Expert Oracle Database Architecture

9*i* and 10*g* Programming Techniques and Solutions

Thomas Kyte

Apress®

Expert Oracle Database Architecture: 9*i* and 10*g* Programming Techniques and Solutions

ISBN (pbk): 1-59059-530-0

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Tony Davis
Technical Reviewer: Jonathan Lewis, Roderick Manalac, Michael Möller, Gabe Romanescu
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Associate Publisher: Grace Wong
Project Manager: Sofia Marchant
Copy Edit Manager: Nicole LeClerc
Assistant Production Director: Kari Brooks-Copony
Production Editor: Katie Stence
Compositor: Dina Quan
Proofreader: Linda Marousek
Indexer: Broccoli Information Management
Artist: Kinetic Publishing Services, LLC
Interior Designer: Van Winkle Designer Group
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Source Code section.

Contents

Foreword

"THINK." In 1914, Thomas J. Watson, Sr. joined the company that was to become IBM, and he brought with him this simple one-word motto. It was an exhortation to all IBM employees, no matter their role, to take care in decision-making and do their jobs with intelligence. "THINK" soon became an icon, appearing on publications, calendars, and plaques in the offices of many IT and business managers within and outside IBM, and even in *The New Yorker* magazine cartoons. "THINK" was a good idea in 1914, and it is a good idea now.

"Think different." More recently, Apple Computer used this slogan in a long-running advertising campaign to revitalize the company's brand, and even more important, to revolutionize how people think of technology in their daily lives. Instead of saying "think differently," suggesting *how* to think, Apple's slogan used the word "different" as the object of the verb "think," suggesting *what* to think (as in, "think big"). The advertising campaign emphasized creativity and creative people, with the implication that Apple's computers uniquely enable innovative solutions and artistic achievements.

When I joined Oracle Corporation (then Relational Software Incorporated) back in 1981, database systems incorporating the relational model were a new, emerging technology. Developers, programmers, and a growing group of database administrators were learning the discipline of database design using the methodology of normalization. The then unfamiliar nonprocedural SQL language impressed people with its power to manipulate data in ways that previously took painstaking procedural programming. There was a lot to think about then—and there still is. These new technologies challenged people not only to learn new ideas and approaches, but also to think in new ways. Those who did, and those who do, were and are the most successful in creating innovative, effective solutions to business problems using database technology to its best advantage.

Consider the SQL database language that was first introduced commercially by Oracle. SQL permits application designers to manipulate sets of rows with a nonprocedural (or "declarative") language, rather than writing iterative loops in conventional languages that process records one at a time. When I was first introduced to SQL, I found it required me to "think at 45 degrees" to figure out how to use set processing operations like joins and subqueries to achieve the result I wanted. Not only was the idea of set processing new to most people, but so also was the idea of a nonprocedural language, where you specified the result you wanted, not how to derive it. This new technology really did require me to "think differently" and also gave me an opportunity to "think different."

Set processing is far more efficient than one-at-a-time processing, so applications that fully exploit SQL in this way perform much better than those that do not. Yet, it is surprising how often applications deliver suboptimal performance. In fact, in most cases, it is application design, rather than Oracle parameter settings or other configuration choices, that most directly determines overall performance. Thus, application developers must learn not only details about database features and programming interfaces, but also new ways to think about and use these features and interfaces their applications.

Much "conventional wisdom" exists in the Oracle community about how to tune the system for best performance or the best way to use various Oracle features. Such "wisdom" sometimes becomes "folklore" or even "mythology," with developers and database administrators adopting these ideas uncritically or extending these ideas without reasoning about them.

One example is the idea that "if one is good, more—lots more—is better." This idea is popular, but only rarely true. Take Oracle's array interface, for example, which allows the developer to insert or retrieve multiple rows in a single system call. Clearly, reducing the number of network messages between the application and the database is a good thing. But, if you think about it, there is a point of diminishing returns. While fetching 100 rows at once is far better than one at a time, fetching 1,000 rows at once instead of 100 is generally not really any more efficient overall, especially when you consider memory requirements.

Another example of uncritical thinking is to focus on the wrong aspects of system design or configuration, rather than those most likely to improve performance (or, for that matter, reliability, availability, or security). Consider the "conventional wisdom" of tuning the system to maximize the buffer hit ratio. For some applications, it's true that maximizing the chance that required data is in memory will maximize performance. However, for most applications it's better to focus attention on performance bottlenecks (what we call "wait states") than it is to focus on specific system-level metrics. Eliminate those aspects of the application design that are causing delays, and you'll get the best performance.

I've found that breaking down a problem into smaller parts and solving each part separately is a great way to think about application design. In this way, you can often find elegant and creative uses of SQL to address application requirements. Often, it is possible to do things in a single SQL statement that at first seem to require complex procedural programming. When you can leverage the power of SQL to process sets of rows at a time, perhaps in parallel, not only are you more productive as an application developer, but the application runs faster as well!

Sometimes, best practices that were based, even in part, on some degree of truth become no longer applicable as the facts change. Consider the old adage, "Put indexes and data in separate tablespaces for best performance." I've often seen database administrators express strong opinions over the merits of this idea, without taking into account changes in disk speeds and capacities over time, or the specifics of given workloads. In evaluating this particular "rule," you should think about the *fact* that the Oracle database caches frequently and recently used database blocks (often blocks belonging to an index) in memory, and the *fact* that it uses index and data blocks sequentially, not simultaneously, for any given request. The implication is that I/O operations for both index and data really should be spread across all simultaneous users, and across as many disk drives as you have. You might choose to separate index and data blocks for administrative reasons or for personal preference, but not for performance. (Tom Kyte provides valuable insights on this topic on the Ask Tom web site, `http://asktom.oracle.com`, where you can search for articles on "index data tablespace".) The lesson here is to base your decisions on facts, and a complete set of current facts at that.

No matter how fast our computers or how sophisticated the database becomes, and regardless of the power of our programming tools, there simply is no substitute for human intelligence coupled with a "thinking discipline." So, while it's important to learn the intricacies of the technologies we use in our applications, it's even more important to know how to think about using them appropriately.

Tom Kyte is one of the most intelligent people I know, and one of the most knowledgeable about the Oracle database, SQL, performance tuning, and application design. I'm pretty sure

Tom is an aficionado of the "THINK" and "Think different" slogans. Tom quite obviously also believes in that anonymous wise saying, "Give a man a fish and you feed him for a day. Teach a man to fish and you feed him for a lifetime." Tom enjoys sharing his knowledge about Oracle, to the great benefit of our community, but rather than simply dispensing answers to questions, he helps others learn to think and reason.

On his web site (`http://asktom.oracle.com`), in his public speaking engagements, and in this book, Tom implicitly challenges people to "think differently" too, as they design database applications with the Oracle database. He rejects conventional wisdom and speculation, instead insisting on relying on facts proven through examples. Tom takes a very pragmatic and simple approach to problem solving, and by following his advice and methodology, you can be more productive and develop better, faster applications.

Not only will Tom's book teach you about features of Oracle and how to use them, but it also reflects many of these simple thoughts:

- Don't believe in myths—reason for yourself.
- Don't follow "conventional wisdom"—often the things everybody knows are simply wrong!
- Don't trust rumors or opinions—test things for yourself and base decisions on proven examples.
- Break apart a problem into simpler questions, and assemble the answers to each step into an elegant, efficient solution.
- Don't do things in your programs when the database can do them better and faster.
- Understand the differences between the ideal and the real.
- Ask questions about and be skeptical of unjustified "company policies" for technical standards.
- Consider the big picture of what's best overall for the requirements at hand.
- Take the time to THINK.

Tom encourages you to treat Oracle as much more than a black box. Instead of you just putting data into and taking data out of Oracle, Tom will help you understand how Oracle works and how to exploit its power. By learning how to apply Oracle technology creatively and thoughtfully, you will be able to solve most application design problems quickly and elegantly.

As you read and enjoy this book, I know you'll learn a lot of new facts about Oracle database technology and important concepts about application design. As you do, I'm confident that you'll also start to "think differently" about the challenges you face.

IBM's Watson once said, "Thought has been the father of every advance since time began. 'I didn't think' has cost the world millions of dollars." This is a thought with which both Tom and I agree. Armed with the knowledge and techniques you'll learn in this book, I hope you'll be able to save the world (or at least your enterprise) millions of dollars, and enjoy the satisfaction of a job well done.

