

Mastering Oracle PL/SQL: Practical Solutions

CONNOR MCDONALD, WITH CHAIM KATZ,
CHRISTOPHER BECK, JOEL R. KALLMAN, AND DAVID C. KNOX

Apress™

Mastering Oracle PL/SQL: Practical Solutions

Copyright © 2004 by Connor McDonald, with Chaim Katz, Christopher Beck, Joel R. Kallman, and David C. Knox

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN (pbk): 1-59059-217-4

Printed and bound in the United States of America 12345678910

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Technical Reviewers: Jakob Hammer-Jakobsen, Torben Holm, Thomas Kyte, Connor McDonald

Technical Editor: Tony Davis

Editorial Board: Steve Anglin, Dan Appleman, Gary Cornell, James Cox, Tony Davis, John Franklin, Chris Mills, Steven Rycroft, Dominic Shakeshaft, Julian Skinner, Martin Streicher, Jim Sumser, Karen Watterson, Gavin Wray, John Zukowski

Assistant Publisher: Grace Wong

Project Manager: Tracy Brown Collins

Copy Editors: Nancy Depper, Nicole LeClerc

Production Manager: Kari Brooks

Production Editor: Janet Vail

Proofreader: Patrick Vincent

Compositor: Gina M. Rexrode, Point n' Click Publishing, LLC

Indexer: Valerie Perry

Artist: Christine Calderwood, Kinetic Publishing Services, LLC

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, email orders@springer-ny.com, or visit <http://www.springer-ny.com>. Outside the United States: fax +49 6221 345229, email orders@springer.de, or visit <http://www.springer.de>.

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, email info@apress.com, or visit <http://www.apress.com>.

The information in this book is distributed on an "as is" basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at <http://www.apress.com> in the Downloads section. You will need to answer questions pertaining to this book in order to successfully download the code.

Contents at a Glance

Foreword to the OakTable Press Series	<i>ix</i>
About the Authors	<i>xi</i>
About the Technical Reviewers	<i>xiii</i>
Acknowledgments	<i>xiv</i>
Introduction	<i>xv</i>
Setting Up	<i>xix</i>
Chapter 1 Efficient PL/SQL	<i>1</i>
Chapter 2 Package It All Up	<i>59</i>
Chapter 3 The Vexed Subject of Cursors	<i>117</i>
Chapter 4 Effective Data Handling	<i>145</i>
Chapter 5 PL/SQL Optimization Techniques	<i>229</i>
Chapter 6 Triggers	<i>307</i>
Chapter 7 DBA Packages	<i>367</i>
Chapter 8 Security Packages	<i>415</i>
Chapter 9 Web Packages	<i>463</i>
Chapter 10 PL/SQL Debugging	<i>505</i>
Appendix A Building DEBUG	<i>559</i>
Index	<i>587</i>

Contents

Foreword to the OakTable Press Series	<i>ix</i>
About the Authors	<i>xi</i>
About the Technical Reviewers	<i>xiii</i>
Acknowledgments	<i>xiv</i>
Introduction	<i>xv</i>
Setting Up	<i>xix</i>
Chapter 1 Efficient PL/SQL	<i>1</i>
Why Use PL/SQL?	<i>1</i>
What is Efficient PL/SQL?	<i>4</i>
Achieving Efficiency	<i>13</i>
Conclusion	<i>56</i>
Chapter 2 Package It All Up	<i>59</i>
Basic Benefits of Packages	<i>59</i>
Standalone Procedures and the Dependency Crisis	<i>65</i>
Breaking the Dependency Chain	<i>75</i>
Enabling Recursion	<i>90</i>
Why Have People Avoided Packages?	<i>91</i>
When Not to Use Packages	<i>96</i>
Delivered Packages	<i>99</i>
Summary	<i>115</i>

Chapter 3 The Vexed Subject of Cursors	117
Implicit vs. Explicit Cursors	117
Cursor Management Across Architectures	132
Summary	143
Chapter 4 Effective Data Handling	145
Taking Control of Your Datatypes	145
From Fields to Rows—Using %ROWTYPE	155
From Records to Objects	162
The Motivation for Collections in PL/SQL	176
Bulk Up with Collections	180
Passing Variables Between PL/SQL Programs	197
Transaction Processing in PL/SQL	212
Autonomous Transactions	218
Summary	228
Chapter 5 PL/SQL Optimization Techniques	229
Minimizing Parsing and Memory Consumption	229
Data Types: Tips and Techniques	254
Calling PL/SQL	267
SQL Within PL/SQL	287
Summary	306
Chapter 6 Triggers	307
Trigger Concepts	307
DML Triggers	319
Instead-of Triggers	324
Mutating Tables	326
Data Auditing	336
Table Versioning	339
Oracle Streams	342
Job Queue (Temporal Event Triggers)	349
DDL Triggers	355
Database Event Triggers	359
Logon Triggers	359
Don't Re-Invent the Wheel	363
Summary	366

