )
6  where dept.deptno = 1;
1 row updated.

```

```

ops$tkyte@ORA920> commit;
Commit complete.

```

That serial execution of statements mimics the two sessions' activities exactly. If all went according to plan, the DEPT table should be up-to-date, refreshed with accurate information. But upon inspection, we discover:

```
ops$tkyte@ORA920> select * from emp;
```

EMPNO	DEPTNO	SALARY
100	2	55
101	1	50
102	2	60

```
ops$tkyte@ORA920> select deptno, sum(salary) from emp group by deptno;
```

DEPTNO	SUM(SALARY)
1	50
2	115

```
ops$tkyte@ORA920> select * from dept;
```

DEPTNO	SUM_OF_SALARY
1	50
2	55

It is obviously wrong. The value for DEPTNO 2 is incorrect. How could that be? If you run this in SQL Server, for example, the scenario would have been slightly different (things would have been forced to execute in a different order), but the numbers would add up. In SQL Server the sequence of events would be:

Time	Session 1 Activity	Session 2 Activity
T1	Insert into EMP (EMPNO, DEPTNO, SALARY) values (102, 2, 60); added new employee to DEPTNO 2.	
T2		Update EMP (set DEPTNO = 2 where EMPNO = 100); transferred employee across departments.

Time	Session 1 Activity	Session 2 Activity
T3		Update DEPT for departments 1 and 2 since we modified a record in both departments. This statement blocks on read of EMP. Row inserted at time T1 is locked, and SQL Server blocks waiting for that lock to be released.
T4	Update DEPT for department 2 (department modified). This statement also blocks on read of row updated at time T2.	
T5	Server detects a deadlock condition, for both sessions 1 and 2. One is chosen as deadlock victim and rolled back—session 1 in this example. Forced rollback.	Statement becomes unblocked.
T6		Commit transaction.

Here, SQL Server, due to its locking and concurrency control mechanism, will not permit these transactions to execute concurrently. Only one of the transactions will execute; the other will be rolled back, and the answer will be “correct” in SQL Server. So, this must be a bug in Oracle, right? No, not at all.

Oracle has a feature called multiversioning and read consistency (a feature that sets Oracle apart from most other relational databases). The way it works is different from how SQL Server works (I would say superior, offering more highly concurrent, correct answers without the wait, but that’s beside the point here.) Due to Oracle’s concurrency model, the documented way in which it treats data, the second session would not see (would not read) any data that was changed since its statement (the update) began. Hence, the update would never see that extra record that was inserted. If the interleaving of the transactions had been just a bit different, the results could have been different as well. (You can read about Oracle’s multiversioning and read consistency feature in the *Concepts Guide*.)

The moral to this story is that the underlying fundamental concurrency and consistency models of the various relational databases are radically different. A sequence of statements in one database can, and sometimes will, result in a different outcome in a different database. It is the “sometimes” part of that statement that is troublesome. These sorts of data-integrity issues are hard to spot, unless you’ve actually mastered your database and read its documentation. And, they are even harder to debug after the fact. Reading a piece of code, it would not be intuitive that the condition described in this example may exist. If you implement your transactions for the target database and don’t expect a transaction from one database to work exactly the same on another database, you’ll be safe from issues like this.

If the Oracle development team members don’t understand how Oracle implements its concurrency and consistency mechanism, or even worse, if they assume it works like the mechanism in SQL Server or DB2 works, the most likely outcome is corrupt data, incorrect analysis, and incorrect answers.

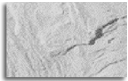
Inability to Quickly Deliver Software

It takes less time to write a database application when you make full use of the database and its feature set. Take that earlier example of using analytic functions (in the “Inability to Perform” section). Not only did it perform in a fraction of the time of the generic example, it took a fraction of the time to develop it! Trying to answer the stated question without using the analytic functions was hard. If I had to use a temporary table to get the results bit by bit, that would have taken quite a bit of procedural code and a lot of time to write.

As another example, suppose you need to provide a feature in your application whereby all changes made are audited. The history of a row from start to finish must be maintained in the database. You have two choices:

- Design, write, debug, and then maintain your own implementation.
- Use a single database command to enable the same functionality.

How quickly do you think you could do the first choice? What if you need to do it for 50 different tables? You would need to type in all of the code to do it yourself. Then you would need to add in the time for testing, debugging, and maintaining it as well.



NOTE

The feature being described here is a generic solution that may work well for you—then again, its generic implementation may have certain performance characteristics that would preclude it from working in your system. As we’ll stress throughout this book you must test and benchmark this (and any) feature before just “turning it on.”

Using the second choice, it would take about a minute to implement auditing for all 50 tables:

```
ops$tkyte@ORA920> create table emp
  2  as
  3  select empno, ename, sal, comm
  4  from scott.emp;
Table created.

ops$tkyte@ORA920> alter table emp
  2  add constraint emp_pk
  3  primary key(empno);
Table altered.

ops$tkyte@ORA920>
ops$tkyte@ORA920> begin
  2      dbms_wm.EnableVersioning
  3      ( 'EMP', 'VIEW_WO_OVERWRITE' );
  4  end;
  5  /
PL/SQL procedure successfully completed.
```

22 Effective Oracle by Design

That is it! One statement, part of the built-in Workspace Manager database feature, did it all. (See the Oracle *Application Developers Guide – Workspace Manager* for complete details on this database feature, which does a lot more than shown here.) Let’s look at some transactions executed against the EMP table after executing this implementation:

```
ops$tkyte@ORA920> update emp set sal = 5000
  2   where ename = 'KING';
1 row updated.

ops$tkyte@ORA920> commit;
Commit complete.

ops$tkyte@ORA920> update emp set comm = 4000
  2   where ename = 'KING';
1 row updated.

ops$tkyte@ORA920> commit;
Commit complete.

ops$tkyte@ORA920> delete from emp
  2   where ename = 'KING';
1 row deleted.

ops$tkyte@ORA920> commit;
Commit complete.
```

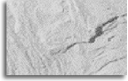
And now we can review the end results. The Workspace Manager built a series of views, one of which is the EMP_HIST view that contains the row-level history of each record. Here, we can see the type of change (insert, update, or delete), as well as when the record was created and retired (either modified or deleted):

```
ops$tkyte@ORA920> select ename, sal, comm, user_name,
  2      type_of_change, createtime,
  3      retiretime
  4  from emp_hist
  5  where ename = 'KING'
  6  order by createtime;
```

ENAME	SAL	COMM	USER_NAME	T	CREATETIM	RETIRETIM
KING	5000		OPS\$TKYTE	I	08-JUN-03	08-JUN-03
KING	5000	4000	OPS\$TKYTE	D	08-JUN-03	
KING	5000		OPS\$TKYTE	U	08-JUN-03	08-JUN-03
KING	5000	4000	OPS\$TKYTE	U	08-JUN-03	08-JUN-03

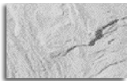
Querying from the EMP_HIST view that the Workspace Manager set up, you can see the history of each and every row in my table. There is another view, EMP, you could query that would appear to have only the current version of the row. I know you thought you had a table EMP, but the Workspace Manager actually renamed your table to EMP_LT and created a view EMP for you to use.

This view hides the additional structures the Workspace Manager added to version enable your data. You could enable this versioning and auditing for an existing application, without changing the application or altering the way it works. And you can do it rapidly.



CAUTION

As with any change, you must understand what is happening under the covers. You must understand the implications and limitations of the implementation. Workspace Manager has great functionality, but it imposes its own limitations and restrictions. Before turning on this feature for every database table in your application, make sure to read about it, understand it, and test it thoroughly!



NOTE

In order to drop the EMP table created in this example, you must disable versioning on the table first. That is accomplished via: “begin DBMS_WM.DisableVersioning ('EMP'); end;” in SQLPlus

Inability to Maximize Your Investment

Our goals as an Oracle development team are to deliver fast, functional, scalable database applications to our users rapidly and for the lowest cost. To achieve these goals, you need to maximize the use of each and every piece of software you purchased, from the operating system on up. You'll need to avoid the black box syndrome to do that.

- You want fast applications, so use the database features like analytics.
- You want applications fast, so use the database features like Workspace Manager.
- You want applications for the least development cost, so use the appropriate database features to avoid doing it yourself.

It's a Database, Not a Data Dump

This section is for everyone who feels that constraints should be verified in the client or middle tier, foreign keys just make the database slow, and primary keys are a nuisance.

“We have a City table, consisting of different cities where our client's offices are located. We have a VB form for inserting employee-related details. We have foreign key on the City column in the Employee table, whose parent key is the City table. One of our consultants recommended that we discard this check for the validity of city entered and suggested that we maintain all city validation checking through front-end coding. The reasons cited were that referential integrity checking at the back-end would be too time-consuming and would slow down the data-entry processing jobs. I wasn't truly convinced by his reasoning. Is his argument valid?”

My recommendation was to get rid of that consultant as fast as humanly possible.

Use Primary and Foreign Keys

There are quite a few reasons why getting rid of foreign key and database validation checks is the worst approach in the world. Here are three questions to ask if someone suggests getting rid of them:

- Is this the only application that will be accessing this data forever?

That is a rhetorical question. History proves time and time again that the answer is a resounding no. This data will be reused in many applications by many development teams (otherwise, it would be pretty useless data). If you hide all of the rules, especially fundamental ones like primary or foreign keys, deep inside the application, what happens in two years when someone else starts using this data? How will that application be prevented from corrupting your database? What happens to your existing applications when they start to query and join data and there is no matching primary key? Suddenly, your application breaks, their application is wrong, and the data is corrupt. The Internet revolution/evolution we all just went through should prove this point in spades. What were the last systems to get moved over to Internet-based ones in your company? Most likely, it was that set of applications that had 100% of the data logic embedded in the client.

- How could doing this processing be faster on the client?

The front-end must make a round-trip to the database in order to do this. Also, if the front-end tries to cache this information, it will only succeed in logically corrupting the data! There may be something to be said for caching on the client *and* on the server, but not just on the client. The end users' experience may be nicer if, on tabbing out of a field, they were notified, "Sorry, missing or invalid foreign key." But that does not mean that you can permanently move this check from the database *to* the client. Sure, you can replicate it, but move it? Never.

- Are the consultants who are advising getting rid of the keys paid by line of code?

This could be true—the more they write, the more they make, the more they have to maintain, and the more they have to debug. As the earlier example of auditing showed (in the "Inability to Quickly Deliver Software " section), sometimes you can do in one line of code that uses a database feature what might take dozens or hundreds of lines of code otherwise. If the database does something, the odds are that it does it better, faster, and cheaper than you could do it yourself.

Test the Overhead of Referential Integrity

To test the performance cost of using referential integrity, we can create a small CITIES table using the ALL_USERS data dictionary table for sample data. Include the requisite primary key constraint on this table. Then create two child tables, which are reliant on the data in the CITIES table. Table T1 includes the declarative foreign key. Oracle will not permit a row in this table unless a matching row exists in the parent CITIES table. The other table has no such constraint; it is up to the application to enforce data integrity.

```
ops$tkyte@ORA920> create table cities
2 as
3 select username city
4   from all_users
5  where rownum<=37;
```

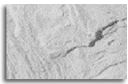
Table created.

```
ops$tkyte@ORA920> alter table cities
  2  add constraint
  3  cities_pk primary key(city);
Table altered.
```

```
ops$tkyte@ORA920>
ops$tkyte@ORA920> create table with_ri
  2  ( x      char(80),
  3      city references cities
  4  );
Table created.
```

```
ops$tkyte@ORA920> create table without_ri
  2  ( x      char(80),
  3      city varchar2(30)
  4  );
Table created.
```

Now, we are ready to test. We'll use the built-in SQL_TRACE capability of the database and TKPROF to analyze the results.



NOTE

TKPROF is part of a SQL Profiling tool built into Oracle. It is an invaluable performance tool. See Chapter 2 for more details on using SQL_TRACE and TKPROF if you are not familiar with them.

The benchmark will test the efficiency of single-row inserts into both tables using a simple loop. We'll insert 37,000 rows into each table.



```
ops$tkyte@ORA920> alter session set sql_trace=true;
Session altered.

ops$tkyte@ORA920> declare
  2      type array is table of varchar2(30) index by binary_integer;
  3      l_data array;
  4  begin
  5      select * BULK COLLECT into l_data from cities;
  6      for i in 1 .. 1000
  7      loop
  8          for j in 1 .. l_data.count
  9          loop
 10              insert into with_ri
 11                  values ('x', l_data(j) );
 12              insert into without_ri
 13                  values ('x', l_data(j) );
 14          end loop;
```

26 Effective Oracle by Design

```
15      end loop;
16  end;
17  /
```

PL/SQL procedure successfully completed.

Now, let's review the TKPROF report resulting from this:



```
INSERT into with_ri values ('x', :b1 )
```

call	count	cpu	elapsed	disk	query	current	rows
-----	-----	-----	-----	-----	-----	-----	-----
Parse	1	0.00	0.02	0	2	0	0
Execute	37000	9.49	13.51	0	566	78873	37000
Fetch	0	0.00	0.00	0	0	0	0
-----	-----	-----	-----	-----	-----	-----	-----
total	37001	9.50	13.53	0	568	78873	37000