Ken Jacobs
Vice President of Product Strategy (Server Technologies),
Oracle Corporation

About the Author

I am **TOM KYTE**. I have been working for Oracle since version 7.0.9 (that's 1993 for people who don't mark time by Oracle versions). However, I've been working *with* Oracle since about version 5.1.5c (the $99 single-user version for DOS on 360KB floppy disks). Before coming to work at Oracle, I worked for more than six years as a systems integrator, building large-scale, heterogeneous databases and applications, mostly for military and government customers. These days, I spend a great deal of my time working with the Oracle database and, more specifically, helping people who are using the Oracle database. I work directly with customers, either in specifying and building their systems or, more frequently, in helping them rebuild or tune them ("tuning" frequently being a synonym for rebuilding). In addition, I am the Tom behind the "Ask Tom" column in *Oracle Magazine*, where I answer people's questions about the Oracle database and tools. On a typical day, I receive and answer dozens of questions at `http://asktom.oracle.com`. Every two months, I publish a "best of" in the magazine (all of the questions asked are available on the Web—stored in an Oracle database, of course). Additionally, I give technical seminars covering much of the material you'll find in this book. Basically, I spend a lot of my time helping people be successful with the Oracle database. Oh yes, in my spare time, I build applications and develop software within Oracle Corporation itself.

This book is a reflection of what I do every day. The material within covers topics and questions that I see people struggling with every day. These issues are covered from a perspective of "When *I* use this, *I* do it this way." It is the culmination of many years experience of using the product, in myriad situations.

About the Technical Reviewers

JONATHAN LEWIS has been involved in database work for more than 19 years, specializing in Oracle for the last 16 years, and working as a consultant for the last 12 years. Jonathan is currently a director of the UK Oracle User Group (UKOUG) and is well known for his many presentations at the UKOUG conferences and SIGs. He is also renowned for his tutorials and seminars about the Oracle database engine, which he has held in various countries around the world.

Jonathan authored the acclaimed book *Practical Oracle 8*i (Addison-Wesley, ISBN: 0201715848), and he writes regularly for the UKOUG magazine and occasionally for other publications, including *OTN* and *DBAZine*. He also finds time sometimes to publish Oracle-related material on his web site, `http://www.jlcomp.demon.co.uk`.

RODERICK MANALAC graduated from the University of California, Berkeley in 1989 with a bachelor's degree in electrical engineering and computer science. He's been an employee of Oracle Support Services ever since. Practically all of that time has been spent in assisting external customers and internal employees (around the globe) gain a better understanding of the subtleties involved with running the Oracle database product on UNIX platforms. Other than that, he spends way too much time playing video games, watching TV, eating snacks, and willing the San Francisco Giants to win the World Series.

MICHAEL MÖLLER has been into computers since his tenth birthday, approximately 40 years ago. He's been involved in pretty much everything related to building and running software systems, as a programmer, principal senior system, design engineer, project manager, and quality assurance manager. He worked in the computer business in the United States, England, and Denmark before joining Oracle Denmark ten years ago, where he worked in Support and later in Premium Support. He has often taught in Oracle Education, even taking the "Oracle Internals" seminar on a whistle-stop tour of Europe. He spent the last two years of his time with Oracle working in development in the United States, creating the course materials for advanced courses, including "Internals on NLS" and "RAC." Nowadays Möller is gainfully employed at Miracle A/S in Denmark with consultancy and education.

GABE ROMANESCU has a bachelor's degree in mathematics and works as an independent Oracle consultant. He discovered relational theory and technologies in 1992 and has found comfort ever since in the promise of applied logic in the software industry. He mostly benefits from, and occasionally contributes to, the Ask Tom and OTN forums. He lives in Toronto, Canada, with his wife, Arina, and their two daughters, Alexandra and Sophia.

Acknowledgments

I would like to thank many people for helping me complete this book.

First, I would like to thank Tony Davis for his work making my work read well. If you enjoy the flow of the sections, the number of section breaks, and the clarity, then that is probably in some part due to him. I have worked with Tony writing technical material since the year 2000 and have watched his knowledge of Oracle grow over that time. He now has the ability to not only "edit" the material, but in many cases "tech edit" it as well. Many of the examples in this book are there because of him (pointing out that the casual reader was not going to "get it" without them). This book would not be what it is without him.

Without a technical review team of the caliber I had during the writing of this book, I would be nervous about the content. Jonathan Lewis, Roderick Manalac, Michael Möller, and Gabe Romanescu spent many hours poring over the material and verifying it was technically accurate as well as useful in the real world. I firmly believe a technical book should be judged not only by who wrote it, but also by who reviewed it.

At Oracle, I work with the best and brightest people I have ever known, and they all have contributed in one way or another. I would like to thank Ken Jacobs in particular for his support and enthusiasm.

I would also like to thank everyone I work with for their support during this book-writing ordeal. It took a lot more time and energy than I ever imagined, and I appreciate everyone's flexibility in that regard. In particular, I would like to thank Tim Hoechst and Mike Hichwa, whom I've worked with and known for over 12 years now. Their constant questioning and pushing helped me to discover things that I would never have even thought of investigating on my own.

I would also like to acknowledge the people who use the Oracle software and ask so many good questions. Without them, I would never even have thought of writing this book. Much of what is included here is a direct result of someone asking me "how" or "why" at one time or another.

Lastly, but most important, I would like to acknowledge the unceasing support I've received from my family. You know you must be important to someone when you try to do something that takes a lot of "outside of work hours" and someone lets you know about it. Without the continual support of my wife Lori, son Alan, and daughter Megan, I don't see how I could have finished this book.

Introduction

The inspiration for the material contained in this book comes from my experiences developing Oracle software, and from working with fellow Oracle developers and helping them build reliable and robust applications based on the Oracle database. The book is basically a reflection of what I do every day and of the issues I see people encountering each and every day.

I covered what I felt was most relevant, namely the Oracle database and its architecture. I could have written a similarly titled book explaining how to develop an application using a specific language and architecture—for example, one using JavaServer Pages that speaks to Enterprise JavaBeans, which in turn uses JDBC to communicate with Oracle. However, at the end of the day, you really do need to understand the topics covered in this book in order to build such an application successfully. This book deals with what I believe needs to be universally known to develop successfully with Oracle, whether you are a Visual Basic programmer using ODBC, a Java programmer using EJBs and JDBC, or a Perl programmer using DBI Perl. This book does not promote any specific application architecture; it does not compare three-tier to client/server. Rather, it covers what the database can do and what you must understand about the way it works. Since the database is at the heart of any application architecture, the book should have a broad audience.

In writing this book, I completely revised an updated the architecture sections from *Expert One-on-One Oracle* and added substantial new material. There have been three database releases since Oracle 8.1.7, the release upon which the original book was based: two Oracle9*i* releases and Oracle Database 10*g* Release 1, which is the current production release of Oracle at the time of this writing. As such, there was a lot of new functionality and many new features to cover.

The sheer volume of new material required in updating *Expert One-on-One Oracle* for 9*i* and 10*g* was at the heart of the decision to split it into two books—an already large book was getting unmanageable. The second book will be called *Expert Oracle Programming*.

As the title suggests, *Expert Oracle Database Architecture* concentrates on the database architecture and how the database itself works. I cover the Oracle database architecture in depth—the files, memory structures, and processes that comprise an Oracle database and instance. I then move on to discuss important database topics such as locking, concurrency controls, how transactions work, and redo and undo, and why it is important for you to know about these things. Lastly, I examine the physical structures in the database such as tables, indexes, and datatypes, covering techniques for making optimal use of them.

What This Book Is About

One of the problems with having plenty of development options is that it's sometimes hard to figure out which one might be the best choice for your particular needs. Everyone wants as much flexibility as possible (as many choices as they can possibly have), but they also want things to be very cut and dried—in other words, easy. Oracle presents developers with almost

unlimited choice. No one ever says, "You can't do that in Oracle"; rather, they say, "How many different ways would you like to do that in Oracle?" I hope that this book will help you make the correct choice.