Chapter 7 DBA Packages	367
Alert File Package	368
Notification Package	394
Proactive Monitoring Package	399
Historical Data Package	406
Summary	413
Chapter 8 Security Packages	415
Design Considerations	415
Triggers for Security	440
Protecting the Source	453
Summary	462
Chapter 9 Web Packages	463
PL/SQL Web Toolkit Basics	463
The htp and htf Packages	468
Using Environment Variables	470
Cookies	476
Managing Files	478
Managing Tables Through the Web	484
HTTP from the Database	495
Summary	503
Chapter 10 PL/SQL Debugging	505
Defensive Coding	505
Tools	512
A Custom DEBUG Utility	540
Summary	558
Appendix A Building DEBUG	559
Database Design and Setup	559
Package Layout	562
Implementation	564
Troubleshooting DEBUG	586
Index	587

Foreword to the OakTable Press Series

Put simply, the OakTable network is an informal organization consisting of a group of Oracle experts who are dedicated to finding ever better ways of administering and developing Oracle-based systems. We have joined forces with Apress to bring you the OakTable Press series of Oracle-related titles.

The members of the network have a few things in common. We take a scientific approach to working with the Oracle database. We don't believe anything unless we've seen it thoroughly tested and proved. We enjoy "moving boundaries," innovating, finding new and better ways to do things. We like good whiskey. These, in essence, are the ideals that we want to bring to the OakTable Press series (well, apart from the last one, possibly). Every book in the series will be written by and/or technically reviewed by at least two members of the OakTable network. It is our goal to help each OakTable Press author produce a book that is rigorous, accurate, innovative, and fun. Ultimately, we hope that each book is as useful a tool as it can possibly be in helping make your life easier.

Who Are the OakTable Network?

It all started sometime in 1998 when a group of Oracle experts, including Anjo Kolk, Cary Millsap, James Morle, and a few others, started meeting once or twice a year, on various pretexts. Each would bring a bottle of Scotch or Bourbon and in return earn the right to sleep on the floor somewhere in my house.

We spent most of our time sitting around my dining table, with computers, cabling, paper, and other stuff all over the place, discussing Oracle, relaying anecdotes, and experimenting with new and better ways of working with the database. By the spring of 2002, the whole thing had grown. One evening, I realized that I had 16 world-renowned Oracle scientists sitting around my dining table. We were sleeping three or four to a room and even had to borrow the neighbor's shower in the mornings. Anjo Kolk suggested we call ourselves the "OakTable network" (after my dining table), and about 2 minutes later, <http://www.OakTable.net> was registered.

Today, a total of 42 people have been admitted to the OakTable network, with perhaps half of them working for Oracle (there's an up-to-date list on the website). A committee, consisting of James Morle, Cary Millsap, Anjo Kolk, Steve Adams, Jonathan Lewis, and myself, reviews suggestions for new members.

You can meet us at various conferences and user group events, and discuss technical issues with us or challenge the OakTable network with a technical question. If we can't answer your question within 24 hours, you get a T-shirt that says, "I challenged the OakTable—and I won," with the three last words printed in very, very small type! We still meet twice a year in Denmark: in January for the Miracle Master Class (2001: Cary Millsap, 2002: Jonathan Lewis, 2003: Steve Adams, and 2004: Tom Kyte), when one of the members will present for three days, and in September/October for the Miracle Database Forum, which is a three-day conference for database people.

Many projects and ideas have come out of the OakTable network, with some of them resulting in courses (such as the Hotsos Clinic), others resulting in new software products, and one that resulted in the OakTable Press series. We hope you'll enjoy the books coming out of it in the coming years.

Best,

Mogens Nørgaard

CEO of Miracle A/S (<http://www.miracleas.dk/>) and cofounder of the OakTable network

About the Authors



Connor McDonald, lead author, has been working with Oracle since the early 1990s. His involvement with the Oracle database started with versions 6.0.36 and 7.0.12. Over the last 11 years he has worked with systems in Australia, the United Kingdom, Southeast Asia, Western Europe, and the United States. Connor is a member of the OakTable network and is a well-known personality both on the Oracle speaker circuit and in online Oracle forums. He hosts a hints and tips website (<http://www.oracledba.co.uk>) to share his passion for Oracle and as part of an endeavor to improve the way in which Oracle software is used within the industry.

Chaim Katz is an Oracle Certified Professional who has worked with Oracle products since Oracle version 4. He specializes in database administration and PL/SQL development and, over the years, he has written numerous articles for various Oracle technical journals. He has taught Logo to children and database systems at the college level. He lives in Montreal, Quebec, where aside from his 9-to-5 job in information systems, he likes to study the Talmud, play clarinet, and discuss eternal problems. He and his wife, Ruthie, are currently enjoying the challenges of raising a large family.

Christopher Beck, who holds a bachelor's degree in computer science from Rutgers University, has worked in the industry for 13 years. Starting off as a junior Ada software developer for a government contractor, he has spent the last 9 years with Oracle Corporation and is now a principal technologist. He specializes in core database technologies and Web application development. When he isn't working for Oracle or spending time with his wife and four young children, he's tinkering with Linux or playing a friendly online game of *Quake III Arena*.