```
*****
INSERT into without_ri values ('x', :b1 )
```

call	count	cpu	elapsed	disk	query	current	rows
-----	-----	-----	-----	-----	-----	-----	-----
Parse	1	0.00	0.03	0	0	0	0
Execute	37000	8.07	12.25	0	567	41882	37000
Fetch	0	0.00	0.00	0	0	0	0
-----	-----	-----	-----	-----	-----	-----	-----
total	37001	8.07	12.29	0	567	41882	37000

As we discovered, for 37,000 single row inserts, we used 0.000256 CPU seconds per row (9.50/3700) with referential integrity. Without referential integrity we used 0.000218 CPU seconds per row. Will your human being end users realize you have imposed this whopping 0.00004 CPU second penalty on them?

All told, the declarative referential integrity in the database added maybe 10% to 15% overhead. For that, you get the peace of mind that lets you sleep at night knowing the integrity of your data is protected and you used the fastest way to develop the application. You know that no matter what new application is added to the system, it will encounter this rule and will not be able to violate it. This same principle works on a much grander scale, beyond simple primary and foreign keys.

Middle Tier Checking Is Not a Panacea

Here is an idea I hear espoused often these days: Use the application's middle tier for doing work such as performing data verification and checking security. Using the middle tier might sound great. The benefits appear to be that it makes the application faster, more flexible, database independent, and secure. But is it? Let's take a closer look at each of these claims.

"We have some consultants building an application for us. They will have an Oracle database that contains only tables, views, and indexes. Most of the work, such as constraint checking, will be in the middle tier. According to them, this makes the application faster.

Also, they say it makes the application more flexible, able to use different databases because most of the code is in the application. Last, the security checking is done at the application level (middle tier), and they have their own auditing feature that creates their own tables in the Oracle database.”

In short, I would say these people will end up with an application that performs slower than it could, takes much longer to develop than it should, and has much more code to maintain than necessary. It will not be database-independent or flexible. In fact, it will be a way to lock the customer into using this consulting firm forever. To top it off, it will have security that is far too easy to get around.

Is It Faster?

If all of the constraint checking is done in the middle tier, typically in Java, this will be faster, they say. Well, that means in order to load data, they must write a loader (one that can do constraint checking). Will their loader be faster than the native direct path load? Their constraints will be checked in Java. Will that be faster than native database code in C? (I would take them on in that race, for sure.) If they have a parent table with a million rows and a child table with ten million rows, and they need to enforce referential integrity, what will be faster: querying the database over the network to check and lock the parent row, or letting the database perform this check upon insertion? That’s a rhetorical question, there will be no contest. Their application will be many times slower performing checks like that.

I would say their claim to faster would need to be backed up with real statistics, from real case studies. I’ve never seen one yet. However, I have seen a lot of cases where the converse has been shown: Doing it yourself is much slower than doing it in the database. Again, consider the auditing example described earlier (in the “Inability to Quickly Deliver Software” section).

Is It More Flexible?

I don’t know about you, but if I needed to alter a constraint—say *X* in table *T* must be changed from between 25 and 100 to between 50 and 100—I would find doing it in the database a bit more flexible. I can do this in two commands: one to add a constraint and one to drop the old constraint. The database will verify the existing data for me, and even report exceptions so I can fix them. If you choose to do this in their middle tier, you must edit the procedural code in many cases and read all of the data out of the database to verify it, bringing it back over the network.

Even if you make this rule “table-driven” or “parameter-driven,” it is no easier than updating a constraint in the database itself. You still need to validate the data. What if the data is known to be valid? Will the fact the constraint isn’t done by the database make it more efficient to implement, since you don’t need to check each row of data? Well, the database can do that as well. You can enable the constraint in the database without validating the underlying data, meaning the database doesn’t need to check the existing data unless you want it to.

Doing constraint checking outside the database does not add flexibility. There is a ton of code to be written, but I don’t see that as being flexible. This sounds more like the application designers don’t know how to use the database.

In addition, this actually makes the entire database less flexible, less able to support the needs of this company over time. Why? Well, the only way to the database is through this application. Does this application accept ad-hoc SQL queries, or does it do only what the developers programmed it to do? Of course it will not accept ad-hoc SQL. It will not be extensible by the customers; it will not be flexible as their needs change over time.

Take this argument back in time five or six years, and change *middle tier* to *client application*. Do you have a custom-developed piece of code written in 1996 that is client/server-based and has all of the business rules cleverly hidden inside it? If you do, you probably are still looking for ways to get out of that situation. Don't do it again with a new technology (application servers). Rest assured that in two years, some new, cool development paradigm will arise. If your data is locked up in some application, it won't be very useful or flexible at that point.

Another consideration is that if all of the security, relationships, caching, and so on are in the application, using third-party ad-hoc query tools cannot be permitted. The security (data filtering and access auditing) would be subverted if you access the data directly. The data integrity is in question (if the application caches). The data relationships are not known, so the query tool has no idea what goes with what in the database.

I've never seen a set of data that the business users are fully satisfied accessing *only* through the application. The set of questions they want to ask of the data is virtually infinite, and the developed application could not have anticipated all of those questions.

Is It Database-Independent?

Let's look at the claim that using the middle tier makes the database independent. It is a transactional system? If so, there are going to be issues with regards to concurrency controls and data consistency. If you didn't need subtly different code for different databases, why wouldn't PeopleSoft and SAP just use Open Database Connectivity (ODBC) or Java Database Connectivity (JDBC) to connect to everyone and be done with it?

Is it an application for a company, custom-coded to that organization? This company has a database, which cost real money. The company managers should not be paying someone else to rewrite the functionality of the database in a middle tier. That would be like getting taxed two times on the same paycheck.

Is It More Secure?

As for the security, to say that the data is secured because the application has good security is really missing the point. The further from the data security lies, the less secure the data is. Consider auditing, for example. If you have a DBA access the data directly, will that be audited? No, absolutely not. If you used the database to audit, you could do things like:

- Detect the DBA doing malicious things.
- If the DBA disabled auditing, you would be able to detect that she did that (and get rid of her).
- If the DBA mucked with the audit trail (not possible if you use operating system auditing and don't give the DBA operating system privileges), you can detect that (and get rid of her).

In short, you can detect that your data has been compromised and take corrective action. If the auditing takes place at the application layer, it is far too easy to get around it.

Build a Test Environment

I have had many conversations that went like this:

Them: “Our production application is behaving badly. It is doing *<something>*.”

Me: “When you ran it in test, what was the outcome there?”

Them: “Test? What is ‘test’?”

Me: “When you tested this process in your test environment, what happened there?”

Them: “Oh, we don’t have a test environment.”

Well, it just goes downhill from there. How many people are furiously mad at me personally because they did something in production I told them to try, without testing it first? I just assumed they would (I’m more careful about assumptions now). How many people have upgraded their database, for example, without testing it? How many people have upgraded their operating system without testing it?

If you do not have a test environment, you are just inviting disaster to lunch. There is a development environment, which most people have (although I have met some who do it on a production system, believe it or not). There is a production environment. There must also be a test environment. It is here that you will prove that your application works as advertised and that anything untoward is found before your end users are exposed to it. Specifically, you should use your test environment to do the following:

- Benchmark your application. Test that it scales to the required load (number of users). Test it for performance (that it can handle the load within required response times). This is key.
- Verify that your fixes actually work, especially in conjunction with the rest of the system, and that they do not cause more harm than good.
- Assure yourself that the upgrade script you worked on in development actually works. Make sure that you can, in fact, upgrade production with your latest releases.
- Make sure that major things like patch upgrades, major version releases, and operating system patches don’t knock out your system.

One of the things I hear frequently is “We can’t afford a test system.” Personally, I fail to understand how anyone can afford *not* to have a test system. If you add up the cost of your production system being down for an afternoon—just one afternoon—I’m sure that would more than pay for a test system, regardless of the size of your organization. If your organization is small, your test system should likewise be small and inexpensive. If your organization is large,

your system should be large and cost more. However, you have many more people who depend on it. Don't forget that cost is measured not only in financial terms, but also in terms of stress, customer anger, and your reputation (which may be irretrievably lost).

A realistic test system must be a fairly good mirror of your target system. Does it need to be exactly the same? No, but it should be as close as possible. The operating systems should be the same, at the same versions and patch levels. The databases should be exactly the same (well, the test systems database might run ahead since you'll *actually test* your upgrade there). The amount, type, and speed of storage should be the same, if at all possible.

Here are some points to bear in mind as you develop your test system:

- Use data that is representative of the data in your real system.
- Test with more than one user.
- Your test environment should be as close as possible to the actual environment in which your system will operate.

Let's take a closer look at each of these suggestions.

Test Against Representative Data

Testing against representative data is crucial. If you want to get a realistic feel for how the system will perform, your test system must be loaded with as much data as your real system. The query that works great in a test system with a thousand rows might become your worst nightmare when executed against a million rows in production.

Some people adopt the strategy of importing the statistics from their production system into their test system. They think they can get the optimizer to generate the plans that will be used in production and test using that data. The theory is that they don't need to actually have one million rows in the table; they can just tell the optimizer that there are one million rows. That approach will work only if you can read a query plan and be 100% confident that the plan is good and will give subsecond response times. I've looked at many query plans in my life, and I don't feel like I could make such a judgment call. Most people are striving to get query plans that use indexes all of the time, without realizing that as you scale up, indexes may not always be the best solution. The only way to determine that your plans will perform as expected is to test them against representative data.

This is not to say that DBMS_STATS with its ability to export and import statistics is not very useful. Quite the contrary—I've seen people use (with great success) the ability to import/export statistics, but not to *tune in test*, but rather to take the results of a statistics gathering done in test and import into production! Quite the reverse of what most people initially consider using DBMS_STATS for. These people will take a recent backup of their production system, restore it to test (for testing) and gather statistics there by using the test machines extra capacity to perform work so that the production system does not have to. Also, using DBMS_STATS to set statistics is useful to see exactly when certain thresholds will cause query plans to change over time as a learning tool then anything else.

Consider Using Data Subsetting

Do you need to load 100% of production data into your test system? Not necessarily. Many times, you can load some horizontal "slice" of it. For example, if your online transaction processing

(OLTP) system is using partitioning and you've ensured that all queries make use of partition elimination (for example, you set up partitioning so that each query will hit a single partition of a segment), you can get away with loading and testing just one or two partitions of each table. This is because you are now testing the queries against the same amount of data as they will hit in production. Partition elimination is removing the other partitions from consideration, just as they will be excluded from the query in your production environment.

Know the Optimizer Will Change Query Plans over Time

Using representative data will not ensure that your query plans in your production and test systems are identical, but you should not be expecting this anyway. Considerations such as the physical placement of data in the table affect how the optimizer develops query plans. For example, suppose we create two tables: one that is physically loaded in sorted order (so the rows are inserted into the table in order of the primary key values) and one that is randomly sorted. These two tables will contain the same data, but in different sorted order:

```
ops$tkyte@ORA920> create table clustered ( x int, data char(255) );
Table created.
ops$tkyte@ORA920> insert /*+ append */
  2   into clustered (x, data)
  3   select rownum, dbms_random.random
  4   from all_objects;
29315 rows created.
ops$tkyte@ORA920> alter table clustered
  2   add constraint clustered_pk primary key (x);
Table altered.
ops$tkyte@ORA920> analyze table clustered compute statistics;
Table analyzed.
ops$tkyte@ORA920> create table non_clustered ( x int, data char(255) );
Table created.
ops$tkyte@ORA920> insert /*+ append */
  2   into non_clustered (x, data)
  3   select x, data
  4   from clustered
  5   ORDER BY data;
29315 rows created.
ops$tkyte@ORA920> alter table non_clustered
  2   add constraint non_clustered_pk primary key (x);
Table altered.
ops$tkyte@ORA920> analyze table non_clustered compute statistics;
Table analyzed.
```

Arguably, the CLUSTERED and NON_CLUSTERED tables are identical, except for the physical order of their rows on disk. One is sorted by its primary key, and the other is not. The optimizer is aware of this via the CLUSTERING_FACTOR:

```
ops$tkyte@ORA920> select index_name, clustering_factor
  2   from user_indexes
  3   where index_name like '%CLUSTERED_PK';
```

32 Effective Oracle by Design

```
INDEX_NAME                                CLUSTERING_FACTOR
-----
CLUSTERED_PK                             1106
NON_CLUSTERED_PK                         29291
ops$tkyte@ORA920> show parameter optimizer_index
NAME                                TYPE                                VALUE
-----
optimizer_index_caching             integer                             0
optimizer_index_cost_adj            integer                             100
ops$tkyte@ORA920> set autotrace traceonly explain
ops$tkyte@ORA920> select * from clustered where x between 50 and 2750;
Execution Plan
-----
 0          SELECT STATEMENT Optimizer=CHOOSE (Cost=109 Card=2702 ...)
 1    0      TABLE ACCESS (BY INDEX ROWID) OF 'CLUSTERED' (Cost=109 Card=2702
 2      1      INDEX (RANGE SCAN) OF 'CLUSTERED_PK' (UNIQUE) (Cost=7 Card=2702)
ops$tkyte@ORA920> select * from non_clustered where x between 50 and 2750;
Execution Plan
-----
 0          SELECT STATEMENT Optimizer=CHOOSE (Cost=109 Card=2702 ...)
 1    0      TABLE ACCESS (FULL) OF 'NON_CLUSTERED' (Cost=109 Card=2702 ...)
```

The point to take from this example is not that the optimizer made a mistake. Rather, realize that the optimizer can and will change query plans over time. That is part of its normal functioning—what it does by design. Here, the optimizer correctly chose a full-table scan when the data was scattered (when the table was not sorted by primary key). It is the correct plan. What might happen if you tested this example on a smaller database, one that was not scaled up? You might have forced the use of an index in the misguided belief that indexes are good and full-table scans are bad. It would work great on your tiny database but fail miserably in production against the live data. Here, if I were trying to tune against a small database and forced an index to avoid the evil full-table scan, I would have ruined performance in the production system.

NOTE

If you run this example, you may well discover that the results are not exactly the same on your system. For example, you may discover both queries FULL SCAN the table or both use INDEXES. This will be a function of many variables as well see in Chapter 6. Individual settings such as the `db_file_multiblock_read_count`, `db_block_size`, and even the number of objects in your database that have `OBJECT_ID` values in the requested range may affect your outcome. If you play with the range of values used in the predicate (50 and 2,750 in this example), you will be able to see the different plans.

So, in short, you won't know all of the query plans perhaps, but you'll have the peace of mind knowing that the code was tested fully against real, representative data. The probability of things going smoothly in production is greatly enhanced.

Don't Test with a Single User

A common mistake is to incorrectly or improperly design applications in a transactional environment. Not understanding how the database works with regards to multiple users accessing data is a sure path to disaster.

"We are doing a stress test with our application developed in Java. The test hangs at the Oracle database. When I looked at the database, it seems one process is blocking another. For example, SID 55 was blocking 50. When I killed SID 55, SID 50 run but blocked SID 35, and so on. Before the stress test, and when we ran application by itself, it worked fine. That means SQL statements are committed properly. But somehow when multiple processes access the database resource, it breaks."

Fortunately, they caught this in their test environment! Unfortunately, they apparently didn't understand what they were observing, and their conclusion that SQL statements were being committed properly is 100% in conflict with the fact that they obviously have massive concurrency issues here. Their application, by design, is serializing (one after the other), and they are not performing the transactions correctly. They need to kill sessions to get the next one going, thus losing the work performed in the course of doing so.

You need to test your application under load, with many sessions concurrently accessing the data, in the way it will be in "real life". You need to test to scale. Another perspective on this point is to test with a realistic user base. A team of developers was working on a container-managed persistence (CMP)-based Java application. All of the code was generated from a tool, and they had no idea how it worked at the database level. Upon testing to scale, they quickly discovered that their generated application serialized immediately. They were doing exactly one transaction at a time, making for a very nonscalable system. It turned out that every user started by locking a single row in a single table, for instant serialization. A little application redesign with that in mind fixed the problem.

The point is that you want to catch things like this in the test system, not in production. If you want to lose credibility with your end users and customers, just roll code right from development into production. It will be the fastest way to accomplish that goal.

Don't Test in a Dust-Free Lab

If your real system will have 50 other things going on when your application is running, make sure that you have either accounted for them in some fashion or that you have them running on your test system.

I worked on an implementation where the test system had all of the components of the actual production system, except for one. An interface to an external system was "stubbed" out on the test system. The team measured how long it would take to call this external system (about 0.1 to 0.5 second maximum), and the stubbed-out test function just waited that long and returned a valid but made-up response. Maybe you can guess what happened: When it went into production, the system screeched to an immediate halt. This was not a problem with the database (a popular culprit); in fact, there was very little happening on the database server.

After days of working backward through the stack, the development team discovered that the external system interface was—you guessed it—serializing requests. This would not have been too bad had this not been a public web site, with high volumes of transactions hitting it every day in an unpredictable order. Every day at lunch, boom! The site would collapse, as potential customers sat at their desks, trying to order something from this site.

Your test environment should mirror reality—every bit and every piece. You don't want to air your dirty laundry in public, and the surest way to do that is to roll out a system tested in pristine conditions.

Design to Perform; Don't Tune to Perform

This heading is quite multifaceted—the “tune” could refer to a SQL query or to the data model. The true message of this section is that if your data model is poorly designed, all the SQL tuning in the world isn't going to help you. You will simply end up redesigning your schema for version 2. After reading this section, you might consider looking forward to Chapter 7, where we'll talk about the “physics” behind schema design.

Don't Use Generic Data Models

Frequently, I see applications built on a generic data model for “maximum flexibility” or applications built in ways that prohibit performance. Many times, these are one and the same thing! For example, it is well known you can represent any object in a database using just four tables:

```
Create table objects ( oid int primary key, name varchar2(255) );
Create table attributes
( attrId int primary key, attrName varchar2(255),
  datatype varchar2(25) );
Create table object_Attributes
( oid int, attrId int, value varchar2(4000),
  primary key(oid,attrId) );
Create table Links ( oid1 int, oid2 int,
  primary key (oid1, oid2) );
```

That's it. No more CREATE TABLE for me! I can fill the ATTRIBUTES table with rows like this:

```
insert into attributes values ( 1, 'DATE_OF_BIRTH', 'DATE' );
insert into attributes values ( 2, 'FIRST_NAME', 'STRING' );
insert into attributes values ( 3, 'LAST_NAME', 'STRING' );
commit;
```

And now I'm ready to create a PERSON record:

```
insert into objects values ( 1, 'PERSON' );
insert into object_Attributes values( 1, 1, '15-mar-1965' );
insert into object_Attributes values( 1, 2, 'Thomas' );
insert into object_Attributes values( 1, 3, 'Kyte' );
commit;
insert into objects values ( 2, 'PERSON' );
insert into object_Attributes values( 2, 1, '21-oct-1968' );
```

```
insert into object_Attributes values( 2, 2, 'John' );
insert into object_Attributes values( 2, 3, 'Smith' );
commit;
```

And since I'm good at SQL, I can even query this record to get the FIRST_NAME and LAST_NAME of all PERSON records:

```
ops$tkyte@ORA920> select
    max( decode(attrName, 'FIRST_NAME', value, null ) ) first_name,
  2  max( decode( attrName, 'LAST_NAME',  value, null ) ) last_name
  3  from objects, object_attributes, attributes
  4  where attributes.attrName in ( 'FIRST_NAME', 'LAST_NAME' )
  5     and object_attributes.attrId = attributes.attrId
  6     and object_attributes.oid = objects.oid
  7     and objects.name = 'PERSON'
  8  group by objects.oid
  9  /
```

FIRST_NAME	LAST_NAME
Thomas	Kyte
John	Smith

Looks great, right? I don't need to create tables anymore, because I can add columns at the drop of a hat (with an insert into the ATTRIBUTES table). The developers can do whatever they want, and the DBA can't stop them. This is ultimate flexibility. I've seen people try to build entire systems based on this model.

But how does this model perform? Miserably, terribly, and horribly. A simple select first_name, last_name from person query is transformed into a three-table join with aggregates and all. Furthermore, if the attributes are NULLABLE—that is, there might not be a row in OBJECT_ATTRIBUTES for some attributes—you may need to use an outer join instead of just joining, which might remove more optimal query plans from consideration.

Writing queries with this model might look straightforward. For example, if I wanted to get everyone who was born in March or has the last name of Smith, I could simply take the query to get the FIRST_NAME and LAST_NAME of all PERSON records and wrap an inline view around it:

```
ops$tkyte@ORA920> select *
  2  from (
  3  select
    max(decode(attrName, 'FIRST_NAME', value, null)) first_name,
  4  max(decode(attrName, 'LAST_NAME',  value, null)) last_name,
  5  max(decode(attrName, 'DATE_OF_BIRTH', value, null))
                                     date_of_birth
  6  from objects, object_attributes, attributes
  7  where attributes.attrName in ( 'FIRST_NAME',
                                   'LAST_NAME', 'DATE_OF_BIRTH' )
  8     and object_attributes.attrId = attributes.attrId
  9     and object_attributes.oid = objects.oid
 10     and objects.name = 'PERSON'
```

36 Effective Oracle by Design

```
11  group by objects.oid
12  )
13  where last_name = 'Smith'
14         or date_of_birth like '%-mar-%'
15  /
```

FIRST_NAME	LAST_NAME	DATE_OF_BIRTH
Thomas	Kyte	15-mar-1965
John	Smith	21-oct-1968

So, it looks easy to query, but think about the performance! If you had a couple thousand OBJECT records and a couple tens of thousands of OBJECT_ATTRIBUTES, Oracle would need to process the entire inner group by query first and then apply the WHERE clause.

This is not a made-up data model, one that I crafted just to make a point. This is an actual data model that I've seen people try to use. Their goal is ultimate flexibility. They don't know what OBJECTS they need, and they don't know what ATTRIBUTES they will have. Well, that is what the database was written for in the first place: Oracle implemented this thing called SQL to define OBJECTS and ATTRIBUTES and lets you use SQL to query them. People who use a data model like this are trying to put a generic layer on top of a generic layer, and it fails each and every time, except in the most trivial of applications.

"I have a table with a BLOB field, for example:

```
Create table trx (  trxId Number(18),
                   trxType Varchar2(20),  objValue      Blob )
```

BLOB fields contain a Java-serialized object, different objects based on types, though all of them implement the same interface. We have always accessed this object through a J2EE container, so it works fine, so far. Now, users want to use reports using SQL*Plus, Crystal Reports, etc. So they want a solution to this BLOB issue."

This is an interesting problem. Here, the application developers have taken this generic model one step further than I had ever imagined! Instead of having objects with attributes, they just "BLOB" all of the object attributes into an unreadable string of binary bits and bytes. In Java, *serialization* means to take a data structure and put it into a "flat" format, one that could be written to a file and read back later to populate that Java data structure. The formal definition from Sun's Java Development Kit (JDK) is:

Object Serialization extends the core Java Input/Output classes with support for objects. Object Serialization supports the encoding of objects, and the objects reachable from them, into a stream of bytes; and it supports the complementary reconstruction of the object graph from the stream. Serialization is used for lightweight persistence and for communication via sockets or Remote Method Invocation (RMI). The default encoding of objects protects private and transient data, and supports the evolution of the classes. A class may implement its own external encoding and is then solely responsible for the external format.

So, here the Java developers have decided to use the database as a big data dump, a bit bucket. They are storing their data in a database, but they are not using the database. They didn't want to be bothered with things like data models and other formalities. They just want to write some code and store some stuff. And now, the end users are clamoring at their door for their data! These guys wanted to know how we could efficiently parse these BLOBs apart in order to support some SQL access to the data. The answer is that it's just not possible to do this in an efficient, high-performance manner.

To both of these groups of people, I have only one reply: You need to be more specific and less generic. Sure, generic is flexible, but generic has lower performance, is harder to query, and is harder to maintain. There is no data dictionary, and there is no metadata. The people who do generic are doing it to cut corners. The maintenance of this code is nightmarish. And just try to have someone who didn't build it fix or enhance it! This approach also removes value from the data itself, because the data is wholly locked up in the application.

Design Your Data Model for Efficiency

You should design your data model to answer your most common queries as efficiently as possible. It should be clear by now that, unless you design your system to efficiently and effectively answer the questions your end users will be asking, your system will fail miserably.

"We have this query we need to run hundreds of times a minute and it takes forever, bringing our system to its knees, Please help!"

```
if type = 'A' join to tablea
if type = 'B' join to tableb
...
```

A query like this does conditional joining. (A similar problem is a query that needs to do *where exists* type processing four or five levels deep.) It is obvious that the data model was not set up to answer the question they ask hundreds of times per minute.

When I set about building a system, the first thing I want to figure out is how we can store the data in such a way that the most frequently executed, performance-critical, important queries are answered as fast as humanly possible. This takes place well before the tables are created, and then we create very specific tables to handle that requirement. We choose very specific indexing technologies. We design the system to perform from day one. Tuning will happen only because I made an incorrect assumption during the design and implementation, not because, "Well, you always tune after you deploy."

The tuning-after-deploying approach just doesn't work. Why? Because the system is already deployed. Can you change your physical structures *after* you've deployed system? Not really. Can you change the fundamental architecture after you've deployed the system? No, not really. Those changes require another version of the product. They are certainly not things you are going to be able to fix overnight.

The requests I walk away from are those that start with, "We are in production. The code cannot be changed. You are not allowed to bring the system down. But make it go faster. You

know, wave a wand or something and set `fast=true init.ora` on. It just doesn't happen like that, ever.

As an example, consider an internal, web-based calendar system my team worked on recently. This application was to be used by between 10,000 to 20,000 users on a daily basis, 24 hours a day, 7 days a week. In addition to the web interface, it had to support a Palm Pilot synchronization. Another group had been working on this project for a while, but had yet to bring it to fruition. So we were asked to take a stab at it (more to get them going than anything else).

We looked at the problem and decided that this system was 90% read; more than 9 out of 10 requests would be of the form "show me my appointments." The other 10% of the access would be writes: "create this one-time appointment," "create this repeating appointment," and so on. The most important query was "show my appointments where appointment date is between START and STOP," where START and STOP would be dictated by the view of the data, which would be a single day, a week, or a month.

Design for Writes or Reads?

So, how should this system be designed? To make writes super fast or to make reads super fast? The other team looked at the requirements and said, "We have to sync with a Palm Pilot. We'll just use their data model. It will make writing that sync routine easy."

We looked at them and said, "Geez, if we don't concentrate on a data model that supports more than 90% of our requests, it'll be really slow and not scale up very well. Besides, the Palm data model was built for a single-user device with a dedicated CPU and small amounts of RAM, which must also double as disk (persistent) storage. We are on a multiple-user device with tens of users per CPU and have virtually unlimited disk storage and gobs of RAM. Our design criteria are really different. We'll sync with the Palm, but we won't store data in our database in the same way the Palm would."

The problem we had with the Palm model was simple. In order to store a repeating appointment, it would store a single record. This record would have attributes like `START_DATE`, `DESCRIPTION`, `REPEAT_TYPE` (daily, weekly, or monthly), `INTERVAL` (for example, if repeat type were daily and interval were 2, it would repeat every OTHER day), and `END_DATE`. So, this single record would need to be procedurally "forecast out" each time a user wanted her calendar. For example, suppose that you had this entry:

```
Start date: 04-jan-2003
Description: Manager Meeting
Repeat_type: Weekly
Interval: 1
End date: 01-jan-2004
```

Then you asked to see your calendar for July 2003. The Palm would loop, starting from the `START_DATE`, adding one week while the result was less than July 2003. It would output the records for the month of July and stop when the result of adding a week took it past the end of July. It would then go to an `EXCEPTIONS` table to see if there were any exceptions for the month of July. For example, the 05-Jul-2003 meeting may have been canceled for the American Fourth of July holiday. It would be recorded as an exception and would be removed from the four meetings in July.

Now, that works fine for a Palm Pilot. When you are not using SQL as your query language, you own the CPU, and storage is at a premium, every byte you save is well worth your effort.

But, put that same record in a relational database and try to write a SQL query that will tell you that in July there are four meetings on the fifth, twelfth, nineteenth, and twenty-sixth, except that the meeting on the fifth was canceled. Although it can be done, it is extremely expensive and requires what I call “a trick.” The relational database hates to make up data, and that is what we would need to trick it into doing here if we used the Palm model. There is only one row. We would need to multiply it out, to synthesize these other rows. It is hard, slow, and unscalable.

On the other hand, it would be rather easy during the creation of this event to loop and insert a row for each occurrence. True, we would put in 52 (or more, if the duration were longer) rows, but you know what? Databases were born to store data. It is what they do best. And we have tons of storage on the server (as opposed to the Palm, with its extremely limited storage). Also, we would query this event many, many times more than we insert it (read-intensive; nine out of ten or more requests are to read data). If the rows exist, I can easily report back events in a given month. If they do not exist, if we used the Palm model, I would need to fetch all of the repeating events and forecast them out, procedurally generating the data.

Consider also that a web calendar would be used to coordinate people’s schedules for meetings and such. The end user asks a simple question like, “What time on Tuesday can Bob, Mary, George, and Sue get together?” If we used the Palm model (built and designed for single user, with extremely limited RAM and disk resources), we couldn’t use the database to answer the query, because the data doesn’t exist in the database. Only the information an application could use to generate the data exists. We would need to pull all the appointments for Bob, Mary, George, and Sue for that day, get all the repeating appointments, forecast them out, sort them, and then figure out what time period all of them had available.

On the other hand, if I had a table that just had start and stop times in it for each user, I could write a SQL query to tell me the available slots. This solution has every attribute we could want: It uses less code, requires less work for the database (it sends out less data, because it can run this query as efficiently as running the queries to return *all* of these users’ data), and it is easier to maintain.

Which Data Model Worked?

The other group decided to optimize writes at the expense of queries. They heard a rumor somewhere that writes are bad and slow, so write as little as possible. They used the Palm model to simplify the development of a Palm sync module. They wrote a lot of procedural code in order to forecast out the schedules.

They felt we were wrong and should use what they called “a normalized schema, like the Palm.” We felt they were really wrong and needed to study the database and how it works a little longer. Their major criticism of us, besides not using this standard Palm model, was that our database would be (they theorized) 9 terabytes (TB) in size. The fact is that no one ever made an appointment that repeated every day for ten years, let alone 1,000 of them.

Their solution didn’t quite “fly,” as they say. They found they could scale to about 6 to 12 concurrent users before they needed to add more application servers. Their approach to scaling was to throw more hardware at it. The database machine was feeding data to the application servers as fast as the servers could eat it, but there just weren’t enough CPU cycles on the application servers to keep the pages going. If you have the physical design fundamentally wrong, no amount of hardware, tuning, or tweaks—short of a reimplementation—will solve your problem.

Our solution was to use a data model that answered the most frequently asked questions and kept the writes to a minimum. We didn’t even order a separate application server machine. Instead, we used the single database server to host the database and the application server. On a fraction

of the hardware, we served pages faster than the other team ever dreamed. The reason was because we designed for performance. We did not choose a design because the code would be easier (heck, they never got to write the Palm sync module, because they spent all of their time coding the middle tier and trying to get it to go faster). We chose a model that made our application perform and scale well.

The Palm sync module was written, and coordinating the two different models wasn't that hard after all. The sync was just a big "bump and grind." We basically could take the contents of the Palm Pilot (we know that they are small), dump them in the database, merge the two versions of reality (sync them), and then send the changes back to the Palm.

The system has been quite successful. In the end, our database was 12.5GB after a year of use with more than 28,000 users and more than 12 million appointments. As the number of appointments has gone up, response times have stayed perfectly flat (after all, we are just querying with a "between A and B"). That someone has 5,000 appointments in total is not relevant (it would be on the Palm model—just try that sometime). A user has only so many appointments in a given day, week, or month, and that's all the database considers using its indexes.

The moral to this story is to make your data model fit your needs, not the other way around. If you want performance, design for it from day one. You won't try to (and won't need to) retrofit it in version 2.0.

Define Your Performance Goals from the Start

When you set out to build an application, it is vital that, right from the very start, you are working with definite, clearly defined metrics that characterize precisely how your application is expected to perform and scale. These metrics include your expected user load, number of transactions per second, acceptable response times, and so on. You can then go about collecting the performance data that will categorically prove (or disprove) that your application is achieving its goals. Many of the difficult situations I encounter arise because people have simply not defined these metrics.

Here are some typical scenarios related to this issue:

- **Make it go faster!** This is a result of not ever having set performance goals for the system. These people never laid down the law regarding how many users they were going to support, doing how many transactions per second, with response times in the x to y millisecond range. They just know their system is going too slow, but they have no idea what is fast enough.
- **Is my system tuned?** The only way to know if a system is tuned is to have some metrics and a goal to achieve. If you are meeting or exceeding that goal, your system is tuned. If not, you have some work to do.
- **Everyone says we should do it.** Some people do some maintenance operation on a scheduled basis; for example, they'll rebuild their indexes every week. When asked why, the response is, "Well, everyone knows you need to rebuild indexes, so we do." They've never measured to see if they are doing more harm than good, or even to see if they are doing any good at all. Odds are their work either has no impact or a negative impact more often than it has a positive impact. These sort of practices are able to perpetuate only because people generally do not set, collect, keep, or analyze hard numbers with regard to their system.

You must work to definite goals, and you must then collect the data that can give a definite measure of your success.

“I’ve got a question I would like to ask and am hoping you can answer or point me in the right direction. I need to generate some monthly reports for upper management to give them an indication of how the DBs are performing. I know there are loads of performance indicators that you can extract from Oracle. However, from my experience, upper-management people don’t care about the technical details. They just want something that’s pretty to look at and easy to understand. Which statistics do you think management need?”

This is a classic quandary. Upper management (whatever that is) just wants to be told everything is okay. They have no clearly defined goals or targets to meet. Their attitude is “just tell me my cache hit ratio is good or something,” or “show me a pretty graph.” In this context, such reports are useless in determining if you are meeting your goals.

Work to Clear, Specific Metrics

I have a theory that a system can always go 1% faster. So, in theory, we should be able to make our databases infinitely fast. Unfortunately, we cannot do this. The reasons are twofold. The first is pure math: You never quite get to zero; As you slice 1% off, it is 1% of a smaller number; hence, the 1% is smaller each time (the 1% performance gain is less and less each time). The other reason is that each 1% is much harder to achieve than the previous 1%. The more you tune, the harder it is to tune more. Every 1% (which is getting smaller) costs more to achieve than the previous 1% did.

So, what you need from the very beginning—way back in development—are hard and fast metrics upon which to benchmark yourself. If you are told you must be able to support 1,000 users, 100 of whom will be active concurrently, with response times measuring 0.25 second given a well-known transaction, you can actually design and size a system to do that. You can set up simulations to prove that you are meeting those benchmarks. You can actually feel 99.9% confident the day that system is turned on that it will meet those goals. And if the system you constructed doesn’t meet your requirements, you are also made painfully aware of that. But at least you know *before* you roll it out.

On the other hand, what if you are told you will have a lot of users, who will be doing some work (not sure what), and your system must be fast? Then you cannot possibly design, let alone size, a system. It is amazing how many people have “specs” like this, however. I’ve been on many a conference call in which the person on the other end wants to know why we cannot tell him how big a machine to buy given “a lot of users, want it fast, doing some transactions.” My only response to that is, “How much money do you have? Buy the biggest machine you can afford and hope for the best.”

Collect and Log Metrics over Time

If you keep a long-term history of specific metrics, you can judge over time how things are going. Statspack (covered in Chapter 2 of this book and fully documented in the Oracle *Performance Tuning Guide and Reference*) is an excellent way to do this. If only every site I walked into had the past six months of Statspack information waiting to be reviewed. If only they did a 15-minute snapshot every day, once or twice during their peak times. Then we would be able to identify

exactly when that query started performing badly (we would watch it rise up through the top SQL section of the report). Or we would be able to see not only when performance started to go down, but what wait event was causing it over time. Or we would see the effect that adding that new application had on our soft parse ratio, whether it was negative or positive.

Additionally, having some sample key transactions and their statistics recorded would be nice. If possible, you would set up a stored procedure, for example, that ran representative sample queries and even transactions that modified the data in the same way the end users did. You would run this job periodically throughout the day, taking a snapshot of performance over time. It would record the response times for these representative queries and transactions. Now, you could proactively monitor performance over time and even graph it. Not only would you have something to show “upper management,” but you would also be able to pinpoint exactly when performance started to go down and by how much. A combination of the monitoring results and Statspack reports would be invaluable when you ask someone to “make it go faster.”

Don’t Do It Because “Everyone Knows You Should”

More often than not, unquestioning acceptance of conventional wisdom does harm to a system as frequently as it helps it. The worst of all possible worlds has DBAs or developers doing something simply because “everyone knows you should.”

“Why does rebuilding an index cause increased redo log generation *after* the index has been built? I have a table with 35 million rows and an index (nothing is partitioned). Transactions against this table are constant. It’s always 500,000 rows per day. This generally creates 10 logs a day. Once a month, the indexes are rebuilt (`alter index rebuild`). On the day following the index rebuild, 50 logs are created. On the following days, 45 logs, then 40, 35, 30, 25, down to 10. At 10 logs, this remains constant. Mining the logs we see that we have increased INTERNAL INDEX UPDATES. Why does this happen? Is this always the case?”

It was interesting to me that the DBA identified the cause of a big performance issue and resource hog—the index rebuild—yet persisted in doing it! Like people, indexes have a certain size they tend to be. Some of us are chubby, some skinny, some tall, and some short. Sure, we can go on a diet, but we tend to gravitate back to the weight we were. The same is true for indexes. In this case, the index wanted to be wide and fat, and every month, the DBA rebuilt it (put it on a diet). It would spend the first half of the month getting fat again and generating gobs of redo due to the block splits it was undergoing to get there. In this case, rebuilding the index on the system had these effects:

- The system would generate five times the redo.
- The system would run slower.
- The system would consume more resources (CPU, I/O, latching, and so on).
- The system would not be able to handle the same user load.

These effects continued until the system got back to where it wanted to be. And then, the DBA would do it all over again! He would destroy the equilibrium that the system worked so hard to get to!

Here, the DBA luckily caught onto the extra redo log generation, and that concerned him enough to investigate further. Fortunately, the solution is rather straightforward: Stop rebuilding that index.

My favorite example is, “We rebuild indexes every week or month. Everyone knows you should do that on a regular basis.” If only they actually measured what they did and checked what happened after they did it. It is the rare index that needs to be rebuilt. It is not a rule that indexes need to be rebuilt. It is rare that any operation like this would need to be done on a regular schedule. It is rare to need to reorganize a table, yet I see people do it just as a matter of standard operating procedure.

Of course, there are extreme cases, particularly with regard to bitmapped indexes after mass data loads, where rebuilding is suggested. But in the day-to-day operation of a transactional system, the need to rebuild indexes is so rare that I’ve never actually done it myself in 15 years, except to move an index from one physical location to another. And we don’t even need to do that much today, because it’s much easier to use logical volumes and move things around outside the database.

The bottom line here is that if you keep hard-and-fast metrics (that Statspack report, for example), you will be able to quantitatively prove that what you are doing is good; that it has a positive effect. If index rebuilding is beneficial, then the day after an index rebuild should have a much better Statspack report than the day before, right? The buffer get statistics for that top query that uses that index should be much lower (and if that index wasn’t in the top queries, why was it so important to rebuild it anyway?), and the overall system performance should be measurably better. If the Statspack report shows nothing changed and your query response times are the same, the best you can say is that the rebuild was harmless. Or, you might find that the procedure is not harmless; it consumes resources, wastes time, and decreases performance until things get back to the way they were.

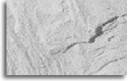
Benchmark, Benchmark, Benchmark

Many people think of benchmarking as something very big, and sometimes it is. To me, benchmarking comes in all shapes and sizes. Most of my benchmarks are done on a very small scale. I benchmark an idea or an approach. Then there is the benchmark I will do in order to see if our application designed for 10,000 concurrent users will actually work. These benchmarks can incur a large amount of time and energy. Both benchmarks are necessary.

A third type of benchmark is to have specific metrics from the beginning of the project, as we covered in the previous section. This is the benchmark in the form of a Statspack report or a custom procedure that measures typical response times over the course of the day for standard and common transactions in your system. This activity benchmarks your ongoing performance to ensure that the system continues to meet or exceed its design specifications over time.

In practice, I find that most people ignore all of these types of benchmarks. They never test their approaches to see how or even if they will scale. They never test their system to see if, after all of the bits and pieces are put together, if it will scale. Then when it fails, what do we hear? “Oh, the database is slow.” Yes, the database does appear slow. But that is a symptom, not the problem. The cause is typically a poorly written program or a poorly architected system that was never tested.

It is important to understand that a benchmark must model *your* reality. A benchmark must be designed, tested, and implemented using your specifications. Running a TPC-C on your hardware will not tell you how your application will perform, unless, of course, your company is built around the limits of a TPC-C benchmark (doubtful). A benchmark is a personal thing, which is unique to you and your application.



NOTE

For information on the Transaction Processing Councils standard suite of benchmarks, please see <http://www.tpc.org/>. The scope of the benchmark, as well as results, are available there.

Small-Time Benchmarking

In what I call “small-time benchmarking,” the Oracle team is interested in deciding between two different approaches to a problem, or someone on the team is trying to prove that a certain technique is infinitely more scalable than some other technique. For this type of benchmarking, you are proving quantitatively that approach A is superior, inferior, or the same as approach B.

These tests are best done in isolation; that is, on a single-user database. You will be measuring statistics and latching (locking) activity that result from your approaches. You do not wish for other sessions to contribute to the system’s load or latching while this is going on. A small test database is perfect for these sorts of tests. I frequently use my desktop PC or laptop, for example. I believe all developers should have a test database they control on which to try ideas, so they don’t need to ask a DBA to do it.

Use a Simple Test Harness

I perform small-time benchmarking so often that I have set up a test harness, which I call Runstats. It measures three things: wall clock (or elapsed) time, system statistics (such as parse calls, for example), and latching. The system statistics show, side by side, how many times each approach did something and the difference between the two. The latching information is the key output of this report.

Latches are a type of mutual exclusion, or locking mechanism. Mutual exclusion/locks are serialization devices. Serialization devices inhibit concurrency. Things that inhibit concurrency are less scalable, support less users, and require more resources. Our goal is to build scalable applications—ones that can service 1 user as well as 1,000 or 10,000 users. The less latching we incur in our approaches, the better off we are. I might choose an approach that takes marginally longer to run on the wall clock but that uses 10% of the latches. I know that the approach that uses fewer latches will scale much better than the approach that uses more latches.

So, what does Runstats entail? Well, I’ll defer a detailed discussion of the guts of Runstats until Chapter 2, when we look at the tools we need to use. Here, I will show the end results of Runstats, so you can see how important it is to do these sorts of benchmarks.

As an example, suppose that we want to benchmark the difference between two different approaches to inserting data into a table. We have a requirement to load data into some database table, but we won’t know which table until runtime (hence, static SQL is out of the question). One approach we take will use *bind variables*, which act as placeholders in a SQL statement and allow for the SQL statement to be used over and over again. The second approach we try will not use bind variables. For whatever reason, this second approach is widely used by many ODBC

and JDBC developers. We'll compare the two methods and analyze the results. We'll start by creating a test table:

```
ops$tkyte@ORA920> create table t ( x varchar2(30) );
Table created.
```

We begin by coding two routines for our two approaches. One routine uses dynamic SQL with bind variables. The other routine uses dynamic SQL without any bind variables. It concatenates the character string literal to be inserted into the SQL statement itself.