This book is aimed at those people who appreciate the choice but would also like some guidelines and practical implementation details on Oracle features and functions. For example, Oracle has a really neat feature called *parallel execution*. The Oracle documentation tells you how to use this feature and what it does. Oracle documentation does not, however, tell you *when* you should use this feature and, perhaps even more important, *when you should not* use this feature. It doesn't always tell you the implementation details of this feature, and if you're not aware of them, this can come back to haunt you (I'm not referring to bugs, but the way the feature is supposed to work and what it was really designed to do).

In this book I strove to not only describe how things work, but also explain when and why you would consider using a particular feature or implementation. I feel it is important to understand not only the "how" behind things, but also the "when" and "why"—as well as the "when not" and "why not"!

Who Should Read This Book

The target audience for this book is anyone who develops applications with Oracle as the database back end. It is a book for professional Oracle developers who need to know how to get things done in the database. The practical nature of the book means that many sections should also be very interesting to the DBA. Most of the examples in the book use SQL*Plus to demonstrate the key features, so you won't find out how to develop a really cool GUI—but you will find out how the Oracle database works, what its key features can do, and when they should (and should not) be used.

This book is for anyone who wants to get more out of Oracle with less work. It is for anyone who wants to see new ways to use existing features. It is for anyone who wants to see how these features can be applied in the real world (not just examples of how to use the feature, but why the feature is relevant in the first place). Another category of people who would find this book of interest is technical managers in charge of the developers who work on Oracle projects. In some respects, it is just as important that they understand why knowing the database is crucial to success. This book can provide ammunition for managers who would like to get their personnel trained in the correct technologies or ensure that personnel already know what they need to know.

To get the most out of this book, the reader should have

- *Knowledge of SQL*. You don't have to be the best SQL coder ever, but a good working knowledge will help.
- *An understanding of PL/SQL*. This isn't a prerequisite, but it will help you to "absorb" the examples. This book will not, for example, teach you how to program a `FOR` loop or declare a record type—the Oracle documentation and numerous books cover this well. However, that's not to say that you won't learn a lot about PL/SQL by reading this book. You will. You'll become very intimate with many features of PL/SQL and you'll see new ways to do things, and you'll become aware of packages/features that perhaps you didn't know existed.

- *Exposure to some third-generation language (3GL), such as C or Java*. I believe that anyone who can read and write code in a 3GL language will be able to successfully read and understand the examples in this book.
- *Familiarity with the* Oracle Concepts *manual*.

A few words on that last point: due to the Oracle documentation set's vast size, many people find it to be somewhat intimidating. If you're just starting out or haven't read any of it as yet, I can tell you that the *Oracle Concepts* manual is exactly the right place to start. It's about 700 pages long and touches on many of the major Oracle concepts that you need to know about. It may not give you each and every technical detail (that's what the other 10,000 to 20,000 pages of documentation are for), but it will educate you on all the important concepts. This manual touches the following topics (to name a few):

- The structures in the database, and how data is organized and stored
- Distributed processing
- Oracle's memory architecture
- Oracle's process architecture
- Schema objects you will be using (tables, indexes, clusters, and so on)
- Built-in datatypes and user-defined datatypes
- SQL stored procedures
- How transactions work
- The optimizer
- Data integrity
- Concurrency control

I will come back to these topics myself time and time again. These are the fundamentals—without knowledge of them, you will create Oracle applications that are prone to failure. I encourage you to read through the manual and get an understanding of some of these topics.

How This Book Is Structured

To help you use this book, most chapters are organized into four general sections (described in the list that follows). These aren't rigid divisions, but they will help you navigate quickly to the area you need more information on. This book has 15 chapters, and each is like a "mini-book"—a virtually stand-alone component. Occasionally, I refer to examples or features in other chapters, but you could pretty much pick a chapter out of the book and read it on its own. For example, you don't have to read Chapter 10 on database tables to understand or make use of Chapter 14 on parallelism.

The format and style of many of the chapters is virtually identical:

- An introduction to the feature or capability.
- Why you might want to use the feature or capability (or not). I outline when you would consider using this feature and when you would not want to use it.
- How to use this feature. The information here isn't just a copy of the material in the SQL reference; rather, it's presented in step-by-step manner: here is what you need, here is what you have to do, and these are the switches you need to go through to get started. Topics covered in this section will include
  - How to implement the feature
  - Examples, examples, examples
  - How to debug this feature
  - Caveats of using this feature
  - How to handle errors (proactively)
- A summary to bring it all together.

There will be lots of examples, and lots of code, all of which is available for download from the Source Code area of `http://www.apress.com`. The following sections present a detailed breakdown of the content of each chapter.

Chapter 1: Developing Successful Oracle Applications

This chapter sets out my essential approach to database programming. All databases are *not* created equal, and in order to develop database-driven applications successfully and on time, you need to understand exactly *what* your particular database can do and *how* it does it. If you do not know what your database can do, you run the risk of continually reinventing the wheel—developing functionality that the database already provides. If you do not know how your database works, you are likely to develop applications that perform poorly and do not behave in a predictable manner.

The chapter takes an empirical look at some applications where a lack of basic understanding of the database has led to project failure. With this example-driven approach, the chapter discusses the basic features and functions of the database that you, the developer, need to understand. The bottom line is that you cannot afford to treat the database as a black box that will simply churn out the answers and take care of scalability and performance by itself.

Chapter 2: Architecture Overview

This chapter covers the basics of Oracle architecture. We start with some clear definitions of two terms that are very misunderstood by many in the Oracle world, namely "instance" and "database." We also take a quick look at the System Global Area (SGA) and the processes behind the Oracle instance, and examine how the simple act of "connecting to Oracle" takes place.

Chapter 3: Files

This chapter covers in depth the eight types of files that make up an Oracle database and instance. From the simple parameter file to the data and redo log files, we explore what they are, why they are there, and how we use them.

Chapter 4: Memory Structures

This chapter covers how Oracle uses memory, both in the individual processes (Process Global Area, or PGA, memory) and shared memory (SGA). We explore the differences between manual and automatic PGA and, in Oracle 10*g*, SGA memory management, and see when each is appropriate. After reading this chapter, you will have an understanding of exactly how Oracle uses and manages memory.

Chapter 5: Oracle Processes

This chapter offers an overview of the types of Oracle processes (server processes versus background processes). It also goes into much more depth on the differences in connecting to the database via a shared server or dedicated server process. We'll also take a look process by process at most of the background processes (such as `LGWR`, `DBWR`, `PMON`, and `SMON`) we'll see when starting an Oracle instance and discuss the functions of each.

Chapter 6: Locking and Latching

Different databases have different ways of doing things (what works well in SQL Server may not work as well in Oracle), and understanding how Oracle implements locking and concurrency control is absolutely vital to the success of your application. This chapter discusses Oracle's basic approach to these issues, the types of locks that can be applied (DML, DDL, and latches) and the problems that can arise if locking is not implemented carefully (deadlocking, blocking, and escalation).

Chapter 7: Concurrency and Multi-versioning

In this chapter, we'll explore my favorite Oracle feature, multi-versioning, and how it affects concurrency controls and the very design of an application. Here we will see that all databases are *not* created equal and that their very implementation can have an impact on the design of our applications. We'll start by reviewing the various transaction isolation levels as defined by the ANSI SQL standard and see how they map to the Oracle implementation (as well as how the other databases map to this standard). Then we'll take a look at what implications multi-versioning, the feature that allows Oracle to provide non-blocking reads in the database, might have for us.

Chapter 8: Transactions

Transactions are a fundamental feature of all databases—they are part of what distinguishes a database from a file system. And yet, they are often misunderstood and many developers do not even know that they are accidentally not using them. This chapter examines how transactions should be used in Oracle and also exposes some "bad habits" that may have been picked up when developing with other databases. In particular, we look at the implications of

atomicity and how it affects statements in Oracle. We also discuss transaction control statements (`COMMIT`, `SAVEPOINT`, and `ROLLBACK`), integrity constraints, distributed transactions (the two-phase commit, or 2PC), and finally autonomous transactions.

Chapter 9: Redo and Undo

It can be said that developers do not need to understand the detail of redo and undo as much as DBAs, but developers do need to know the role they play in the database. After first defining redo, we examine what exactly a `COMMIT` does. We discuss how to find out how much redo is being generated and how to significantly reduce the amount of redo generated by certain operations using the `NOLOGGING` clause. We also investigate redo generation in relation to issues such as block cleanout and log contention.