Joel R. Kallman is a software development manager for Oracle Corporation. Over the past 14 years, he has focused on database and content management, from SGML databases and publishing systems to text and document management. He is currently managing the development of Oracle HTML DB, a solution that allows customers to easily build database-centric Web applications.

When the daily advances in computer technology aren't consuming all his time, Joel enjoys football, woodworking, investing, and working out at the local "Y." Joel is a proud alumnus of The Ohio State University, where he received his bachelor's degree in computer engineering. He and his wife, Kristin, reside in Powell, Ohio.

David C. Knox is the chief engineer for Oracle's Information Assurance Center. He joined Oracle Corporation in June 1995. While at Oracle, he has worked on many security projects for various customers, including the U.S. Department of Defense (DoD), intelligence agencies, state and local governments, financial services organizations, and healthcare organizations. His computer security expertise derives not only from working knowledge and experience with Oracle's security products and database security, but also from his academic studies in the areas of multilevel security, cryptography, LDAP, and PKI. David earned a bachelor's degree in computer science from the University of Maryland and a master's degree in computer science from Johns Hopkins University.

About the Technical Reviewers

Jakob Hammer-Jakobsen was born in 1965. He earned his master's degree in 1992 and has been working with Oracle since 1986 (starting with Oracle version 5). He has worked primarily as developer of business systems on Oracle (and other databases), but over the last 5 years he's moved to the DBA segment as well. Jakob has taught all kinds of Oracle-related courses worldwide; his most recent course was "Developing Java Portlets." He is an Oracle Certified Developer and a member of OakTable.net. Other organizations he's worked for include the Department of Higher Education, University of Roskilde; Denmark's International Student Foundation (housing); Tom Pedersen International (the original distributor of Oracle in Europe); Oracle Denmark; Miracle Australia; and Miracle Denmark.

Torben Holm is a member of the OakTable network. He has been in the computer business as a developer since 1998, as a staff sergeant in the Royal Danish Airforce. He has been working with Oracle since 1992—his first 4 years as system analyst and application developer (Oracle 7 and Forms 4.0/Reports 2.0 and DBA), then 2 years as developer (Oracle6/7, Forms 3.0 and RPT, and DBA). He then worked 2 years in Oracle Denmark in the Premium Services group as a senior principal consultant, where he performed application development and DBA tasks. He worked as an instructor in PL/SQL, SQL, DBA, and WebDB courses. For the last 3 years, Torben has worked for Miracle A/S (<http://www.miracleas.dk/>) as application developer and DBA. He is Developer 6i Certified (and partly 8i Certified, for what it's worth—he didn't have the time to finish that certification). His "main" language is PL/SQL.

Tom Kyte is VP, Core Technologies at Oracle Corporation, and he has over 16 years of experience designing and developing large-scale database and Internet applications. Tom specializes in core database technologies, application design and architecture, and performance tuning. He is a regular columnist for *Oracle Magazine* and is the Tom behind the AskTom website (<http://asktom.oracle.com/>), where technical professionals can come to get answers to their questions. He's also the author of *Effective Oracle by Design*, an Oracle best practices book, and *Expert One-on-One Oracle*, a book that describes how to architect systems using core Oracle technologies, and he's the coauthor of *Beginning Oracle*, a book aimed at new Oracle developers.

Acknowledgments

First, I want to acknowledge the wonderful help afforded to me over the years by the OakTable network. I've never met a group of people who are so generous with their time and knowledge (and vast knowledge it is indeed!). It's a privilege to be associated with them. In particular, I want to thank Tom Kyte, Jonathan Lewis, and Dave Ensor, all of whom inspired me to explore Oracle more deeply, and Mogens Nørgaard, an OakTable network "founder," for his amazing hospitality and whiskey! Thanks also to my editor, Tony Davis, toward whom my emotions have oscillated between gratitude and wanting to knock him senseless, which pretty much means he's doing his job well.

I would also like to thank the companies I've worked with that use Oracle. There is no better way of exploring the power of PL/SQL and other Oracle technologies than to be challenged with solving problems and optimizing the use of the Oracle infrastructure.

Most important, I want to thank my wife, Gillian, for her support and tolerance of the time I spend exploring Oracle—time I should be spending with her! Exploring technology often means being locked away for hours or days in a dark room in front of a computer screen, but knowing that you've got the most beautiful and wonderful woman in the world just a few steps away in the next room makes finding inspiration easy.

—Connor McDonald

I would like to thank Tony Davis at Apress for sticking with me through this process, even though my submissions were always very late. Sorry again, Tony. I would also like to thank my wife, Marta, for her constant encouragement and support. *Ti amo*.