```
ops$tkyte@ORA920> declare
2     procedure method1( p_data in varchar2 )
3     is
4     begin
5         execute immediate
6             'insert into t(x) values(:x) '
7             using p_data;
8     end method1;
9
10    procedure method2( p_data in varchar2 )
11    is
12    begin
13        execute immediate
14            'insert into t(x) values( ''' ||
15            replace( p_data, ''', ''''' ) || ' ' )';
16    end method2;
```

First, notice that it is actually harder to code without using bind variables! We must be careful of special characters like quotation marks (*quotes* for short). In fact, the goal of the REPLACE function call in method2 is to make sure the quotes are doubled up, as they must be.

Now, we are ready to test our two different methods. We do this in simple loops. And the last thing we do is print the elapsed timings and finish the PL/SQL block:

```
17  begin
18      runstats_pkg.rs_start;
19      for i in 1 .. 10000
20      loop
21          method1( 'row ' || I );
22      end loop;
23      runstats_pkg.rs_middle;
24      for i in 1 .. 10000
25      loop
26          method2( 'row ' || I );
27      end loop;
28      runstats_pkg.rs_stop;
29  end;
30  /
884 hsecs
```

46 Effective Oracle by Design

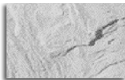
```
2394 hsecs
run 1 ran in 36.93% of the time
```

Now, we can start to see concrete evidence of which approach is superior. Already, we see `method1` with bind variables comes in at 37% of the time that `method2` does. Going further and looking at the statistics and latches used, we find even more evidence:

Name	Run1	Run2	Diff
...			
LATCH.row cache enqueue latch	72	40,096	40,024
LATCH.row cache objects	88	40,128	40,040
LATCH.library cache pin	60,166	108,563	48,397
LATCH.library cache pin alloca	116	78,490	78,374
LATCH.child cursor hash table	19	79,194	79,175
LATCH.shared pool	30,181	162,931	132,750
LATCH.library cache	60,363	249,568	189,205

PL/SQL procedure successfully completed.

As you can see, the method that didn't use bind variables used considerably more latches (remember that latches are locks of a type, and locks are scalability inhibitors). In fact, if you add them up, you will find that `method1` used about 275,000 latches and `method2` used about 884,000, for a difference of well over 600,000. That is huge! This conclusively proves that using bind variables is not only faster, but is also much more scalable. The more latching and locking going on in a system, the fewer users you have doing things concurrently. Latching issues cannot be fixed with more CPUs. Locking is not something you can fix with hardware. Rather, you need to remove the source of contention, and, in this case, we found it.



TIP
If you would like to see more case studies using Runstats, just go to <http://asktom.oracle.com/> and search for "run stats" (in quotes). As of the writing of this book, I had more than 101 such examples online.

Other Benchmarking Tools

The following are some other tools that you can use for small-time benchmarking:

- **TKPROF, TIMED_STATISTICS, and SQL_TRACE** Excellent methods to see exactly what your programs are doing and how well they are doing it.
- **DBMS_PROFILER** For fine-tuning PL/SQL code.
- **Explain Plan** To see what the query is going to do.
- **Autotrace** To see what the query actually did.

We will explore these tools in Chapter 2.

Big-Time Benchmarking

By “big-time benchmarking,” I mean that you are testing the system as a whole; you are testing to scale. There is no other way to know whether the system that works great with 5 users will work just as well with 1,000 users. There is no other way to know whether the system that works great with 5,000 rows will work just as well with 500,000 rows. Unless you want to look really bad on the opening day for your system, you will benchmark. You will simulate. You will do it realistically. You will spend the money, time, and energy for it (and it will cost some of each). It is not easy to do, but unless you really just want to throw something at the wall and see if it sticks, you will do big-time benchmarking.

This one step is key to success; it is crucial. And this is the one step the large preponderance of Oracle development teams skip entirely. Time after time, I see systems that are lacking because of situations like one of these:

- They have never been tested.
- They have been tested with unrealistic numbers of users. (Sure, we tested with 10 users, and it works fine; turn it on for the 1,000 real ones now.)
- They have been tested with tiny databases. (It’s really fast with one row, so you can ship it.)

Test with Representative Amounts of Data

One example will clarify the need to test with a realistic amount of data. I was part of a team that was a beta test site for a product (that will remain nameless). We installed it, set it up, and turned it on. It was about the slowest thing on the planet. It took 30 to 45 seconds for the initial page to come up, and every page after that was the same. We did the old export/import of our database schema to their machines, and the developers could not reproduce the problem.

After a little TKPROF work on our own, we quickly discovered the problem. Our database had more than 40,000 named accounts in it, and their database contained 10 accounts. The queries they had written against the data dictionary to check privileges worked great with 10 users, but fell apart fast with 40,000 users. They had just never bothered to test with more than one user account on their system. So, while they benchmarked with “thousands of sessions,” all of the sessions were running as the same user! Their benchmark tests were unrealistic; they did not match the real-world situations in which their application would be used. After the developers created a couple of thousand accounts, they found, and subsequently fixed, their performance issues.

Test with Realistic Inputs

Another benchmark I was pulled into concerned a three-tier, Java-based application. The middle tier was code generated from a piece of software. The developers were disappointed in the performance they were receiving. The problem turned out not to be the database, or even the application itself ultimately. It was an ill-designed benchmark. It was easier to script with a single-user account, so that is what they did. They did not realize that the first line of SQL code in each container was:

```
SELECT * FROM USER_TABLE WHERE USERNAME = :X FOR UPDATE
```

The container was serializing transactions at the user level. They had one user using the system (heavily), so that user was waiting in a really long line to perform the transaction. This is similar to the issue described in the previous section, but instead of being a query performance issue, it was a performance issue inflicted due to concurrency controls gone awry. And they had gone awry because of an impossible real-world situation: a single user having hundreds of transactions going on concurrently!

By spending more time on the benchmarking aspect and setting up a realistic test, the developers achieved their goal. The scary part is that this particular example is a two-way street. Fortunately for this group, the benchmark returned disappointing results, so the developers looked deeper. As shown in the previous example of using a single user in the tests, however, it would be just as easy to feed the system unrealistic inputs that would cause it to run faster and scale better than it could in reality. It is important to ensure you are testing reality.

Make Sure You Verify the Results

I was called onsite to help a customer resolve severe performance issues in a production system. The system hardly ran at all. It became the production system after a successful benchmark “proved” it could scale to more than 10,000 users. But now the production system was failing with fewer than 200 concurrent users. What happened?

As it turns out, the numbers the developers got back from the benchmark were wholly unrealistic. The response times reported by the tool they were using just didn’t reflect numbers that were possible. Every screen, every web page, had 0.01 to 0.05 second response times. This system could scale infinitely according to the benchmark. They stopped at 10,000 users, but they could have gone higher.

What the developers didn’t check was whether the benchmark process was working. Actually, every page was returning the infamous “404 – Not found” error. They didn’t know this, because they never checked. They didn’t record any metrics (so they had no idea if transactions were being performed). They didn’t instrument their code (see the next section for more on that topic), so they didn’t know it was not even really being run. They just thought, “What a fine system we built!”

When we fixed the benchmark, it became painfully obvious what was happening. Their system did not use bind variables! If you refer to the previous section about small benchmarks, you will understand what happened. As you add more users, you add more contention; more contention means more waiting; more waiting means slower response times. As they added users, the system spent most of its time keeping them in line, waiting to parse SQL. This was a really huge machine we were running on—a 48 CPU box. But no amount of additional CPUs would help them here. The problem was the long lines waiting to get into the shared pool. After two weeks of really long days finding and fixing code, the system was back up (up for the first time, actually).

The users had no confidence in the development team after this. They had totally lost any respect for them. For weeks, the end users were sitting staring at screens that didn’t work. In this respect, it’s worth noting that from a user perspective, performance is no different from functionality. A lot of developers will say “it works, but it’s slow,” whereas the user perspective (the only correct one after all) is that “it does not work, because it’s slow.” A mistake like this will take a long time to get over. For those in charge, it can be a career-changing event; that is, they may need to find a new one.

Don't View Benchmarking As a Chore

Benchmarking is one of the most important steps in development, beyond getting the design right (designing for performance). This is the step where you get to prove that your design is sound (or to be shown otherwise). It is your chance to do this without incurring the wrath and scorn of the end users, your customers. I would much rather fail in a small room in front of a few people and have an opportunity to fix the problem than to fail in front of a crowd of hundreds (or thousands), and then scramble to try and patch it as quickly as possible.

If you develop real stuff, benchmark it. If you aren't benchmarking, ask your manager why the work you do isn't important enough to actually test. If you cannot afford to test it before inflicting it on your customers, is it really that important to them?

Not only will you avoid embarrassment by benchmarking, you'll save time and money as well. How is that? Doesn't it take time and money to conduct these tests? Certainly, it can take a lot of time and money actually, but consider the cost of having 1,000 people sitting on their hands for two weeks while you "fix" the system? How much time was that? What is the cost of not only those 1,000 people, but also the people they service? For example, if it's a system to order products, not only are 1,000 people prevented from working, but the work they are not doing is the way that the company made money in the first place. The cost is huge.

Do not let a tight schedule at the end of development force you to skip the benchmarking process. The pain of trying to fix a production system that was not tested to scale far outweighs the pain of pushing the deployment schedule back a week or two (or longer if the test fails!).

Instrument the System

By "instrument the system," I mean that you should liberally and in great volume spread debug code throughout your application. To instrument the code means to embed in your systems the ability to generate copious amounts of trace information at will.


This will provide you the ability to do two things:

- **Debug the code without a debugger.** When you run into a problem with the database, the support technicians tell you to enable trace events that generate trace files containing tons of diagnostic information. This enables the support and development team to debug your problems, without ever touching your system. Imagine for a moment the impact of Oracle not having SQL_TRACE or the event system.
- **Identify where bad performance is in your system.** If you are building N-tier applications with many moving pieces, this is extremely vital. When the support desk calls you and says "it is going slow," how will you find out where your system is slowing down without the ability to have the code itself tell you what is taking a long time?

There are those that will say, "But this will add overhead. I really don't need to do this." Instrumentation is not overhead. Overhead is something you can remove without losing much benefit. Removing (or not having) instrumentation takes away considerable functionality. You wouldn't need to do this if your systems never break, never need diagnostics, and never suffer from performance issues. If that is true, you don't need to instrument your system (and send me your email address, because I have a job offer for you).

Trace from asktom.oracle.com

To see instrumentation in action, go to asktom.oracle.com and click on any article from the home page. You'll see a URL similar to:

 `http://asktom.oracle.com/pls/ask/f?p=...:NO:...`

If you simply type over that URL and replace the word NO with YES, you'll see the same page, but with a lot of state and timing information dumped into it. (This example is not a functional URL; go to the asktom web site to get a real one to test this on.)

Every single application I work on has this feature—this tracing ability—in some form. If someone reports a slow page, I can easily and quickly determine which region of the page is causing the problem. If a certain query that paints part of the page is taking an inordinate amount of time, it will be obvious where that is.

In addition to this instrumentation inside the page, I audit each and every page access as well. That means that for each page accessed in my application—every page, every time—I insert an audit trail record containing whatever information I desire. On the asktom web site, I nominally capture the IP address, browser, time of request, duration of the request, what page in what application was requested, and other bits of data. This allows me to rapidly respond to questions like, “How many people use asktom?” and “How many pages (versus images and such) do you serve up every day?”

“Over the last 24 hours, I have observed that accessing your site and clicking on Ask Question, Read Question, Review Question, etc., all go slowly (taking around one to three minutes instead of couple of seconds, as before). Are others facing the same problem?”

I just went to my statistics page, generated right from my audit trail as part of my system (clicked a button) and saw this:

	Last 24 hours	Last 60 minutes	Last 60 seconds
Page Views	27,348	759	9
Page Views/Sec	0.317	0.211	0.150
IP Addresses	2,552	130	4
Views per IP Address	10.716	5.838	2.250
Users	2,986	147	6
Views per User	9.159	5.163	1.500
Distinct Pages	20	16	3
Avg Elap/Page (secs)	0.25	0.40	0.77
Fastest Page (secs)	0.13	0.1	0.28

I instantly saw that the access problem was not caused by the database. In fact, it probably wasn't a widespread problem at all (that is, this person was having an isolated experience). I

could tell it wasn't the database by the Avg Elap/Page value, which fell right into the norm (the database was generating pages as per normal). I could tell it wasn't a widespread problem because the number of page views fell well within my normal range. (I have another report I look at from time to time that tells me page views per day/week and so on.) Basically, I know that during the week I get between 25,000 and 35,000 page views a day on average, from 2,000 to 3,000 different IP addresses. The last 24 hours on asktom.oracle.com fell well within the norm. If pages had been taking two to three minutes to generate instead of well under a second, people would have stopped clicking on them, and the hit counts would have been way down.

So, within a couple of seconds, I was able to rule out the database, the web server, and my front-end network as the causes for the visitor's slow responses. I also asked people coming to asktom.oracle.com to tell me how it was performing, and without exception, their answer was that its performance was normal, further confirming that this was an isolated incident.

Later, the poser of this question informed me that the problem was resolved. As he was somewhere deep in Australia, and I'm on the East Coast of the United States, I can only speculate that there was some nasty network issue that got corrected.

Instrument for Remote Debugging

Another example of the benefits of instrumentation involves a piece of code I wrote a long time ago, when the Internet was just catching on. Many of you are familiar with `mod_plsql`, an Apache module Oracle provides that allows URLs to run PL/SQL stored procedures; this is the way asktom.oracle.com works, for example. The precursor to `mod_plsql` was called OWA, for the Oracle Web Agent. This was originally a CGI-BIN program that shipped with the Oracle Internet Server (OIS) version 1.0, which became the Oracle Web Server (OWS) version 2.0 and 2.1.

The OWA *cartridge*, as it was called, was fairly simple in its functionality: It ran stored procedures and that was about it. I reimplemented the same concept in 1996, but added support for file uploading and downloading, flexible argument passing, database authentication, an `<ORACLE>` tag (like PSPs, or PL/SQL Server Pages), compression, web-timed statistics, and so on—many features you now see in the `mod_plsql` module. (The success of my piece of software prompted the developers to include these features in the supported code.)

I called my piece of software OWAREPL and put it up on the Internet. It was downloaded thousands of times, and I still hear about people using it today. It is a piece of C code, about 3,500 lines of it, using OCI (the Oracle Call Interface) to interact with the database.

Since I do not write 100% bug-free code, and people used this software in ways I never anticipated (as people are known to do), I needed to be able to remotely debug this piece of code. Fortunately, I had the code fully instrumented and able to dump huge amounts of diagnostic information. Time after time, I was able to remotely diagnose and either fix or suggest a workaround for issues with this piece of software, all via email. All I had to do when someone hit a problem was send the following message:

Please set `debugModules = all` in the `sv<webservername>.app` configuration file. That will generate a trace file after you restart the web server. Run your application, reproduce the issue, and then email me the resulting trace file.

I never needed to physically log in to another machine outside Oracle in order to collect the diagnostic information. Not only that, but since this piece of code ran as a CGI-BIN or as a

dynamically loaded cartridge under OWS/OAS, using a debugger was out of the question. If I didn't have this tracing ability, my only answer to questions about problems with the program would have been, "Gee, I don't know. Sorry."

By now, you realize how beneficial this technique is. The question remains, "How can I instrument my code?" There are many ways, and we'll explore them from the database on out to the typical application tiers (client server and *N*-tier applications).

Use DBMS_APPLICATION_INFO Everywhere

DBMS_APPLICATION_INFO is one of the many database packages supplied with Oracle. It is also one of the most underutilized packages.

Did you know that if you execute a long-running command such as CREATE INDEX or UPDATE on a million rows in Oracle (*long-running* is defined as longer than three to five seconds), a dynamic performance view V\$SESSION_LONGOPS will be populated with information? This view includes the following information:

- When the command started
- How far it has progressed
- Its estimated time to completion

DBMS_APPLICATION_INFO allows you to set values in the view V\$SESSION_LONGOPS. This capability is useful for recording the progress of long-running jobs.

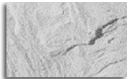
You may have wished that such a facility existed for your own long-running stored procedures and other operations. Well, you might be surprised to learn that, since version 8.0 of the database, DBMS_APPLICATION_INFO can help to answer all of these questions:

- What is the session doing, what form is it running, what code module is executing?
- How far along is that stored procedure?
- How far along is that batch job?
- What bind variable values were being used on that query?

DBMS_APPLICATION_INFO allows you to set up to three columns in your row of the V\$SESSION table: the CLIENT_INFO, ACTION, and MODULE columns. It provides functions not only to set these values, but also to return them. Furthermore, there is a parameter to the built-in USERENV or SYS_CONTEXT function that will allow you to access the CLIENT_INFO column easily in any query. I can select userenv('CLIENT_INFO') from dual, for example, or use where some_column = sys_context('userenv', 'CLIENT_INFO') in my queries. The MODULE column should be the name of your major process, such as the package name. The ACTION column is suitable for storing the procedure name you are executing in a package.

NOTE

The values you set in the V\$ tables are immediately visible. You do not need to commit them to see them, making them very useful for communicating with the outside world.




My general guidelines for using the DBMS_APPLICATION_INFO package are as follows:

- Use the SET_CLIENT_INFO call to store useful “state” information in V\$SESSION. Use it for data you would like the DBAs or yourself to be able to see to gain some insight into what the program is doing. For example, you might consider putting the bind variable values in here before executing some long-running query. Then anyone with access to the V\$ tables cannot only see your SQL, but also view the inputs to that SQL.
- Use the SET_MODULE/GET_MODULE call inside your PL/SQL routines as you enter and exit routines, at least the major routines. In this fashion, you can query V\$SESSION and see what routine you are in currently. SET_MODULE allows you to set two components in V\$SESSION: the MODULE and ACTION columns.
- Use the SET_SESSION_LONGOPS functionality for any routine that will take more than a couple of seconds, in general. This will allow you to monitor its progress, see exactly how far along it is, and estimate its time to completion.


If you are interested in more details on the DBMS_APPLICATION_INFO package, refer to the *Oracle Supplied Packages Guide*.

Use DEBUG.F in PL/SQL

Another facility I use heavily is a custom-developed package I call DEBUG, with a function in it called simply F. It is a technique I have been using in my code for well over 16 years now. It started with a C implementation I modeled after `printf` (hence, the name DEBUG.F; in C the routine was called `debugf`). It is a package designed to allow you to add logging or trace statements to a PL/SQL application. It is simple to use. For example, you might code:

```
 Create procedure p( p_owner in varchar2, p_object_name in varchar2)
As
    L_status number := 0;
Begin
    Debug.f( 'Entering procedure, inputs "%s", "%s"',
            P_owner, p_object_name );
    ... some code ...
    debug.f( 'Normal exit, status = %d', l_status );
end;
```

That would (if the debug package were told to generate debug trace files for this module via a call to DEBUG.INIT) generate a trace file message in the operating system that looked like this:

```
 12062002 213953(P.PROCEDURE 5) Enter procedure inputs "A", "B"
12062002 213955(P.PROCEDURE 56) Normal exit, status = 0
```

This shows three bits of information:

- When procedure P was called (12062002 is the date in MMDDYYYY format, and 213953 is the time in HH24MISS format)

- If you print them, the inputs that were sent—any other DEBUG.F calls would have their output interspersed in this trace file
- That the routine was exited normally with a status of 0 two seconds later

From this sort of output, you can easily resolve debug issues, such as those that arise when someone calls you and says, “Hey, I’m getting an error from an end user running your code. What should we do?” The answer becomes, “In SQL*Plus, call DEBUG.INIT and tell it to debug (trace) procedure P for that user and that user only.” After they reproduce the error, you generally have sufficient information in your trace file to debug that on your own now (for example, you have the inputs, or maybe the error is clear just from the generated trace and you can fix it).

Additionally, you have timestamps on processes. This is a very easy way to pinpoint where a slowdown is in a big, multiprocedure process. When you get the message, “Hey, your stuff is slow,” you can enable tracing for a user and simply look at the timestamps to see what is slow (just like asktom.oracle.com, as described earlier). In a matter of seconds, you can track down where the performance issue is and zero in on it. You can review and download the DEBUG.F implementation from <http://asktom.oracle.com/~tkyte/debugf.html>.

Turn on SQL_TRACE in Your Application

One of the most powerful tuning tools in the database is SQL_TRACE, which allows you to trace all SQL statements and PL/SQL blocks your application executes. It also includes information such as the number of I/O operations performed, how long the query ran (elapsed times), how many times the SQL was executed, and so on. Without a facility like SQL_TRACE, tuning a SQL application would be exceedingly difficult, at best. When you use SQL_TRACE, every piece of code that executes SQL in the database is already on its way to being partially instrumented. (For more information about SQL_TRACE, see the Oracle *Performance Tuning Guide and Reference*.)

So, why do so many systems that are implemented make it virtually impossible to use SQL_TRACE? The answer lies in the fact that most people don’t think about tracing *until* after the system is having a performance issue (you know, after it is in production, without having been tested to scale). It is then that they discover the way they crafted their application makes it virtually impossible to trace using this vital facility.

“We have a web-based application where transactions are managed by MS-DTC (Microsoft Distributed Transaction Coordinator). All of the SQL to do select/insert/update/delete for all tables is in stored procedures, which are in packages. These stored procedures are called by COM+ components. The COM+ components use one username (app_user) to log in to the database and execute these packages. Connection pooling is managed by IIS, and depending on the users hitting the web site, I can see multiple app_user sessions connected to the database. In this scenario, how can I run TKPROF?”

Well, unfortunately for them, running TKPROF isn’t going to happen very easily. The normal database methods of doing this, such as a SCHEMA LOGON trigger that enabled tracing for a session based on a single user, are useless. The use of a single username to log in to the database obviates that approach, and the connection pool further aggravates it. With a typical connection pool, a database session (the level at which we generally trace) is shared across multiple unrelated end user sessions. Without a connection pool the application would own a database connection, the level at which Oracle is built to “trace” at from start to

finish. With a connection pool, that one connection is shared by perhaps every end user session in the system. We end up with a trace file that has not only the trace information we are interested in, but the trace information from any end user session that used the connection (in short, a mess). The resulting trace files from such a thing would be meaningless (your SQL would be intermingled with my SQL, and with their SQL, and with everyone's SQL).

So, what they need to do in their application is

- Add the ability to set a flag as a property in a configuration file somewhere, so their COM components can be told "turn on tracing for GEORGE," or "turn on tracing for module X."
- When their COM component grabs a connection from the pool, it should issue `alter session set sql_trace=true` if appropriate.
- Immediately prior to returning the connection to the pool, if they enabled trace, they would need to set `sql_trace=false`.

Now, their application, which knows when GEORGE is running it or when module X is being executed, can enable and disable trace at will. They can selectively get the trace just for that user or that module. Now, the traces might all end up going into separate trace files on the server (since each time they grab a connection from the pool, they might get another session, and the trace files are session-based), but at least these files will contain only trace information for their user/module. They can run TKPROF on any one of the trace files and have meaningful information.

Use Industry-Standard APIs

Even programming languages are getting into this "instrument-you-code" mode. For example, the web page at <http://java.sun.com/j2se/1.4/docs/guide/util/logging/> describes a very sophisticated, extensible, logging Application Programming Interface (API) for J2EE-based applications. This API follows the same concepts I've been promoting here: Liberally sprinkle log messages throughout your code. This logger package generates log messages in XML, for example. Sun even has a nice Document Type Definition (DTD) for it, and you can code an XSL (Extensible Stylesheet Language) file to format the XML into a nice report. The log message might look something like this:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE log SYSTEM "logger.dtd">
<log>
<record>
  <date>2002-12-06 23:21:05</date>
  <millis>967083665789</millis>
  <sequence>1234</sequence>
  <logger>demo.test.foo</logger>
  <level>INFO</level>
  <class>demo.test.LogTest</class>
  <method>writeLog</method>
  <thread>10</thread>
```

```
<message>Entered routine, inputs = 5 and 'Hello'</message>
</record>
</log>
```

As you can see, this shows information that is similar to what the DEBUG.F routine (described earlier) generates. It outputs all of the information necessary to diagnose major performance issues (timestamps), and if you put enough diagnostic capabilities in there, it generates enough information to debug your applications without a debugger.

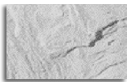
The logger package is new with J2EE 1.4, but there are many other solutions, such as the Log4J logging API or the Jakarta Common Logging Component. Search www.google.com for those keywords, and you'll find tons of information on them.

Build Your Own Routines

If you are not using PL/SQL or Java J2EE, you can build your own DEBUG.F routines. I've done it for every language I've worked with. For example, earlier I described OWAREPL, a piece of software I wrote that many people downloaded and used over the years. I implemented `debugf` in C for that program. It works on the same principles as DEBUG.F in PL/SQL and the logger package in Java. If you are interested, you can download the source code (although it is now obsolete) from <http://asktom.oracle.com/~tkyte/owarepl/doc/> and see it at work in `startup.c` and `owarepl.h`.

Audit Is Not a Four-Letter Word

Many people are of the opinion that auditing is something to be avoided—unnecessary overhead. I find auditing to be extremely useful every day, in every way. I'm frequently asked, "How can we find out who dropped this table?" Well, after the fact, unless you had auditing enabled, you won't be able to do this easily. More important, an audit trail will show you useful information over time that allows you to pinpoint performance issues, usage patterns, and what's popular (and conversely what isn't). It allows you to see how people actually use your application or how they abuse it.



NOTE

In Oracle9i with supplemental logging, you can use Log Miner to see who dropped a table. But an audit trail makes this easy and more efficient, because it can be selective. Redo is either generated for every drop operation or not at all.

For example, on asktom.oracle.com, I noticed a sudden jump in hits on my site, from a normal of around 30,000 page views to more than 150,000. Wow, I thought, it has gotten really popular. Then I thought, or has it? That increase was too sudden and too large. In reviewing the audit trail, I detected a new browser—a web "whacker" if you will. Someone was crawling the asktom.oracle.com site and downloading the entire thing. Well, the problem was that every time that user hit the site with the web whacker, she got a new session ID in the URL, meaning that she downloaded the entire site every time.

The asktom.oracle.com site does not lend itself to incremental downloads. So, what did I do? I just added code to the header for asktom.oracle.com (the code that runs at the top of each page) that looked at the browser type. If it is a web whacker, the code returns a page that effectively

says, “Please don’t try to use download tools,” with no links. Instantly, the page view totals on asktom.oracle.com went right back to their normal levels. Now, every time I see a spike in activity like that, my audit trail tells me what happened. It gives me the new browser type, and I filter that one out as well.

The funny thing about audit trails is that you don’t realize that you need them until after you need them! Recently, I was asked (and I get this question about once a week—just replace *function* with *package*, *procedure*, *sequence*, *table*, *view* ...), “How do I get the following information from system tables? How do I identify the DDL actions performed in the last 24 hours? If a developer overwrites some function, how can I tell who did it?” The only way to know for sure is to have had auditing enabled—either basic auditing, using the built-in features of the database, or custom auditing, using system event triggers (BEFORE CREATE, BEFORE DROP, and so on).

In your application, you should audit activities as well, beyond the normal auditing that can take place in the database. You should audit into database tables. This is my preference because it’s easy to mine the data in the database, but it’s hard to mine data if it is in a file or XML document somewhere. Use a system like the one I have on asktom.oracle.com: every page view is an audit event, an insert into a table. Remember that databases were born to insert into; it is one of the things they do best. Don’t be afraid to add an insert here and there to audit. The benefits far outweigh any perceived performance hit.

Question Authority

Not long ago, a message came to me on asktom.oracle.com that began something like this:

I’ve recently read an article about performance on <some web site>, and I couldn’t believe what I was reading about commits. I usually read your answers on this site, and I’ve read your book as well. You suggest avoiding frequent commits. Below is an extract of the article. Please let me know what you think about the author suggestion regarding commit.

“Issue Frequent COMMIT Statements”

Whenever possible, issue frequent COMMIT statements in all your programs. By issuing frequent COMMIT statements, the performance of the program is enhanced and its resource requirements are minimized as COMMIT frees up the following resources: 1) Information held in the rollback segments to undo the transaction, if necessary 2) All locks acquired during statement processing 3) Space in the redo log buffer cache and 4) Overhead associated with any internal Oracle mechanisms to manage the resources in the previous three items”

Well, the funny thing was this performance article actually had more than 50 points in it—over 50 things to do to enhance performance. The problem was that, almost without exception, every item was either dead wrong (as this item on committing is) or totally misleading.

So, after debunking one item—this myth about commits (a very popular myth; for some reason, it seems many people have heard it and live by it)—the author of this paper got in touch with me. Concerned that this one item was wrong, he asked if I would review the rest and let him know if there were other issues. So I did. I wrote a point-by-point rebuttal for the first 25 items. Then, being really tired from all that typing, I just glanced at the rest and confirmed they were of the same quality. I sent the email off to the author, and he responded, “Thanks for pointing out all the errors. It’s really surprising that the Tuning Book contained so many errors.”

What’s this? Turns out the 50-plus items he was attributing to himself weren’t really his original ideas, but rather items he pulled from books on performance tuning. He never even tried the ideas to prove that they worked! He took verbatim as truth every word from those books.

Beware of Universal “Bests”

Anything related to performance tuning can be proven (via benchmarking, for example). Not only can it be proven, but it should be proven. And even more important, the caveats must be pointed out as well. What works well in a certain set of circumstances may fail miserably in others.

For example, some people swear that you should never use IN (*subquery*); you should always use WHERE EXISTS (*correlated subquery*). There is another camp that believes the reverse. How could two groups come to such different—opposite, in fact—conclusions? Easily—they had different conditions under which they did their tuning. One group found that using IN for their set of queries with their data was really bad and using WHERE EXISTS was really good. The other group with different data and different queries discovered the opposite. Neither group is correct; both are wrong, and both are right. Certain techniques are applicable in certain conditions. There are no rules of thumb in performance tuning that are universally applicable. This is why I avoid answering questions like these:

- What is the best way to analyze tables?
- What is the best kind of table?
- What is the best way to synchronize two databases?

If there were a universal best way, the software would provide only that way. Why bother with inferior methods?

Suspect Ratios and Other Myths

You will hear unproven statements like, “Your cache hit ratio should be above 96% on a well-tuned system. Your goal is to adjust the buffer cache in order to achieve that.” You should wonder why this is so. Did they prove it? Do others agree with it? For example, that buffer cache one is a popular myth. I have an “anti-myth” that contradicts it.