In the undo section of the chapter, we examine the role of undo data and the operations that generate the most/least undo. Finally, we investigate the infamous `ORA-01555: snapshot too old` error, its possible causes, and how to avoid it.

Chapter 10: Database Tables

Oracle now supports numerous table types. This chapter looks at each different type—heap organized (i.e., the default, "normal" table), index organized, index clustered, hash clustered, nested, temporary, and object—and discusses when, how, and why you should use them. Most of time, the heap organized table is sufficient, but this chapter will help you recognize when one of the other types might be more appropriate.

Chapter 11: Indexes

Indexes are a crucial aspect of your application design. Correct implementation requires an in-depth knowledge of the data, how it is distributed, and how it will be used. Too often, indexes are treated as an afterthought in application development, and performance suffers as a consequence.

This chapter examines in detail the different types of indexes, including B*Tree, bitmap, function-based, and application domain indexes, and discuss where they should and should not be used. I'll also answer some common queries in the "Frequently Asked Questions and Myths About Indexes" section, such as "Do indexes work on views?" and "Why isn't my index getting used?"

Chapter 12: Datatypes

There are a lot of datatypes to choose from. This chapter explores each of the 22 built-in datatypes, explaining how they are implemented, and how and when to use each one. First up is a brief overview of National Language Support (NLS), a basic knowledge of which is necessary to fully understand the simple string types in Oracle. We then move on to the ubiquitous `NUMBER` type and look at the new Oracle 10*g* options for storage of numbers in the database. The `LONG` and `LONG RAW` types are covered, mostly from a historical perspective. The main objective here is to show how to deal with legacy `LONG` columns in applications and migrate them to the `LOB` type. Next, we delve into the various datatypes for storing dates and time, investigating how to manipulate the various datatypes to get what we need from them. The ins and outs of time zone support are also covered.

Next up are the LOB datatypes. We'll cover how they are stored and what each of the many settings such as `IN ROW`, `CHUNK`, `RETENTION`, `CACHE`, and so on mean to us. When dealing with LOBs, it is important to understand how they are implemented and how they are stored by default—especially when it comes to tuning their retrieval and storage. We close the chapter by looking at the `ROWID` and `UROWID` types. These are special types, proprietary to Oracle, that represent the address of a row. We'll cover when to use them as a column datatype in a table (which is almost never!).

Chapter 13: Partitioning

Partitioning is designed to facilitate the management of very large tables and indexes, by implementing a "divide and conquer" logic—basically breaking up a table or index into many smaller and more manageable pieces. It is an area where the DBA and developer must work together to maximize application availability and performance. This chapter covers both table and index partitioning. We look at partitioning using local indexes (common in data warehouses) and global indexes (common in OLTP systems).

Chapter 14: Parallelism

This chapter introduces the concept of and uses for parallel execution in Oracle. We'll start by looking at when parallel processing is useful and should be considered, as well as when it should not be considered. After gaining that understanding, we move into the mechanics of parallel query, the feature most people associate with parallel execution. Next we cover parallel DML (PDML), which allows us to perform modifications using parallel execution. We'll see how PDML is physically implemented and why that implementation leads to a series of restrictions regarding PDML.

We then move into parallel DDL. This, in my opinion, is where parallel execution really shines. Typically DBAs have small maintenance windows in which to perform large operations. Parallel DDL gives DBAs the ability to fully exploit the machine resources they have available, permitting them to finish large, complex operations in a fraction of the time it would take to do them serially.

The chapter closes on procedural parallelism, the means by which we can execute application code in parallel. We cover two techniques here. The first is parallel pipelined functions, or the ability of Oracle to execute stored functions in parallel dynamically. The second is do-it-yourself (DIY) parallelism, whereby we design the application to run concurrently.

Chapter 15: Data Loading and Unloading

This first half of this chapter focuses on SQL*Loader (SQLLDR) and covers the various ways in which we can use this tool to load and modify data in the database. Issues discussed include loading delimited data, updating existing rows and inserting new ones, unloading data, and calling SQLLDR from a stored procedure. Again, SQLLDR is a well-established and crucial tool, but it is the source of many questions with regard to its practical use. The second half of the chapter focuses on external tables, an alternative and highly efficient means by which to bulk load and unload data.

Source Code and Updates

As you work through the examples in this book, you may decide that you prefer to type in all the code by hand. Many readers choose to do this because it is a good way to get familiar with the coding techniques that are being used.

Whether you want to type the code in or not, all the source code for this book is available in the Source Code area of the Apress web site (`http://www.apress.com`). If you like to type in the code, you can use the source code files to check the results you should be getting—they should be your first stop if you think you might have typed in an error. If you don't like typing, then downloading the source code from the Apress web site is a must! Either way, the code files will help you with updates and debugging.

Errata

Apress makes every effort to make sure that there are no errors in the text or the code. However, to err is human, and as such we recognize the need to keep you informed of any mistakes as they're discovered and corrected. Errata sheets are available for all our books at `http://www.apress.com`. If you find an error that hasn't already been reported, please let us know.

The Apress web site acts as a focus for other information and support, including the code from all Apress books, sample chapters, previews of forthcoming titles, and articles on related topics.

Setting Up Your Environment

In this section, I cover how to set up an environment capable of executing the examples in this book, specifically with regard to the following topics:

- How to set up the SCOTT/TIGER demonstration schema properly
- The environment you need to have up and running
- How to configure AUTOTRACE, a SQL*Plus facility
- How to install Statspack
- How to install and run runstats and other custom utilities used throughout the book
- The coding conventions used in this book

All of the non-Oracle-supplied scripts are available for download from the Source Code area of the Apress web site (http://www.apress.com).

Setting Up the SCOTT/TIGER Schema

The SCOTT/TIGER schema may already exist in your database. It is generally included during a typical installation, but it is not a mandatory component of the database. You may install the SCOTT example schema into any database account—there is nothing magic about using the SCOTT account. You could install the EMP/DEPT tables directly into your own database account if you wish.

Many of the examples in this book draw on the tables in the SCOTT schema. If you would like to be able to work along with them, you will need these tables as well. If you are working on a shared database, it is advisable to install your own copy of these tables in some account other than SCOTT to avoid side effects caused by other users using the same data.

To create the SCOTT demonstration tables, simply

1. cd [ORACLE_HOME]/sqlplus/demo.
2. Run demobld.sql when connected as any user.

Note In Oracle 10*g* and later, you must install the demonstration subdirectories from the companion CD. I have reproduced the necessary components of demobld.sql later as well.

---

demobld.sql will create and populate five tables for you. When it is complete, it exits SQL*Plus automatically, so don't be surprised when SQL*Plus disappears after running the script—it is supposed to do that.

The standard demo tables do not have any referential integrity defined on them. Some of my examples rely on them having referential integrity. After you run demobld.sql, it is recommended that you also execute the following:

```
alter table emp add constraint emp_pk primary key(empno);
alter table dept add constraint dept_pk primary key(deptno);
alter table emp add constraint emp_fk_dept
                              foreign key(deptno) references dept;
alter table emp add constraint emp_fk_emp foreign key(mgr) references emp;
```

This finishes off the installation of the demonstration schema. If you would like to drop this schema at any time to clean up, you can simply execute [ORACLE_HOME]/sqlplus/demo/demodrop.sql. This will drop the five tables and exit SQL*Plus.