—Christopher Beck

I would first like to thank my wife, Sandy, for supporting me in writing this book. I would have never been able to do it without her allowing me to work on this project during "vacation." I would also like to acknowledge my colleagues at Oracle—Tom Kyte and Patrick Sack, in particular—for their technical perspectives and valuable input. I would also like to thank my editor, Tony Davis; my fellow authors; and the technical reviewers, all of whom have helped to make this book a success.

—David C. Knox

Introduction

I went to an online bookstore recently, typed **PL/SQL** in the Search box, and got 38 results back, excluding this book. Thirty-eight books! As far as I could see, none of them was listed alongside the Harry Potter books as worldwide top-sellers, so what on earth would inspire a group of authors to come together to produce the thirty-ninth book on this topic?

The reason is that, despite the plethora of available books, we still encounter a great deal of poor quality or antiquated PL/SQL code in Oracle applications. From a personal perspective, I've worked with Oracle systems around the world, and although the applications, architectures, and methodologies have been very diverse, I've found two common themes in almost all of these systems. Either they steer away from Oracle-specific functionality altogether, or they use it in a haphazard and less-than-optimal fashion. Nowhere is this more evident than in the use of PL/SQL, which has been less “used” and more “abused” in many of the systems that I've encountered.

At least part of the problem is that the majority of PL/SQL books are only about syntax. They'll show you how to code PL/SQL so that it will compile and execute on your systems (some books extend themselves to giving guidelines for good naming standards and coding structure). But, as with any programming language, there's a big difference between just using the language and using it well. The key to building successful applications is the ability to take your syntax knowledge and apply it intelligently to build programs that are robust, efficient, and easily maintained. This is the motivation for our book and its title. We don't want to make you a PL/SQL programmer—we want to make you a *smart* PL/SQL programmer.

What Does This Book Cover?

This book offers a wealth of tips, techniques, and complete strategies for maximizing the benefits of PL/SQL within your organization. By the end of this book, you'll be as convinced as we are that PL/SQL isn't just a useful tool—it's an integral part of any Oracle application you'll ever develop.

We'll demonstrate techniques that are applicable for all versions of Oracle, from 8i to 10g. The vast majority of the examples in this book were tested using Oracle9i R2, and all you'll need to run them is SQL*Plus.

The following is a chapter-by-chapter breakdown that summarizes some of the key topics we'll cover:

- **Setting Up.** The next section of this book shows you how to set up an effective SQL*Plus environment and how to get up and running with the performance tools that we use throughout the book, namely AUTOTRACE, SQL_TRACE, TKPROF, and RUNSTATS.
- **Chapter 1: Efficient PL/SQL.** This chapter defines what we mean by “efficient PL/SQL” and introduces the book’s pervading theme of *demonstrability*—that is, the need to prove conclusively that your code meets performance targets under *all* reasonable conditions. It demonstrates why PL/SQL is almost always the right tool for programming within the database, but it also explores situations in which PL/SQL might not be appropriate, by presenting a few innovative uses of SQL as a means to avoiding procedural code altogether.
- **Chapter 2: Package It All Up.** Packages are much more than just a “logical grouping of procedures.” They offer numerous advantages, from overloading and encapsulation to protection from dependency and recompilation issues. This chapter clearly demonstrates these advantages and also discusses interesting uses for some of the Oracle-supplied packages.
- **Chapter 3: The Vexed Subject of Cursors.** There is much debate and contention surrounding the issue of explicit versus implicit cursors. This chapter demonstrates why you might not need explicit cursors as often as you may think. It also looks at effective uses of cursor variables and cursor expressions in distributed applications.
- **Chapter 4: Effective Data Handling.** This chapter shows you how to maximize the integration between the data structures in the database and the data structures in your PL/SQL program, leading to code that is more robust and resilient to change. It also looks at how to make effective use of collections in passing data in bulk from your program to the database and vice versa.
- **Chapter 5: PL/SQL Optimization Techniques.** This chapter provides a number of ready-made solutions to some commonly encountered problems in PL/SQL development. It shows you how to avoid some of the hidden overheads and highlights “gotchas” that can trip up the unwary.
- **Chapter 6: Triggers.** This chapter covers fundamental trigger concepts and effective uses for some of the various types of triggers available. It also delves into the relatively new topic of Oracle Streams and shows how to use them to implement a centralized data audit trail.

- **Chapter 7: DBA Packages.** This chapter provides a “DBA toolkit”—a set of packages that can be used to automate recurring administrative activities, such as performance diagnosis and troubleshooting, backup and recovery, and monitoring the database for faults.
- **Chapter 8: Security Packages.** This chapter looks at the use of PL/SQL packages and triggers to implement effective security mechanisms in the database. It covers fundamental issues such as use of the invoker and definer rights models, package construction, and schema design, and then it moves on to present specific solutions for such issues as auditing database activity and protecting your source code.
- **Chapter 9: Web Packages.** This chapter investigates a set of built-in database packages, collectively known as the PL/SQL Web Toolkit, which allow developers to present dynamic Web pages from directly within the database. It covers issues such as use of cookies, management of tables and files, and how to invoke a Web Service directly from within a PL/SQL stored procedure.
- **Chapter 10: PL/SQL Debugging.** Few people get it right first time, so this chapter presents a range of techniques for effective debugging of your PL/SQL code, from the simple use of `DBMS_OUTPUT` to more complex packages such as `DBMS_APPLICATION_INFO` and `UTL_FILE`. It culminates with the development of `DEBUG`, a sophisticated custom debugging utility.
- **Appendix A: Building `DEBUG`.** This appendix presents the full code listing for the `DEBUG` utility used in Chapter 10.