I have a theory that a system with a high cache hit ratio is a system in need of serious tuning! Why? Because you are doing too many logical I/Os. Logical I/Os require latches. Latches are locks. Locks are serialization devices, which inhibit scalability. A high cache hit ratio could be indicative of overusing and abusing indexes on your system. Consider this small example:

```
... Ok, here you go. So, joe (or josephine) SQL coder needs
to run the following query:
select t1.object_name, t2.object_name
  from t1, t2
 where t1.object_id = t2.object_id
       and t1.owner = 'WMSYS'
```

Here, T1 and T2 are huge tables, with more than 1.8 million rows each (copies of the view ALL_OBJECTS so you have an idea about the width of the tables as well). Joe runs the query and uses SQL_TRACE on it to see how it is doing. This is the result:

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	35227	5.63	9.32	23380	59350	0	528384
total	35229	5.63	9.33	23380	59350	0	528384

Rows	Row Source Operation
528384	HASH JOIN
8256	TABLE ACCESS FULL T1
1833856	TABLE ACCESS FULL T2

“Stupid, stupid Cost Based Optimizer (CBO),” Joe says. “I have indexes. Why won’t it use it? We all know that indexes mean fast! Not only that, but look at that cache hit for that query—about 50%. That’s terrible! Totally unacceptable according to this book I read. Okay, let me use the faithful Rule Based Optimizer (RBO) and see what happens.”

```
select /*+ RULE */ t1.object_name, t2.object_name
  from t1, t2
 where t1.object_id = t2.object_id
        and t1.owner = 'WMSYS'
```

So, Joe explains that, with these results:

Execution Plan

```
0      SELECT STATEMENT Optimizer=HINT: RULE
1    0      TABLE ACCESS (BY INDEX ROWID) OF 'T2'
2    1      NESTED LOOPS
3    2      TABLE ACCESS (FULL) OF 'T1'
4    2      INDEX (RANGE SCAN) OF 'T2_IDX' (NON-UNIQUE)
```

“Excellent,” he says, “Look at that, it is finally using my index. I know indexes are good.” Joe’s problem is solved, until he runs the query, that is. TKPROF shows the true story:

call	count	cpu	elapsed	disk	query current	rows
Parse	1	0.00	0.00	0	0	0
Execute	1	0.00	0.00	0	0	0
Fetch	35227	912.07	3440.70	1154555	121367981	0
total	35229	912.07	3440.70	1154555	121367981	0

Be careful of what you ask for! Joe got his index used, but it was deadly. Yes, that is ten seconds versus one hour, and all because of indexes and good cache hit ratios. The DBA might be pleased now:

```
1  SELECT phy.value,
2         cur.value,
3         con.value,
4         1-((phy.value)/((cur.value)+(con.value))) "Cache hit ratio"
```

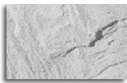


```
5 FROM v$sysstat cur, v$sysstat con, v$sysstat phy
6 WHERE cur.name='db block gets'
7 AND   con.name='consistent gets'
8* AND   phy.name='physical reads'
ops$tkyte@ORA920.US.Oracle.COM> /
```

VALUE	VALUE	VALUE	Cache hit ratio
1277377	58486	121661490	.989505609

A cache hit ratio of 98.9%—boy, did Joe do a good job or not! Obviously not, but many people would say “Well, that is a finely tuned system based on that cache hit. You know that a physical I/O is 10,000 times more expensive than a logical I/O is!” (that is said tongue in cheek of course, a physical IO is not anywhere near 10,000 times as expensive as a cache read).

When you read generalizations like, “cache hit ratios of X% indicate you’ve tuned well,” be suspicious. Remember that there are no universal rules that apply in all cases. In fact, if you read about ratios at all, be skeptical. It is not that ratios are useless; it is just that they are generally not useful all by themselves. They must be used in conjunction with other variables in order to make sense. The tools that show “green” lights when the cache hit ratio is 90% or more, versus red when it falls below, are somewhat misleading. A 99% cache hit ratio could be as indicative of a problem as it could be indicative of a well-run system.



NOTE
I know of exactly one ratio that is meaningful all by itself, and even then, it doesn’t apply to a data warehouse. This is the ratio of soft to hard parses, which should be near 100%. Bind variables that allow for soft parses are something you want to use. See the “Small-Time Benchmarking” section earlier in this chapter for proof.

Don’t Look for Shortcuts

Don’t look for shortcuts. Everyone is in search of the `fast=true init.ora` parameter. Let me bring you in on a secret: It doesn’t exist (although there are some `slow=yes` ones out there!).

Frequently, I am asked about undocumented parameters or Oracle internals. My response is, “Have you read the *Concepts Guide* from cover to cover yet?” Invariably, they have not. They haven’t mastered the basics yet, and they want “internals.”

People believe there must be some hidden nugget—some magic thing—they can just turn on in their database, and it will go faster and run better. That way, they don’t need to bother with learning about the database.

The real problem is that undocumented parameters have side effects that the uninitiated just cannot anticipate. They do not have enough knowledge of the basics to understand what ramifications setting some of these parameters might have. Lack of breadth of knowledge here, coupled with undocumented parameters, can lead to disaster.

Recently, we had a discussion on the Internet Usenet newsgroups about this. The thread on `comp.databases.oracle.server` was entitled, “Why are people so afraid of underscore parameters”

(all Oracle database undocumented parameters start with an underscore character). Someone suggested, “Why not take `_trace_files_public` as an example? That may be technically the most harmless underscore parameter.”

As its name implies, the `_trace_files_public` parameter makes the trace files generated by Oracle accessible to everyone. On a development instance, where the developers need to run a TKPROF report for tuning, this is very handy. On a production instance (where you should *not* be tuning anyway), turning on this parameter is a rather large security issue.

Sensitive information, such as passwords, can be generated in these trace files. There is an undocumented but well-known method to dump the library cache (a portion of the SGA, or shared global area), and this dump goes into a trace file. If a user has ALTER SESSION privileges and knowledge of this command, and `_trace_files_public` is set to true, you have a rather large security issue on your hands and you don’t even realize it (you also have another security issue related to the ALTER SESSION privilege, but that is another story). Who can say what other unintended side effects will rear their ugly heads with the other “amazing” undocumented parameters.

In short, undocumented parameters on a production instance should be set only under the direction of Oracle support. No ifs, ands, or buts about it. This applies to even the most experienced Oracle teams. And, even when you are instructed to use an undocumented parameter, you need to reevaluate its usage when you upgrade your system. Typically, if you need to set an undocumented parameter, there is a reason, sometimes known as a *bug*. Bugs get fixed, and having the old, undocumented parameters set after an upgrade can be disastrous.

For example, for Oracle7.3, there was an undocumented way to enable bitmap indexes using an underscore parameter to set an event (before these indexes went into production in version 7.3.3). Word of this got around, and people started setting this parameter and training others to do it (if you search groups.google.com for “bitmap indexes 7.3 group:comp.databases.oracle.*”, you’ll find out how). Then they would upgrade to 7.3.3 and leave the event set. So, what happened? Well, the event caused the 7.3.3 code to run the old bitmap code, not the new production code (oops). As you can imagine, this caused some serious support issues, especially over time, as issues with the old code arose. In fact, some people left these events in all of the way up through Oracle8i, causing severe performance degradation of InterMedia text searches, a special indexing technique for textual information in the database. That got filed as a bug—a performance bug, if you left that event on.

In my experience, I have not used internal knowledge nor undocumented magic incantations to make database applications perform faster, scale better, and achieve their goals. I used my knowledge of the basics—the knowledge I gained from the standard documentation, plus a lot of experience and common sense. As I stressed earlier in the chapter, to get the most out of your database, begin by reading the *Concepts Guide*. If you read 100% of that guide and retain even just 10% of it, you’ll already know 90% more than most people. It is that basic knowledge that will make you successful, not some magic parameter.

Keep It Simple

I follow the KISS principle (for keep it simple, silly; others have another word for the last S). I prefer the path of least resistance, or the course of action that is easiest to achieve without error.

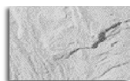
Consider Alternate Approaches

I frequently get questions that start with, “Without doing *X*, how can I do *Y*?” Time and time again, I see people opting for the hardest possible approach to a problem. They have a solution in mind, and they are going to use that solution regardless. They have precluded all other approaches.

For example, someone recently asked me about how to add a table column between existing columns, without dropping and re-creating the table. Her table (T) had the column order Code, Date, Status, and Actiondate, and she wanted to put a new column (New_Column) between the Status and Actiondate columns. I pointed out that the physical order of columns isn’t something you should rely on. In order to achieve what she wants, she can take these simple steps:

- Rename <their table> to <their_table>_TABLE
- Alter the renamed table and add the new column
- Create view <their table> as `select code, date, status, new_column, actiondate from T_TABLE`

Her response was, “I want to add a column in the middle of the table, but I do not want to rebuild it and I don’t want to use a view. Try again.” Well, what could I say to that? She wanted to get from point A to point C and refused to go through point B, even though that is the only logical way to go. It would take all of five seconds to do this with a view; this is what views are all about. No application would be the wiser (unless the addition of this column broke it, of course). It is the smart, clean, and correct solution for the problem.



NOTE

People have told me that views are slow. That is just not true. A view is nothing more than a stored query! If the stored query text is slow, sure the view (or queries against the view) will be slow. A view cannot be any slower than the query itself. When you use the specific query as defined by the view in straight SQL, you’ll see the same performance.

So, someone else comes along with this “great” idea (this was in version 8i, which does not have a rename column function, or else the steps would be shorter). Here T represents their original tablename and NEW_COLUMN is the column they wish to add:

- Alter table T and add NEW_COLUMN
- Alter table T and add ACTIONDATE_TEMP
- Update T and set ACTIONDATE_TEMP = ACTIONDATE
- Alter table T and drop column ACTIONDATE
- Alter table T and add ACTIONDATE
- Update T and set ACTIONDATE = ACTIONDATE_TEMP
- Alter table T and drop ACTIONDATE_TEMP

And voilà, you have “added” `NEW_COLUMN` in the middle. Let’s see—she didn’t want to rebuild the table, and she didn’t want to use a view, but to this she says, “Thanks, that’s my answer!” Never mind the fact that “her answer” rewrites the table over and over, most likely migrating many rows due to the updates and negatively impacting performance. Never mind that on a table of any size it would take many times longer than rebuilding the table. Never mind that no matter the size, it is infinitely complex when compared to using a view.

I am appalled that someone would think of this solution as viable and use it. It is the worst approach to a relatively simple problem. A rebuild would rewrite the table once. A view would be done with the change in five seconds. But she really wants to rewrite the table repeatedly. One false step, and the table would be destroyed as well (it is error prone).

Let the Database Do What It Does Best

Another classic example of keeping it simple is to let the database do what it does best. Do not try to outsmart it.

As an example, I was recently asked my advice about outer joins. The DBA explained that he had a big `CUSTOMERS` table and seven or eight small tables referenced to `CUSTOMERS` columns (`COUNTRY`, `STATE`, `CITY`, `CUSTOMER_GROUP`, and so on). The data in the tables was language-sensitive, with the language depending on the web client language `in` parameter. In some cases, outer joins would be needed. He wanted to know the best way to handle this kind of query, and he had three ideas:

- Several outer joins and filtering with language `in` parameter
- Function-based column: `fn_country(customer.id_country, p_inlang)`
- Only `CUSTOMER` columns in cursor, open it and apply `fn_country` on every fetch from ready and filtered customer data

I replied that his first idea, using outer joins, was the only really good choice, and there is more than one way to code it. Outer joins are not *evil*. When used appropriately, they are no faster or slower than a regular join. You can either code:

```
select t.*, t2.c1, t3.c2
  from t, t2, t3
 where t.key1 = t2.key1(+)
        and t.key2 = t3.key2(+)
```

or

```
select t.*, (select c1 from t2 where t2.key1 = t.key1) c1,
           (select c2 from t3 where t3.key2 = t.key2) c2
  from t;
```

The DBA’s second idea, to use a function-based column, is an attempt to outsmart the database, essentially saying, “I can do an outer join faster than you can.” Well, you cannot. The `fn_country()` approach would cause SQL to call PL/SQL, and that PL/SQL itself will call SQL again to either find a row or not. This is much less efficient than just letting SQL do its job in the first place. The third idea, applying `fn_country` on every fetch, is even worse, because it involves a client round-trip.

But this DBA was convinced that his second or third idea would work the best. He felt that the less burden on SQL, the faster it will go. After much back and forth, he finally called me on it. He said, “You demonstrate most of what you say with examples and numbers. I would appreciate it if you can show an example or two that can prove the above statements.” Touché. If you refer back to the “Question Authority” section, you will see that he nailed it—it was time for me to put up or shut up.

I pulled out my trusty Runstats tool (described in the “Small-time Benchmarking” section of this chapter). I started with two tables to compare outer joining versus trying to outsmart the engine:

```
create table t1 as select * from all_objects;
create table t2 as select * from all_objects where rownum <= 15000;

alter table t1 add constraint t1_pk primary key(object_id);
alter table t2 add constraint t2_pk primary key(object_id);

analyze table t1 compute statistics
for table for all indexes for all indexed columns;

analyze table t2 compute statistics
for table for all indexes for all indexed columns;
```

Then I created a function in the style the DBA suggested:

```
create or replace function get_data( p_object_id in number ) return varchar2
is
    l_object_name t2.object_name%type;
begin
    select object_name into l_object_name
    from t2
    where object_id = p_object_id;
    return l_object_name;
exception
    when no_data_found then
        return NULL;
end;
/
```

Now, I was ready to compare these two equivalent queries:

```
select a.object_id, a.object_name oname1, b.object_name oname2
from t1 a, t2 b
where a.object_id = b.object_id(+);

select object_id, object_name oname1, get_data(object_id) oname2
from t1;
```

I fired up the Runstats test harness, and the results were incredible:

```

ops$tkyte@ORA920> begin
2     runstats_pkg.rs_start;
3     for x in ( select a.object_id,
4                   a.object_name oname1,
5                   b.object_name oname2
6                   from t1 a, t2 b
7                   where a.object_id = b.object_id(+) )
8     loop
9         null;
10    end loop;
11    runstats_pkg.rs_middle;
12    for x in ( select object_id,
13                object_name oname1,
14                get_data(object_id) oname2
15                from t1 )
16    loop
17        null;
18    end loop;
19    runstats_pkg.rs_stop;
20 end;
21 /
84 hsecs
2803 hsecs
run 1 ran in 3% of the time

```

This ran in less than 5% of the runtime by the wall clock, simply by letting the database do what databases were born to do! Not only that, but look at the latching/statistic report generated by Runstats:

Name	Run1	Run2	Diff
STAT...consistent gets - exami	21	78,155	78,134
STAT...session logical reads	15,553	109,775	94,222
STAT...consistent gets	15,524	109,747	94,223
LATCH.cache buffers chains	31,118	141,477	110,359
LATCH.library cache pin	167	126,460	126,293
LATCH.shared pool	383	126,698	126,315
LATCH.library cache	380	189,780	189,400
STAT...session pga memory max	196,608	0	-196,608
STAT...session uga memory max	1,767,880	0	-1,767,880
Run1 latches total versus runs -- difference and pct			
33,264	590,820	557,556	5.63%

PL/SQL procedure successfully completed.