In the event you do not have access to demobld.sql, the following is sufficient to run the examples in this book:

```
CREATE TABLE EMP
(EMPNO NUMBER(4) NOT NULL,
 ENAME VARCHAR2(10),
 JOB VARCHAR2(9),
 MGR NUMBER(4),
 HIREDATE DATE,
 SAL NUMBER(7, 2),
 COMM NUMBER(7, 2),
 DEPTNO NUMBER(2)
);

INSERT INTO EMP VALUES (7369, 'SMITH',  'CLERK',     7902,
TO_DATE('17-DEC-1980', 'DD-MON-YYYY'),  800, NULL, 20);
INSERT INTO EMP VALUES (7499, 'ALLEN',  'SALESMAN',  7698,
TO_DATE('20-FEB-1981', 'DD-MON-YYYY'), 1600,  300, 30);
INSERT INTO EMP VALUES (7521, 'WARD',   'SALESMAN',  7698,
TO_DATE('22-FEB-1981', 'DD-MON-YYYY'), 1250,  500, 30);
INSERT INTO EMP VALUES (7566, 'JONES',  'MANAGER',   7839,
TO_DATE('2-APR-1981', 'DD-MON-YYYY'),  2975, NULL, 20);
INSERT INTO EMP VALUES (7654, 'MARTIN', 'SALESMAN',  7698,
TO_DATE('28-SEP-1981', 'DD-MON-YYYY'), 1250, 1400, 30);
INSERT INTO EMP VALUES (7698, 'BLAKE',  'MANAGER',   7839,
TO_DATE('1-MAY-1981', 'DD-MON-YYYY'),  2850, NULL, 30);
INSERT INTO EMP VALUES (7782, 'CLARK',  'MANAGER',   7839,
TO_DATE('9-JUN-1981', 'DD-MON-YYYY'),  2450, NULL, 10);
INSERT INTO EMP VALUES (7788, 'SCOTT',  'ANALYST',   7566,
TO_DATE('09-DEC-1982', 'DD-MON-YYYY'), 3000, NULL, 20);
INSERT INTO EMP VALUES (7839, 'KING',   'PRESIDENT', NULL,
TO_DATE('17-NOV-1981', 'DD-MON-YYYY'), 5000, NULL, 10);
INSERT INTO EMP VALUES (7844, 'TURNER', 'SALESMAN',  7698,
```

```
TO_DATE('8-SEP-1981', 'DD-MON-YYYY'),  1500,    0, 30);
INSERT INTO EMP VALUES (7876, 'ADAMS',  'CLERK',     7788,
TO_DATE('12-JAN-1983', 'DD-MON-YYYY'), 1100, NULL, 20);
INSERT INTO EMP VALUES (7900, 'JAMES',  'CLERK',     7698,
TO_DATE('3-DEC-1981', 'DD-MON-YYYY'),   950, NULL, 30);
INSERT INTO EMP VALUES (7902, 'FORD',   'ANALYST',   7566,
TO_DATE('3-DEC-1981', 'DD-MON-YYYY'),  3000, NULL, 20);
INSERT INTO EMP VALUES (7934, 'MILLER', 'CLERK',     7782,
TO_DATE('23-JAN-1982', 'DD-MON-YYYY'), 1300, NULL, 10);

CREATE TABLE DEPT
(DEPTNO NUMBER(2),
 DNAME VARCHAR2(14),
 LOC VARCHAR2(13)
);

INSERT INTO DEPT VALUES (10, 'ACCOUNTING', 'NEW YORK');
INSERT INTO DEPT VALUES (20, 'RESEARCH',   'DALLAS');
INSERT INTO DEPT VALUES (30, 'SALES',      'CHICAGO');
INSERT INTO DEPT VALUES (40, 'OPERATIONS', 'BOSTON');
```

The Environment

Most of the examples in this book are designed to run 100 percent in the SQL*Plus environment. Other than SQL*Plus, there is nothing else to set up and configure. I can make a suggestion, however, on using SQL*Plus. Almost all the examples in this book use `DBMS_OUTPUT` in some fashion. For `DBMS_OUTPUT` to work, the following SQL*Plus command must be issued:

```
SQL> set serveroutput on
```

If you are like me, typing in this command each and every time will quickly get tiresome. Fortunately, SQL*Plus allows us to set up a `login.sql` file—a script that is executed each and every time we start SQL*Plus. Further, it allows us to set an environment variable, `SQLPATH`, so that it can find this `login.sql` script, no matter what directory it is in.

The `login.sql` I use for all examples in this book is as follows:

```
define _editor=vi
set serveroutput on size 1000000
set trimspool on
set long 5000
set linesize 100
set pagesize 9999
column plan_plus_exp format a80
column global_name new_value gname
set termout off
define gname=idle
column global_name new_value gname
```

```
select lower(user) || '@' || substr( global_name, 1,
   decode( dot, 0, length(global_name), dot-1) ) global_name
  from (select global_name, instr(global_name,'.') dot from global_name );
set sqlprompt '&gname> '
set termout on
```

An annotated version of this is as follows:

- `DEFINE _EDITOR=VI`: This sets up the default editor SQL*Plus will use. You may set the default editor to be your favorite text editor (not a word processor) such as Notepad or emacs.
- `SET SERVEROUTPUT ON SIZE 1000000`: This enables `DBMS_OUTPUT` to be on by default (hence you don't have to type it in each and every time). It also sets the default buffer size as large as possible.
- `SET TRIMSPOOL ON`: When spooling text, lines will be blank-trimmed and not fixed width. If this is set to `OFF` (the default), spooled lines will be as wide as your `LINESIZE` setting.
- `SET LONG 5000`: This sets the default number of bytes displayed when selecting `LONG` and `CLOB` columns.
- `SET LINESIZE 100`: This sets the width of the lines displayed by SQL*Plus to be 100 characters.
- `SET PAGESIZE 9999`: This sets the `PAGESIZE`, which controls how frequently SQL*Plus prints out headings, to a large number (you get one set of headings per page).
- `COLUMN PLAN_PLUS_EXP FORMAT A80`: This sets the default width of the explain plan output you receive with AUTOTRACE. `A80` is generally wide enough to hold the full plan.

The next bit in `login.sql` sets up the SQL*Plus prompt:

```
define gname=idle
column global_name new_value gname
select lower(user) || '@' || substr( global_name,1,
   decode( dot, 0, length(global_name), dot-1) ) global_name
  from (select global_name, instr(global_name,'.') dot from global_name );
set sqlprompt '&gname> '
```

The directive `COLUMN GLOBAL_NAME NEW_VALUE GNAME` tells SQL*Plus to take the last value it retrieves for any column named `GLOBAL_NAME` and place it into the substitution variable `GNAME`. I then select the `GLOBAL_NAME` out of the database and concatenate this with the username I am logged in with. That makes my prompt look like this:

```
ops$tkyte@ora10g>
```

so I know who I am as well as where I am.

Setting Up AUTOTRACE in SQL*Plus

AUTOTRACE is a facility within SQL*Plus that shows you the explain plan of the queries you've executed and the resources they used. This book makes extensive use of the AUTOTRACE facility.

There is more than one way to get AUTOTRACE configured. This is what I like to do to get AUTOTRACE working:

1. `cd [ORACLE_HOME]/rdbms/admin`.
2. log into SQL*Plus as `SYSTEM`.
3. Run `@utlxplan`.
4. Run `CREATE PUBLIC SYNONYM PLAN_TABLE FOR PLAN_TABLE;`.
5. Run `GRANT ALL ON PLAN_TABLE TO PUBLIC;`.

You can replace the `GRANT TO PUBLIC` with some user if you want. By making the `PLAN_TABLE` public, you let anyone trace using SQL*Plus (not a bad thing, in my opinion). This prevents each and every user from having to install his or her own plan table. The alternative is for you to run `@utlxplan` in every schema from which you want to use AUTOTRACE.

The next step is creating and granting the `PLUSTRACE` role:

1. `cd [ORACLE_HOME]/sqlplus/admin`.
2. Log in to SQL*Plus as `SYS` or as `SYSDBA`.
3. Run `@plustrce`.
4. Run `GRANT PLUSTRACE TO PUBLIC;`.

Again, you can replace `PUBLIC` in the `GRANT` command with some user if you want.

About AUTOTRACE

You can automatically get a report on the execution path used by the SQL optimizer and the statement execution statistics. The report is generated after successful SQL DML (i.e., `SELECT`, `DELETE`, `UPDATE`, `MERGE`, and `INSERT`) statements. It is useful for monitoring and tuning the performance of these statements.

Controlling the Report

You can control the report by setting the AUTOTRACE system variable:

- `SET AUTOTRACE OFF`: No AUTOTRACE report is generated. This is the default.
- `SET AUTOTRACE ON EXPLAIN`: The AUTOTRACE report shows only the optimizer execution path.
- `SET AUTOTRACE ON STATISTICS`: The AUTOTRACE report shows only the SQL statement execution statistics.

- SET AUTOTRACE ON: The AUTOTRACE report includes both the optimizer execution path and the SQL statement execution statistics.
- SET AUTOTRACE TRACEONLY: This is like SET AUTOTRACE ON, but it suppresses the printing of the user's query output, if any.