Who Should Read This Book?

This book is targeted primarily toward the DBA or developer charged with the implementation of effective data handling, security, and database administration mechanisms in the Oracle database. However, it will also have great appeal to *any* developer whose applications rely on an Oracle database and who needs a sound understanding of how to use PL/SQL effectively.

If you're brand new to PL/SQL, then you'll want to take some time to get familiar with the language before tackling this book. It's not for the total beginner. But once you're up and running, we believe you'll find our book an invaluable guide for ensuring that the PL/SQL solutions you build are robust, perform well, and are easy to maintain.

—Connor McDonald

Setting Up

In this section we'll describe how to set up an environment capable of executing the examples in this book. We'll cover the following topics:

- How to set up the SCOTT/TIGER demonstration schema
- How to configure the SQL*Plus environment
- How to configure AUTOTRACE, a SQL*Plus facility that shows you either how Oracle performed a query or how it will perform the query, along with statistics regarding the processing of that query
- How to set up to use SQL_TRACE, TIMED_STATISTICS, and TKPROF, two parameters and a command-line tool that will tell you what SQL your application executed and how that SQL performed
- How to set up and use the RUNSTATS utility

Note that we provide only basic setup instructions here for the various performance tools, so that you may quickly configure your environment to run to the examples in this book. For full instructions and information on how to interpret the data that these tools provide, we refer you to the Oracle documentation set or to a relevant book, such as Thomas Kyte's *Expert One-on-One Oracle* (Apress, ISBN: 1-59059-243-3).

Setting Up the SCOTT/TIGER Schema

Many of the examples in this book draw on the EMP/DEPT tables in the SCOTT schema. We recommend that you install your own copy of these tables in some account other than SCOTT to avoid side effects caused by other users using and modifying the same data. To create the SCOTT demonstration tables in your own schema, simply perform the following:

1. From the command line, run `cd [ORACLE_HOME]/sqlplus/demo.`
2. Log into SQL*Plus as the required user.
3. Run `@DEMOBLD.SQL.`

The DEMOBLD.SQL script will create and populate five tables for you. When it's complete, it exits SQL*Plus automatically, so don't be surprised when SQL*Plus disappears after running the script. If you would like to drop this schema at any time to clean up, you can simply execute [ORACLE_HOME]/sqlplus/demo/demodrop.sql.

The SQL*Plus Environment

The examples in this book are designed to run in the SQL*Plus environment. SQL*Plus provides many useful options and commands that we'll make frequent use of throughout this book. For example, a lot of the examples in this book use DBMS_OUTPUT in some fashion. In order for DBMS_OUTPUT to work, the following SQL*Plus command must be issued:

```
SQL> set serveroutput on
```

Alternatively, SQL*Plus allows us to set up a LOGIN.SQL file, a script that is executed each and every time we start a SQL*Plus session. In this file, we can set parameters such as SERVEROUTPUT automatically. An example of a LOGIN.SQL script is as follows (you can edit it to suit your own particular environment):

```
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
```

Furthermore, we can use this script to format our SQL*Plus prompt so that we always know who we're logged in as and on which database. For example, as you work through this book, you'll encounter prompts of the following format:

```
scott@oracle9i_test>
```

This tells you that you're logged into the SCOTT schema on the ORACLE9I_TEST database. The following is the code in the LOGIN.SQL script that achieves this:

```
column global_name new_value gname
set termout off
select lower(user) || '@' ||
```

```
global_name from global_name;  
set sqlprompt '&gname> '  
set termout on
```

This login script will only be run once, on startup. So, if you login on startup as SCOTT and then change to a different account, this won't register on your prompt:

```
SQL*Plus: Release 8.1.7.0.0 - Production on Sun Mar 16 15:02:21 2003
```

```
(c) Copyright 2000 Oracle Corporation. All rights reserved.
```

```
Enter user-name: scott/tiger
```

```
Connected to:  
Personal Oracle8i Release 8.1.7.0.0 - Production  
With the Partitioning option  
JServer Release 8.1.7.0.0 - Production
```

```
scott@ORATEST> connect tony/davis  
Connected.  
scott@ORATEST>
```

The following CONNECT.SQL script will solve this:

```
set termout off  
connect &1  
@login  
set termout on
```

Then you simply run this script (which connects, and then runs the login script) every time you want to change accounts:

```
scott@ORATEST> @connect tony/davis  
tony@ORATEST>
```

To get SQL*Plus to run the login script automatically on startup, you need to save it in a directory (put CONNECT.SQL in the same directory) and then set the SQLPATH environment variable to point at that directory. If you're working on Windows, navigate to the Start button, select Run, and type **regedit**. Navigate to HKEY_LOCAL_MACHINE/SOFTWARE/ORACLE and find the SQLPATH file (mine was in HOME0). Double-click it and set the path to the directory where you stored the scripts (for example, C:\oracle\ora81\sqlplus\admin).