You can see there are 95,000 less logical I/O operations, for half million fewer latches (latches = locks, remember). Again, this was achieved by just letting the database do its thing.

By keeping it simple—not trying to outguess or outsmart the database—we achieved our goal. And this path was easier. It is so much easier to just outer join than it is to write a little function for each and every outer join we anticipate doing.

The point is that if there is an apparently easy way to do something and a convoluted, hard way to do the same thing, and they result in the same goal being met, by all means, be lazy like me and take the easy way. Not only is the easy way, well, easier, it is also usually the correct way as well.

Use Supplied Functionality

Time and time again, I see people reinventing database functionality. Here are some of the top functions people refuse to use:

- **Auditing** Rather than use the built in AUDIT command of fine-grained auditing, some people are bent on doing it themselves via triggers. The most common reason is, “We’ve heard auditing is slow.”
- **Replication** Rather than use master-to-master replication, snapshot-based replication, or streams, some people do it themselves. The most common reason is, “We’ve heard the built-in stuff is complicated.”
- **Message queuing** Rather than use the advanced queuing (AQ) software in the database, some people look for undocumented features, tricky ways to have multiple processes read separate records from a “queue” table and process them efficiently. The most common reason is, “We just don’t want to use AQ.”
- **Maintaining a history of record changes** Rather than use the Workspace Manager, which can do this automatically, some people write tons of code in triggers to attempt to do the same thing. The most common reason is, “We didn’t know that functionality existed.”
- **Sequences** Rather than use a built-in sequence number, some people create their own approach using database tables. Then they want to know how to scale that up (which isn’t possible, because if you use a table yourself, it will be a serial process, and serial means not scalable). The most common reason is, “Not every database has a sequence. We are developing for databases X and Y, and can use only features they both have.”

The reasons for not using database functionality span from FUD (fear, uncertainty, doubt), to ignorance of what the database provides, to the quest for database independence. None of these are acceptable. We’ll take a look at each reason in turn.

We Heard Feature X Is Slow

This excuse has to do with the “fear” part of FUD. It seems to be a common perception that the built-in auditing facility in Oracle is slow. I believe this stems from the fact that if you have auditing turned on, the system runs slower than when auditing is not turned on (seems obvious). So, the mythology begins with:

- We didn’t have auditing on, and things ran okay.
- We turned on auditing, and they ran slower.
- Therefore, auditing is really slow.

And that story is passed down from generation to generation of Oracle development staff. When I hear this, I ask these simple questions:

- Did you benchmark it?
- Do you have the hard and fast numbers?
- What exactly is the overhead?
- What are the numbers?

You probably can guess that no one has tried it; they just heard that it was slow. What you will find out if you benchmark this is that built-in auditing has the following benefits over any other solution:

- It's faster, because it is internal. It is in the kernel, in C code that will be faster than the trigger you use to emulate its functionality.
- It's easier because it is a single command, rather than a design of an entire subsystem for your application, plus the code, plus the schema design, and so on.
- It's more secure because it is part of the database. It is evaluated to be "correct" by independent third parties. It is not a simple trigger someone can disable.
- It's more flexible because it is in the Oracle audit trail, or maybe better yet, it is in the operating system audit trail. There are hundreds of tools that know these schemas/formats. How many tools out there know your tables? You would have a very large do-it-yourself project, not only to capture but to report and make sense of the audit trail.

Let's look at a simple benchmark that you can try yourself. I wanted to audit inserts into a table. Here is one way (after making sure AUDIT_TRAIL was set in the init.ora parameter file):

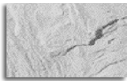
```
create table t1 ( x int );
audit insert on t1 by access;
```

Here is another way:

```
create table t2 ( x int );
create table t2_audit
as
select sysdate dt, a.*
  from v$session a
 where 1=0;
create index t2_audit_idx on t2_audit(sid,serial#);


create trigger t2_audit
after insert on t2
begin
  insert into t2_audit
  select sysdate, a.*
    from v$session a
```

```
        where sid = (select sid
                      from v$mystat
                      where rownum=1);
end;
/
```



NOTE
There are many ways to implement this “do it yourself” auditing. I chose a method that involved the least amount of code and used techniques I frequently see others implementing. Another valid approach would be to populate an application context upon logging into the system—removing the need to query the V\$SESSION/V\$MYSTAT table for every insert statement on T2. This will decrease the CPU used but you will lose some of the information that can change in V\$SESSION during the course of your session.

Now, it should be obvious which one was easier to develop and required less thought on my part, but which method performed better? Well, in order to answer this question, I set up a small procedure:

```
create table t1_times ( xstart timestamp, xstop timestamp );
create or replace procedure p1( n in number )
as
    l_rowid rowid;
begin
    insert into t1_times (xstart) values (systimestamp)
    returning rowid into l_rowid;
    for i in 1 .. n
    loop
        insert into t1 values (i);
        commit;
    end loop;
    update t1_times set xstop = systimestamp where rowid = l_rowid;
    commit;
end;
/
```

And did the same for T2. Then I ran this procedure five times in the background, each inserting 30,000 rows. Before and after doing this, I took a Statspack snapshot. Here are some of the relevant numbers:

	T1 with Native Auditing	T2 with DIY Auditing	Comment
Transactions/Second	380	278	27% decrease in transactions/second
CPU Time	302	443	146% of the CPU time

Even for this simple example, the results are clear. It is easier, faster, and all around more efficient to use the native functionality. Again, do not accept word of mouth. Prove that a method is slower (or faster), or have it proven to you.

We Heard Feature X Is Complicated

Just the other day, someone asked me how the MD5 function from the DBMS_OBFUSCATION_TOOLKIT worked. When I asked why she was interested, she said she wanted to “optimize” a refresh of a slowly changing dimension in a data warehouse table. Her approach was to bring the primary key and a checksum of the rest of the data over a database link, and compute and compare the sent checksum to a checksum of the local data. If the checksums were different, they would go back and pull the entire row over the database link to update the row in the local database. If the row didn’t exist, it would be inserted. After pulling changes and inserting new rows, she had to turn the problem around and send all of the primary keys over to the remote database to see if they still existed. If not, those keys had to be deleted locally.

My solution was to create a snapshot log on the remote table, create a snapshot locally with a refresh interval, and be done with it.

What happened here was she had heard that replication is complicated. Rather than investigate what it would take to accomplish her goal using that database feature, she set about doing it herself. Would her approach work? Well, she didn’t consider some possibilities. What happens if someone modifies the remote table while making multiple passes on it and such (data-integrity issues)? What about the infrastructure you need to design and build to monitor such a solution? (There isn’t a single systems management tool on the market that knows how to watch their process and page a DBA when it breaks). There were other things she hadn’t yet considered, but would need to address in a production environment.

Yes, it’s true that replication in general is complicated. It is not that the implementation in Oracle is complicated, but rather that the replication itself is hard stuff. It is not something you just throw into a system. There are many things to consider. But consider how hard it is to implement this feature yourself! For example, read-only snapshots, a feature available with Oracle since 1993 in version 7.0, completely satisfy the requirements of the above example. This feature is manageable and implemented in hundreds of locations. Doing it yourself, on the other hand, requires tons of design, code, and time.

Everything is complicated until you learn it. Driving a stick-shift car was complicated the first time I did it. Now, that I drive one every day, I don’t even think about it.

We Don’t Want to

I have never been able to figure out why people go into the “I don’t want to” mode with regard to database features. But, invariably, once they have this attitude, changing their minds is virtually impossible.

“Hey Tom, can you take a look at this SQL for me. I can get a much quicker result if I use `not in`, instead of `not exists`, but I would rather not.

```
[huge query cut out]
```

```
TKPROF REPORT:
```

```
call  cnt          cpu          elap          disk    query    rows
```

```

-----
Parse      1          0.02          0.04          0          0          0
Execute    1          0.00          0.00          0          0          0
Ftch 1939  25772.65    25976.95    149793  29294754  29061
-----
tot   1941  25772.67    25976.99    149793  29294754  29061

```

I tried using all the anti-join hints in the subquery without much success. (There are appropriate indexes on all tables and everything has been analyzed). If I were to modify it and say

```
AND a.x not in (select b.y from b WHERE b.a = 'X')
```

then it comes back in like 5 minutes as opposed to the almost 8 hours for the original. What are your thoughts?"

My thoughts? My thoughts are that he knows the answer and knows exactly what to do. But, for whatever reason, he doesn't want to do it. My solution is that he should forget about what he would rather do and do what he knows he needs to do.

A technology that people commonly just "don't want to" use is the AQ software in the database. A common business problem is that applications will place messages into tables. These are records that represent work to be done (for example, process an order). The technical problem we need to solve here is how do we make it so that the following goals are met:

- Each message gets processed at most once.
- Each message gets processed at least once.
- Many messages must be worked on simultaneously by many background processes.

How do we have many users working concurrently on these rows while at the same time making sure each message gets processed at least and at most one time? We cannot just update a record and mark it in process. If we did that and did not commit, all of the other sessions trying to get a message would block on that row trying to update it (so the messages wouldn't be worked on simultaneously). If we update a record and mark it in process and commit it before actually processing it, the message won't be processed. If our process fails, the message is marked as processed (or in process). No one else would ever pick it up. In short, this is a very sticky problem in a relational database—we really want a multiple-consumer queue table. Fortunately, the database provides one for us. AQ does exactly this:

- It has simple routines to create a queue.
- It has simple stored procedures to enqueue a message, allowing a client to put a message in the queue.
- It has a simple stored procedure to dequeue a message, allowing a background process to retrieve a message. In fact, the dequeue is built to be highly concurrent. You could have two, three, or dozens of dequeue processes running concurrently against the same queue.

So, it just wouldn't make sense not to want to use AQ to solve our problem.

The only way I've succeeded in changing the "I don't want to" attitude is to apply peer pressure or pressure from management. A manager asking you, "Why are you writing hundreds of lines of code when you could just do this?" does seem to carry some weight.

We Didn't Know

The worst reason not to use a database feature is because you didn't know about it. I will refer back to the "Read the Documentation" section earlier in this chapter, where I plead with you to read the manuals, or at least the *New Features Guide* and *Concepts Guide* (with each release).

What you don't know is costing you time, money, energy, and brain cells. The database incorporates a lot of functionality. Don't be penny wise but pound foolish here. Taking an afternoon to skim these manuals and discover what is out there will pay for itself in a week.

We Want Database Independence

Another reason that people do things the hard way 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 those as simple as stored procedures or sequences, because that will lock them into a database. But, in reality, 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 (not Oracle!), 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. You'll find that, given the same exact transaction mix, reports come out with different answers in different databases because of fundamental implementation differences. It is a very rare application that can simply be picked up and moved from one database to another. There will always be differences in the way that the SQL is interpreted and processed.

As an example, consider a recent project to build a web-based product using Visual Basic (VB), ActiveX Controls, Internet Information Services (IIS) Server, and Oracle. The developers expressed concern that since the business logic had been written in PL/SQL, the product had become database-dependent. They wanted to know how to correct this. I was taken aback by this question. In looking at the list of chosen technologies, I could not figure out how being database dependent was a bad thing:

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

Every other technology choice they had made locked them into a very specific configuration. In fact, the only technology that offered them any choice as far as open systems went was the database. 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, especially when it was crucial to their success?

We can put a slightly different spin on this argument if we consider it from the perspective of “openness.” You put all of your data into the database. The database is a very open tool. It supports data access via SQL, Enterprise JavaBeans (EJBs), Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), Server Message Block (SMB), and many other protocols and access mechanisms. Sounds great so far—the most open thing in the world.

Then you put all your application logic and security outside the database—perhaps in your beans that access the data, or in the Java Server Pages (JSPs) that access the data, or in your VB code running under Microsoft Transaction Server (MTS). The end result is that you have just closed off your database. 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 remember that the “whiz bang” technology of today, such as EJBs, are yesterday’s concepts and tomorrow’s old, tired technology. What has persevered for over 20 years in the relational world (and probably most of the object implementations as well) is the database itself. The front-ends to the data change almost yearly, and as they do, the applications that have all of the security built inside themselves, not in the database, become obstacles, or road blocks to future progress.

Let’s consider one Oracle-specific database feature called fine-grained access control (FGAC). 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, you can enforce security such that, for example:

- Any query executed outside 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 nonsensitive information would go to a remote client terminal.

Basically, it allows you to locate access control in the database, right next to the data. It no longer matters if the user comes at the data from an EJB, 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, 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 on an implementation here)
- The solution that allows access from anything that can talk to the database, over protocols as diverse as Secure Sockets Layer (SSL), HTTP, and Net8 (and others) or using APIs such as ODBC, JDBC, OCI, and so on

I have yet to see an ad-hoc reporting tool that will query your VB code. However, I know of dozens that can do SQL.

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; squeeze every last bit of functionality out of that product you can. 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.

Summary

In this chapter, we looked at many of the softer issues surrounding an Oracle database implementation. We've looked at the DBA/developer relationship and how that should work. We've looked into why it is important to understand the tool you have at your disposal—the Oracle database. I've stressed how it is more than just a bit bucket, a place to dump data. We've gone over what happens if you don't use a test environment, as well as covered many other issues.

I wanted to get across two very important messages. One is that it isn't true unless it can be proven to be true. And, if it can be proven to be true (or false) it should be. Remember your earlier education, when you were taught the scientific method: Formulate a test to fortify your hypothesis (or equally as likely, destroy it). Use this technique every day when developing your systems. Put simply, question authority.

The other main point to take away from this chapter is to exploit what you have. Exploit your operating system, your language, your middle tier, and your database. Use as much supplied functionality as possible. It is faster to develop, cheaper, and generally leaves you with a better product at the end of the day. Use the extra time you have to benchmark, instrument, and test!

In the next chapter, we'll take a look at the tools you can use not only to develop and tune your applications, but also to question authority and benchmark with. We've mentioned a few of them in passing in this chapter such as SQL_TRACE, TKPROF, and Runstats already. Now we are ready to go into greater detail.