Setting Up Statspack

Statspack is designed to be installed when connected as SYSDBA (CONNECT / AS SYSDBA). To install it, you must be able to perform that operation. In many installations, this will be a task that you must ask the DBA or administrators to perform.

Once you have the ability to connect, installing Statspack is trivial. You simply run @spcreate.sql. You can find that script in [ORACLE_HOME]\rdbms\admin, and you should execute it when connected as SYSDBA via SQL*Plus. It looks something like this:

```
[tkyte@desktop admin]$ sqlplus / as sysdba
SQL*Plus: Release 10.1.0.4.0 - Production on Sat Jul 23 16:26:17 2005
Copyright (c) 1982, 2005, Oracle.  All rights reserved.
Connected to:
Oracle Database 10g Enterprise Edition Release 10.1.0.4.0 - Production
With the Partitioning, OLAP and Data Mining options

sys@ORA10G> @spcreate
... Installing Required Packages
... <output omitted for brevity> ...
```

You'll need to know three pieces of information before running the spcreate.sql script:

- The password you would like to use for the PERFSTAT schema that will be created
- The default tablespace you would like to use for PERFSTAT
- The temporary tablespace you would like to use for PERFSTAT

The script will prompt you for this information as it executes. In the event you make a typo or inadvertently cancel the installation, you should use spdrop.sql to remove the user and installed views prior to attempting another installation of Statspack. The Statspack installation will create a file called spcpkg.lis. You should review this file for any errors that might have occurred. The Statspack packages should install cleanly, however, as long as you supplied valid tablespace names (and didn't already have a PERFSTAT user).

Custom Scripts

In this section, I describe the requirements (if any) needed by various scripts used throughout this book. As well, we investigate the code behind the scripts.

Runstats

Runstats is a tool I developed to compare two different methods of doing the same thing and show which one is superior. You supply the two different methods and runstats does the rest. Runstats simply measures three key things:

- *Wall clock or elapsed time*: This is useful to know, but it isn't the most important piece of information.
- *System statistics*: This shows, side by side, how many times each approach did something (e.g., a parse call) and the difference between the two.
- *Latching*: This is the key output of this report.

As you'll see in this book, latches are a type of lightweight lock. Locks are serialization devices. Serialization devices inhibit concurrency. Applications that inhibit concurrency are less scalable, can support fewer users, and require more resources. Our goal is always to build applications that have the potential to scale—ones that can service 1 user as well as 1,000 or 10,000. The less latching we incur in our approaches, the better off we will be. I might choose an approach that takes longer to run on the wall clock but that uses 10 percent of the latches. I know that the approach that uses fewer latches will scale substantially better than the approach that uses more latches.

Runstats is best used in isolation—that is, on a single-user database. We will be measuring statistics and latching (locking) activity that result from our approaches. We do not want 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.

Note I believe all developers should have a test bed database they control to try ideas on, without needing to ask a DBA to do something all of the time. Developers definitely should have a database on their desktop, given that the licensing for the personal developer version is simply "Use it to develop and test with, do not deploy, and you can just have it." This way, there is nothing to lose! Also, I've taken some informal polls at conferences and seminars and discovered that virtually every DBA out there started as a developer. The experience and training developers could get by having their own database—being able to see how it really works—pays large dividends in the long run.

To use runstats, you need to set up access to several V$ views, create a table to hold the statistics, and create the runstats package. You will need access to three V$ tables (those magic dynamic performance tables): `V$STATNAME`, `V$MYSTAT`, and `V$LATCH`. Here is a view I use:

```
create or replace view stats
as select 'STAT...' || a.name name, b.value
      from v$statname a, v$mystat b
     where a.statistic# = b.statistic#
    union all
    select 'LATCH.' || name,  gets
      from v$latch;
```

Either you can have SELECT on V$STATNAME, V$MYSTAT, and V$LATCH granted directly to you (that way you can create the view yourself) or you can have someone that does have SELECT on those objects create the view for you and grant SELECT privileges to you.

Once you have that set up, all you need is a small table to collect the statistics:

```
create global temporary table run_stats
( runid varchar2(15),
  name varchar2(80),
  value int )
on commit preserve rows;
```

Last, you need to create the package that is runstats. It contains three simple API calls:

- RS_START (runstats start) to be called at the beginning of a runstats test
- RS_MIDDLE to be called in the middle, as you might have guessed
- RS_STOP to finish off and print the report

The specification is as follows:

```
ops$tkyte@ORA920> create or replace package runstats_pkg
  2  as
  3      procedure rs_start;
  4      procedure rs_middle;
  5      procedure rs_stop( p_difference_threshold in number default 0 );
  6  end;
  7  /
Package created.
```

The parameter P_DIFFERENCE_THRESHOLD is used to control the amount of data printed at the end. Runstats collects statistics and latching information for each run, and then prints a report of how much of a resource each test (each approach) used and the difference between them. You can use this input parameter to see only the statistics and latches that had a difference greater than this number. By default this is zero, and you see all of the outputs.

Next, we'll look at the package body procedure by procedure. The package begins with some global variables. These will be used to record the elapsed times for our runs:

```
ops$tkyte@ORA920> create or replace package body runstats_pkg
  2  as
  3
  4  g_start number;
  5  g_run1     number;
  6  g_run2     number;
  7
```

Next is the RS_START routine. This will simply clear out our statistics-holding table and then populate it with the "before" statistics and latches. It will then capture the current timer value, a clock of sorts that we can use to compute elapsed times in hundredths of seconds:

```
 8  procedure rs_start
 9  is
10  begin
11      delete from run_stats;
12
13      insert into run_stats
14      select 'before', stats.* from stats;
15
16      g_start := dbms_utility.get_time;
17  end;
18
```

Next is the RS_MIDDLE routine. This procedure simply records the elapsed time for the first run of our test in G_RUN1. Then it inserts the current set of statistics and latches. If we were to subtract these values from the ones we saved previously in RS_START, we could discover how many latches the first method used, how many cursors (a statistic) it used, and so on.

Last, it records the start time for our next run:

```
19  procedure rs_middle
20  is
21  begin
22      g_run1 := (dbms_utility.get_time-g_start);
23
24      insert into run_stats
25      select 'after 1', stats.* from stats;
26      g_start := dbms_utility.get_time;
27
28  end;
29
30  procedure rs_stop(p_difference_threshold in number default 0)
31  is
32  begin
33      g_run2 := (dbms_utility.get_time-g_start);
34
35      dbms_output.put_line
36          ( 'Run1 ran in ' || g_run1 || ' hsecs' );
37      dbms_output.put_line
38          ( 'Run2 ran in ' || g_run2 || ' hsecs' );
39      dbms_output.put_line
40      ( 'run 1 ran in ' || round(g_run1/g_run2*100,2) ||
41        '% of the time' );
42          dbms_output.put_line( chr(9) );
43
44      insert into run_stats
45      select 'after 2', stats.* from stats;
46
47      dbms_output.put_line
48      ( rpad( 'Name', 30 ) || lpad( 'Run1', 10 ) ||
```

```
49        lpad( 'Run2', 10 ) || lpad( 'Diff', 10 ) );
50
51      for x in
52      ( select rpad( a.name, 30 ) ||
53          to_char( b.value-a.value, '9,999,999' ) ||
54          to_char( c.value-b.value, '9,999,999' ) ||
55          to_char( ( (c.value-b.value)-(b.value-a.value)), '9,999,999' ) data
56          from run_stats a, run_stats b, run_stats c
57         where a.name = b.name
58           and b.name = c.name
59           and a.runid = 'before'
60           and b.runid = 'after 1'
61           and c.runid = 'after 2'
62           and (c.value-a.value) > 0
63           and abs( (c.value-b.value) - (b.value-a.value) )
64                 > p_difference_threshold
65        order by abs( (c.value-b.value)-(b.value-a.value))
66      ) loop
67          dbms_output.put_line( x.data );
68      end loop;
69
70          dbms_output.put_line( chr(9) );
71      dbms_output.put_line
72      ( 'Run1 latches total versus runs -- difference and pct' );
73      dbms_output.put_line
74      ( lpad( 'Run1', 10 ) || lpad( 'Run2', 10 ) ||
75        lpad( 'Diff', 10 ) || lpad( 'Pct', 8 ) );
76
77      for x in
78      ( select to_char( run1, '9,999,999' ) ||
79               to_char( run2, '9,999,999' ) ||
80               to_char( diff, '9,999,999' ) ||
81               to_char( round( run1/run2*100,2 ), '999.99' ) || '%' data
82          from ( select sum(b.value-a.value) run1, sum(c.value-b.value) run2,
83                        sum( (c.value-b.value)-(b.value-a.value)) diff
84                   from run_stats a, run_stats b, run_stats c
85                  where a.name = b.name
86                    and b.name = c.name
87                    and a.runid = 'before'
88                    and b.runid = 'after 1'
89                    and c.runid = 'after 2'
90                    and a.name like 'LATCH%'
91                  )
92      ) loop
93          dbms_output.put_line( x.data );
94      end loop;
95  end;
```

```
 96
 97  end;
 98  /
Package body created.
```