Setting Up AUTOTRACE in SQL*Plus

Throughout the book it will be useful for us to monitor the performance of the queries we execute. SQL*Plus provides an AUTOTRACE facility that allows us to see the execution plans of the queries we've executed and the resources they used. The report is generated after successful SQL DML. This book makes extensive use of this facility. There is more than one way to configure the AUTOTRACE facility, but the following is a recommended route:

1. Access `cd $ORACLE_HOME/rdbms/admin.`
2. Log into SQL*Plus as any user with `CREATE TABLE` and `CREATE PUBLIC SYNONYM` privileges.
3. Run `@UTLXPLAN` to create a `PLAN_TABLE` for use by AUTOTRACE.
4. Run `CREATE PUBLIC SYNONYM PLAN_TABLE FOR PLAN_TABLE`, so that everyone can access this table without specifying a schema.
5. Run `GRANT ALL ON PLAN_TABLE TO PUBLIC`, so that everyone can use this table.
6. Exit SQL*Plus and change directories as follows:
`cd $ORACLE_HOME/sqlplus/admin.`
7. Log into SQL*Plus as a SYSDBA.
8. Run `@PLUSTRCE.`
9. Run `GRANT PLUSTRACE TO PUBLIC.`

You can test your setup by enabling AUTOTRACE and executing a simple query:

```
SQL> set AUTOTRACE traceonly
SQL> select * from emp, dept
      2  where emp.deptno=dept.deptno;
```

14 rows selected.

Execution Plan

```
-----
0      SELECT STATEMENT Optimizer=CHOOSE
1      0  MERGE JOIN
```

```

2    1    SORT (JOIN)
3    2    TABLE ACCESS (FULL) OF 'DEPT'
4    1    SORT (JOIN)
5    4    TABLE ACCESS (FULL) OF 'EMP'

```

Statistics

```

-----
      0 recursive calls
      8 db block gets
      2 consistent gets
      0 physical reads
      0 redo size
  2144 bytes sent via SQL*Net to client
   425 bytes received via SQL*Net from client
      2 SQL*Net roundtrips to/from client
      2 sorts (memory)
      0 sorts (disk)
     14 rows processed

```

```
SQL> set AUTOTRACE off
```

For full details on the use of AUTOTRACE and interpretation of the data it provides, see Chapter 11 of *Oracle9i Database Performance Tuning Guide and Reference* in the Oracle documentation set or Chapter 9 of *SQL*Plus User's Guide and Reference*.

Performance Tools

In addition to using AUTOTRACE, we make use of various other performance tools throughout the book. We'll present brief setup instructions in this section.

TIMED_STATISTICS

The TIMED_STATISTICS parameter specifies whether Oracle should measure the execution time for various internal operations. Without this parameter set, there is much less value to the trace file output. As with other parameters, you can set TIMED_STATISTICS either on an instance level (in INIT.ORA) or on a session level. The former shouldn't affect performance, so it's generally recommended. Simply

add the following line to your `INIT.ORA` file and then the next time you restart the database, it will be enabled:

```
timed_statistics=true
```

On a session level, you would issue this:

```
SQL> alter session set timed_statistics=true;
```

SQL_TRACE and TKPROF

Together, the `SQL_TRACE` facility and the `TKPROF` command-line utility enable detailed tracing of the activity that takes place within the database. In short, `SQL_TRACE` is used to write performance information on individual SQL statements down to trace files in the file system of the database server. Under normal circumstances, these trace files are hard to comprehend directly. For that purpose, you use the `TKPROF` utility to generate text-based report files from the input of a given trace file.

SQL_TRACE

The `SQL_TRACE` facility is used to trace all SQL activity of a specified database session or instance down to a trace file in the database server operating system. Each entry in the trace file records a specific operation performed while the Oracle server process is processing a SQL statement. `SQL_TRACE` was originally intended for debugging, and it's still well suited for that purpose, but it can just as easily be used to analyze the SQL activity of the database for tuning purposes.

Setting Up SQL_TRACE

`SQL_TRACE` can be enabled for either a single session or a whole database instance. It is, however, rarely enabled at a database level, because that would cause serious performance problems. Remember that `SQL_TRACE` writes down every SQL statement processed down to a log file, with accompanying I/O activity.