And now we are ready to use runstats. By way of example, we'll demonstrate how to use runstats to see which is more efficient, a single bulk `INSERT` or row-by-row processing. We'll start by setting up two tables into which to insert 1,000,000 rows:

```
ops$tkyte@ORA10GR1> create table t1
  2  as
  3  select * from big_table.big_table
  4  where 1=0;
Table created.

ops$tkyte@ORA10GR1> create table t2
  2  as
  3  select * from big_table.big_table
  4  where 1=0;
Table created.
```

Next, we perform the first method of inserting the records: using a single SQL statement. We start by calling `RUNSTATS_PKG.RS_START`:

```
ops$tkyte@ORA10GR1> exec runstats_pkg.rs_start;
PL/SQL procedure successfully completed.

ops$tkyte@ORA10GR1> insert into t1 select * from big_table.big_table;
1000000 rows created.

ops$tkyte@ORA10GR1> commit;
Commit complete.
```

Now we are ready to perform the second method, which is row-by-row insertion of data:

```
ops$tkyte@ORA10GR1> exec runstats_pkg.rs_middle;

PL/SQL procedure successfully completed.

ops$tkyte@ORA10GR1> begin
  2          for x in ( select * from big_table.big_table )
  3          loop
  4                  insert into t2 values X;
  5          end loop;
  6          commit;
  7  end;
  8  /

PL/SQL procedure successfully completed.
```

and finally, we generate the report:

```
ops$tkyte@ORA10GR1> exec runstats_pkg.rs_stop(1000000)
Run1 ran in 5810 hsecs
Run2 ran in 14712 hsecs
run 1 ran in 39.49% of the time

Name                                  Run1        Run2        Diff
STAT...recursive calls                8,089   1,015,451   1,007,362
STAT...db block changes             109,355   2,085,099   1,975,744
LATCH.library cache                   9,914   2,006,563   1,996,649
LATCH.library cache pin               5,609   2,003,762   1,998,153
LATCH.cache buffers chains          575,819   5,565,489   4,989,670
STAT...undo change vector size    3,884,940  67,978,932  64,093,992
STAT...redo size                118,854,004 378,741,168 259,887,164

Run1 latches total versus runs -- difference and pct
Run1        Run2        Diff       Pct
825,530  11,018,773  10,193,243      7.49%

PL/SQL procedure successfully completed.
```

Mystat

`mystat.sql` and its companion, `mystat2.sql`, are used to show the increase in some Oracle "statistic" before and after some operation. `mystat.sql` simply captures the begin value of some statistic:

```
set echo off
set verify off
column value new_val V
define S="&1"

set autotrace off
select a.name, b.value
from v$statname a, v$mystat b
where a.statistic# = b.statistic#
and lower(a.name) like '%' || lower('&S')||'%'
/
set echo on
```

and `mystat2.sql` reports the difference for us:

```
set echo off
set verify off
select a.name, b.value V, to_char(b.value-&V,'999,999,999,999') diff
from v$statname a, v$mystat b
```

```
where a.statistic# = b.statistic#
and lower(a.name) like '%' || lower('&S')||'%'
/
set echo on
```

For example, to see how much redo is generated by some UPDATE, we can do the following:

```
big_table@ORA10G> @mystat "redo size"
big_table@ORA10G> set echo off

NAME                                 VALUE
------------------------------ ----------
redo size                             496

big_table@ORA10G> update big_table set owner = lower(owner)
  2  where rownum <= 1000;

1000 rows updated.

big_table@ORA10G> @mystat2
big_table@ORA10G> set echo off

NAME                                     V DIFF
------------------------------ ---------- ----------------
redo size                             89592           89,096
```

That shows our UPDATE of 1,000 rows generated 89,096 bytes of redo.

SHOW_SPACE

The SHOW_SPACE routine prints detailed space utilization information for database segments. Here is the interface to it:

```
ops$tkyte@ORA10G> desc show_space
PROCEDURE show_space
 Argument Name                  Type                    In/Out Default?
 ------------------------------ ----------------------- ------ --------
 P_SEGNAME                      VARCHAR2                IN
 P_OWNER                        VARCHAR2                IN     DEFAULT
 P_TYPE                         VARCHAR2                IN     DEFAULT
 P_PARTITION                    VARCHAR2                IN     DEFAULT
```

The arguments are as follows:

- P_SEGNAME: Name of the segment (e.g., the table or index name).
- P_OWNER: Defaults to the current user, but you can use this routine to look at some other schema.

- P_TYPE: Defaults to TABLE and represents the type of object you are looking at. For example, SELECT DISTINCT SEGMENT_TYPE FROM DBA_SEGMENTS lists valid segment types.
- P_PARTITION: Name of the partition when you show the space for a partitioned object. SHOW_SPACE shows space for only one partition at a time.

The output of this routine looks as follows, when the segment resides in an Automatic Segment Space Management (ASSM) tablespace:

```
big_table@ORA10G> exec show_space('BIG_TABLE');
Unformatted Blocks .....                    0
FS1 Blocks (0-25)  .....                    0
FS2 Blocks (25-50) .....                    0
FS3 Blocks (50-75) .....                    0
FS4 Blocks (75-100).....                    0
Full Blocks        .....                14,469
Total Blocks............................ 15,360
Total Bytes............................ 125,829,120
Total MBytes............................    120
Unused Blocks...........................    728
Unused Bytes............................  5,963,776
Last Used Ext FileId....................      4
Last Used Ext BlockId...................   43,145
Last Used Block.........................    296

PL/SQL procedure successfully completed.
```

The items reported are as follows:

- Unformatted Blocks: The number of blocks that are allocated to the table and are below the high-water mark (HWM), but have not been used. Add unformatted and unused blocks together to get a total count of blocks allocated to the table but never used to hold data in an ASSM object.
- FS1 Blocks–FS4 Blocks: Formatted blocks with data. The ranges of numbers after their name represent the "emptiness" of each block. For example, (0–25) is the count of blocks that are between 0 and 25 percent empty.
- Full Blocks: The number of blocks so full that they are no longer candidates for future inserts.
- Total Blocks, Total Bytes, Total MBytes: The total amount of space allocated to the segment measured in database blocks, bytes, and megabytes.
- Unused Blocks, Unused Bytes: These represent a portion of the amount of space never used. These blocks are allocated to the segment but are currently above the HWM of the segment.
- Last Used Ext FileId: The file ID of the file that contains the last extent that contains data.

- `Last Used Ext BlockId`: The block ID of the beginning of the last extent; the block ID within the last used file.
- `Last Used Block`: The offset of the last block used in the last extent.

When you use `SHOW_SPACE` to look at objects in user space managed tablespaces, the output resembles this:

```
big_table@ORA10G> exec show_space( 'BIG_TABLE' )
Free Blocks.............................              1
Total Blocks............................        147,456
Total Bytes.............................  1,207,959,552
Total MBytes............................          1,152
Unused Blocks...........................          1,616
Unused Bytes............................     13,238,272
Last Used Ext FileId....................              7
Last Used Ext BlockId...................        139,273
Last Used Block.........................          6,576

PL/SQL procedure successfully completed.
```

The only difference is the `Free Blocks` item at the beginning of the report. This is a count of the blocks in the first freelist group of the segment. My script reports only on this freelist group. You would need to modify the script to accommodate multiple freelist groups.

The commented code follows. This utility is a simple layer on top of the `DBMS_SPACE` API in the database.

```
create or replace procedure show_space
( p_segname in varchar2,
  p_owner   in varchar2 default user,
  p_type    in varchar2 default 'TABLE',
  p_partition in varchar2 default NULL )
-- this procedure uses authid current user so it can query DBA_*
-- views using privileges from a ROLE, and so it can be installed
-- once per database, instead of once per user who wanted to use it
authid current_user
as
    l_free_blks                 number;
    l_total_blocks              number;
    l_total_bytes               number;
    l_unused_blocks             number;
    l_unused_bytes              number;
    l_LastUsedExtFileId         number;
    l_LastUsedExtBlockId        number;
    l_LAST_USED_BLOCK           number;
    l_segment_space_mgmt        varchar2(255);
    l_unformatted_blocks number;
    l_unformatted_bytes number;
    l_fs1_blocks number; l_fs1_bytes number;
```

```
    l_fs2_blocks number; l_fs2_bytes number;
    l_fs3_blocks number; l_fs3_bytes number;
    l_fs4_blocks number; l_fs4_bytes number;
    l_full_blocks number; l_full_bytes number;

    -- inline procedure to print out numbers nicely formatted
    -- with a simple label
    procedure p( p_label in varchar2, p_num in number )
    is
    begin
        dbms_output.put_line( rpad(p_label,40,'.') ||
                              to_char(p_num,'999,999,999,999') );
    end;
begin
   -- this query is executed dynamically in order to allow this procedure
   -- to be created by a user who has access to DBA_SEGMENTS/TABLESPACES
   -- via a role as is customary.
   -- NOTE: at runtime, the invoker MUST have access to these two
   -- views!
   -- this query determines if the object is an ASSM object or not
   begin
      execute immediate
          'select ts.segment_space_management
             from dba_segments seg, dba_tablespaces ts
            where seg.segment_name      = :p_segname
              and (:p_partition is null or
                  seg.partition_name = :p_partition)
              and seg.owner = :p_owner
              and seg.tablespace_name = ts.tablespace_name'
             into l_segment_space_mgmt
            using p_segname, p_partition, p_partition, p_owner;
   exception
       when too_many_rows then
          dbms_output.put_line
          ( 'This must be a partitioned table, use p_partition => ');
          return;
   end;

   -- if the object is in an ASSM tablespace, we must use this API
   -- call to get space information, otherwise we use the FREE_BLOCKS
   -- API for the user-managed segments
   if l_segment_space_mgmt = 'AUTO'
   then
     dbms_space.space_usage
     ( p_owner, p_segname, p_type, l_unformatted_blocks,
       l_unformatted_bytes, l_fs1_blocks, l_fs1_bytes,
```

```
      l_fs2_blocks, l_fs2_bytes, l_fs3_blocks, l_fs3_bytes,
      l_fs4_blocks, l_fs4_bytes, l_full_blocks, l_full_bytes, p_partition);

    p( 'Unformatted Blocks ', l_unformatted_blocks );
    p( 'FS1 Blocks (0-25)  ', l_fs1_blocks );
    p( 'FS2 Blocks (25-50) ', l_fs2_blocks );
    p( 'FS3 Blocks (50-75) ', l_fs3_blocks );
    p( 'FS4 Blocks (75-100)', l_fs4_blocks );
    p( 'Full Blocks        ', l_full_blocks );
  else
    dbms_space.free_blocks(
      segment_owner     => p_owner,
      segment_name      => p_segname,
      segment_type      => p_type,
      freelist_group_id => 0,
      free_blks         => l_free_blks);

    p( 'Free Blocks', l_free_blks );
  end if;

  -- and then the unused space API call to get the rest of the
  -- information
  dbms_space.unused_space
  ( segment_owner     => p_owner,
    segment_name      => p_segname,
    segment_type      => p_type,
    partition_name    => p_partition,
    total_blocks      => l_total_blocks,
    total_bytes       => l_total_bytes,
    unused_blocks     => l_unused_blocks,
    unused_bytes      => l_unused_bytes,
    LAST_USED_EXTENT_FILE_ID => l_LastUsedExtFileId,
    LAST_USED_EXTENT_BLOCK_ID => l_LastUsedExtBlockId,
    LAST_USED_BLOCK => l_LAST_USED_BLOCK );

    p( 'Total Blocks', l_total_blocks );
    p( 'Total Bytes', l_total_bytes );
    p( 'Total MBytes', trunc(l_total_bytes/1024/1024) );
    p( 'Unused Blocks', l_unused_blocks );
    p( 'Unused Bytes', l_unused_bytes );
    p( 'Last Used Ext FileId', l_LastUsedExtFileId );
    p( 'Last Used Ext BlockId', l_LastUsedExtBlockId );
    p( 'Last Used Block', l_LAST_USED_BLOCK );
end;
/
```

BIG_TABLE

For examples throughout this book, I use a table called BIG_TABLE. Depending on which system I use, this table has between 1 record and 4 million records, and varies in size from 200MB to 800MB. In all cases, the table structure is the same.

To create BIG_TABLE, I wrote a script that does the following:

- Creates an empty table based on ALL_OBJECTS. This dictionary view is used to populate BIG_TABLE.
- Makes this table NOLOGGING. This is optional. I did it for performance. Using NOLOGGING mode for a test table is safe; you won't use it in a production system, so features like Oracle Data Guard won't be enabled.
- Populates the table by seeding it with the contents of ALL_OBJECTS and then iteratively inserting into itself, approximately doubling its size on each iteration.
- Creates a primary key constraint on the table.
- Gathers statistics.
- Displays the number of rows in the table.

To build the BIG_TABLE table, you can run the following script at the SQL*Plus prompt and pass in the number of rows you want in the table. The script will stop when it hits that number of rows.

```
create table big_table
as
select rownum id, a.*
  from all_objects a
 where 1=0
/
alter table big_table nologging;

declare
    l_cnt number;
    l_rows number := &1;
begin
    insert /*+ append */
    into big_table
    select rownum, a.*
      from all_objects a;

    l_cnt := sql%rowcount;

    commit;

    while (l_cnt < l_rows)
    loop
        insert /*+ APPEND */ into big_table
        select rownum+l_cnt,
```

```
                OWNER, OBJECT_NAME, SUBOBJECT_NAME,
                OBJECT_ID, DATA_OBJECT_ID,
                OBJECT_TYPE, CREATED, LAST_DDL_TIME,
                TIMESTAMP, STATUS, TEMPORARY,
                GENERATED, SECONDARY
           from big_table
          where rownum <= l_rows-l_cnt;
         l_cnt := l_cnt + sql%rowcount;
         commit;
    end loop;
end;
/

alter table big_table add constraint
big_table_pk primary key(id)
/

begin
   dbms_stats.gather_table_stats
   ( ownname    => user,
     tabname    => 'BIG_TABLE',
     method_opt => 'for all indexed columns',
     cascade    => TRUE );
end;
/
select count(*) from big_table;
```

I gathered baseline statistics on the table and the index associated with the primary key. Additionally, I gathered histograms on the indexed column (something I typically do). Histograms may be gathered on other columns as well, but for this table, it just isn't necessary.

Coding Conventions

The one coding convention I use in this book that I would like to point out is how I name variables in PL/SQL code. For example, consider a package body like this:

```
create or replace package body my_pkg
as
   g_variable varchar2(25);

   procedure p( p_variable in varchar2 )
   is
      l_variable varchar2(25);
   begin
      null;
   end;
end;
/
```

Here I have three variables: a global package variable, G_VARIABLE; a formal parameter to the procedure, P_VARIABLE; and finally a local variable, L_VARIABLE. I name my variables after the scope they are contained in. All globals begin with G_, parameters with P_, and local variables with L_. The main reason for this is to distinguish PL/SQL variables from columns in a database table. For example, a procedure such as the following:

```
create procedure p( ENAME in varchar2 )
as
begin
   for x in ( select * from emp where ename = ENAME ) loop
      Dbms_output.put_line( x.empno );
   end loop;
end;
```

will always print out every row in the EMP table, where ENAME is not null. SQL sees ename = ENAME, and compares the ENAME column to itself (of course). We could use ename = P.ENAME—that is, qualify the reference to the PL/SQL variable with the procedure name—but this is too easy to forget and leads to errors.

I just always name my variables after the scope. That way, I can easily distinguish parameters from local variables and globals, in addition to removing any ambiguity with respect to column names and variable names.