To enable tracing for the current session, you should issue `ALTER SESSION`, as shown here:

```
SQL> alter session set sql_trace=true;
```


Enable tracing for a session at a selected interval and avoid having tracing in effect for long periods of time. To disable the current trace operation, you execute the following:

```
SQL> alter session set sql_trace=false;
```

Controlling the Trace Files

The trace files generated by SQL_TRACE can eventually grow quite large. A few global initialization parameters, set in INIT.ORA for the database instance or session settings, affect the trace files. If enabled, SQL_TRACE will write to a file in the operating system directory indicated by the USER_DUMP_DEST initialization parameter. You should note that trace files for USER processes (dedicated servers) go to the USER_DUMP_DEST directory. Trace files generated by Oracle background processes such as the shared servers used with MTS and job queue processes used with the job queues will go to the BACKGROUND_DUMP_DEST. Use of SQL_TRACE with a shared server configuration isn't recommended. Your session will hop from shared server to shared server, generating trace information in not one but in many trace files, rendering it useless.

Trace files are usually named

```
ora<spid>.trc,
```

where <spid> is the server process ID of the session for which the trace was enabled. On Windows, the following query may be used to retrieve your session's trace file name:

```
SQL> select c.value || '\ORA' || to_char(a.spid,'fm00000') || '.trc'
 2   from v$process a, v$session b, v$parameter c
 3   where a.addr = b.paddr
 4         and b.audsid = userenv('sessionid')
 5         and c.name = 'user_dump_dest';
```

On Unix, this query can be used to retrieve the session's trace file name:

```
SQL> select c.value || '/' || d.instance_name || '_ora_' ||
 2         to_char(a.spid,'fm99999') || '.trc'
 3   from v$process a, v$session b, v$parameter c, v$instance d
 4   where a.addr = b.paddr
 5         and b.audsid = userenv('sessionid')
 6         and c.name = 'user_dump_dest';
```

The size of the trace files is restricted by the value of the `MAX_DUMP_FILE_SIZE` initialization parameter set in `INIT.ORA` for the database instance. You may also alter this at the session level using the `ALTER SESSION` command, for example:

```
SQL> alter session set max_dump_file_size = unlimited;
Session altered.
```

TKPROF

The `TKPROF` utility takes a `SQL_TRACE` trace file as input and produces a text-based report file as output. It's a very simple utility, summarizing a large set of detailed information in a given trace file so that it can be understood for performance tuning.

Using TKPROF

`TKPROF` is a simple command-line utility that is used to translate a raw trace file to a more comprehensible report. In its simplest form, `TKPROF` can be used as shown here:

```
tkprof <trace-file-name> <report-file-name>
```

To illustrate the joint use of `TKPROF` and `SQL_TRACE`, we'll set up a simple example. Specifically, we'll trace the query we used previously in our `AUTOTRACE` example and generate a report from the resulting trace file. First, we log onto `SQL*Plus` as the intended user and then execute the following code:

```
SQL> select c.value || '\ORA' || to_char(a.spid,'fm00000') || '.trc'
 2      from v$sqlprocess a, v$sqlsession b, v$sqlparameter c
 3      where a.addr = b.paddr
 4             and b.audsid = userenv('sessionid')
 5             and c.name = 'user_dump_dest';
```

```
C.VALUE||'\ORA'||TO_CHAR(A.SPID,'FM00000')||'.TRC'
```

```
-----
C:\oracle\admin\oratest\udump\ORA01528.trc
```

```
SQL> alter session set timed_statistics=true;
```

```
Session altered.
```

```
SQL> alter session set sql_trace=true;
```

Session altered.

```
SQL> select * from emp, dept
      2  where emp.deptno=dept.deptno;
```

```
SQL> alter session set sql_trace=false;
```

```
SQL> exit
```

Now, we simply format our trace file from the command line using TKPROF, as follows:

```
C:\oracle\admin\oratest\udump>tkprof ORA01528.TRC tkprof_rep1.txt
```

Now we can open the TKPROF_REP1.TXT file and view the report. We don't intend to discuss the output in detail here, but briefly, at the top of the report we should see the actual SQL statement issued. Next, we get the execution report for the statement. This report is illustrated for the three different phases of Oracle SQL processing: parse, execute, and fetch. For each processing phase, we see the following:

- The number of times that phase occurred
- The CPU time elapsed for the phase
- The real-world time that elapsed
- The number of physical I/O operations that took place on the disk
- The number of blocks processed in “consistent-read” mode
- The number of blocks read in “current” mode (reads that occur when the data is changed by an external process during the duration of the statement processing)
- The number of blocks that were affected by the statement

The execution report is as follows:

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.01	0.02	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	2	0.00	0.00	0	2	8	14
total	4	0.01	0.02	0	2	8	14

Following the execution report, we can see optimizer approach used and the user ID of the session that enabled the trace (we can match this ID against the ALL_USERS table to get the actual username):

```
Misses in library cache during parse: 0
Optimizer goal: CHOOSE
Parsing user id: 52
```

Additionally, we see the number of times the statement wasn't found in the library cache. The first time a statement is executed, this count should be 1, but it should be 0 in subsequent calls if bind variables are used. Again, watch for the absence of bind variables—a large number of library cache misses would indicate that.

Finally, the report displays the execution plan used for this statement. This information is similar to that provided by AUTOTRACE, with the important difference that the number of actual rows flowing out of each step in the plan is revealed to us:

Rows	Row Source	Operation
14	MERGE JOIN	
5	SORT JOIN	
4	TABLE ACCESS FULL DEPT	
14	SORT JOIN	
14	TABLE ACCESS FULL EMP	

For full details on use of SQL_TRACE and TKPROF, and interpretation of the trace data, see Chapter 10 of *Oracle9i Database Performance Tuning Guide and Reference*.

RUNSTATS

RUNSTATS is a simple test harness that allows comparison of two executions of code and displays the costs of each in terms of the elapsed time, session-level

statistics (such as parse calls), and latching differences. The latter of these, latching, is the key piece of information that this tool provides.

NOTE The RUNSTATS tool was originally built by Tom Kyte, the man behind the <http://asktom.oracle.com> website. Full information and an example usage of RUNSTATS can be found at <http://asktom.oracle.com/~tkyte/runstats.html>. In Chapter 4 we provide a useful customization of this tool that makes use of collections.

To run this test harness, you must have access to V\$STATNAME, V\$MYSTAT, and V\$LATCH. You must be granted *direct* SELECT privileges (not via a role) on SYS.V_\$STATNAME, SYS.V_\$MYSTAT, and SYS.V_\$LATCH. You can then create the following view:

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

View created.

All you need then is a small table to store the statistics:

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

The code for the test harness package is as follows:

```
create or replace package runstats_pkg
as
  procedure rs_start;
  procedure rs_middle;
  procedure rs_stop( p_difference_threshold in number default 0 );
end;
/
```

Setting Up

```
create or replace package body runstats_pkg
as

g_start number;
g_run1  number;
g_run2  number;

procedure rs_start
is
begin
    delete from run_stats;

    insert into run_stats
    select 'before', stats.* from stats;

    g_start := dbms_utility.get_time;
end;

procedure rs_middle
is
begin
    g_run1 := (dbms_utility.get_time-g_start);

    insert into run_stats
    select 'after 1', stats.* from stats;
    g_start := dbms_utility.get_time;

end;

procedure rs_stop(p_difference_threshold in number default 0)
is
begin
    g_run2 := (dbms_utility.get_time-g_start);

    dbms_output.put_line
    ( 'Run1 ran in ' || g_run1 || ' hsecs' );
    dbms_output.put_line
    ( 'Run2 ran in ' || g_run2 || ' hsecs' );
    dbms_output.put_line
    ( 'run 1 ran in ' || round(g_run1/g_run2*100,2) ||
      '% of the time' );
    dbms_output.put_line( chr(9) );
```

```

insert into run_stats
select 'after 2', stats.* from stats;

dbms_output.put_line
( rpad( 'Name', 30 ) || lpad( 'Run1', 10 ) ||
  lpad( 'Run2', 10 ) || lpad( 'Diff', 10 ) );

for x in
( select rpad( a.name, 30 ) ||
  to_char( b.value-a.value, '9,999,999' ) ||
  to_char( c.value-b.value, '9,999,999' ) ||
  to_char( ( (c.value-b.value)-(b.value-a.value)), '9,999,999' ) data
  from run_stats a, run_stats b, run_stats c
  where a.name = b.name
  and b.name = c.name
  and a.runid = 'before'
  and b.runid = 'after 1'
  and c.runid = 'after 2'
  and (c.value-a.value) > 0
  and abs( (c.value-b.value) - (b.value-a.value) )
    > p_difference_threshold
  order by abs( (c.value-b.value)-(b.value-a.value) )
) loop
  dbms_output.put_line( x.data );
end loop;

dbms_output.put_line( chr(9) );
dbms_output.put_line
( 'Run1 latches total versus runs -- difference and pct' );
dbms_output.put_line
( lpad( 'Run1', 10 ) || lpad( 'Run2', 10 ) ||
  lpad( 'Diff', 10 ) || lpad( 'Pct', 8 ) );

for x in
( select to_char( run1, '9,999,999' ) ||
  to_char( run2, '9,999,999' ) ||
  to_char( diff, '9,999,999' ) ||
  to_char( round( run1/run2*100,2 ), '999.99' ) || '%' data
  from ( select sum(b.value-a.value) run1, sum(c.value-b.value) run2,
    sum( (c.value-b.value)-(b.value-a.value)) diff
    from run_stats a, run_stats b, run_stats c
    where a.name = b.name
    and b.name = c.name
    and a.runid = 'before'

```

```
                and b.runid = 'after 1'  
                and c.runid = 'after 2'  
                and a.name like 'LATCH%'  
            )  
        ) loop  
            dbms_output.put_line( x.data );  
        end loop;  
end;  
  
end;  
/
```

Using RUNSTATS

To demonstrate the information that we can get out of RUNSTATS, we'll compare the performance of a lookup on a normal heap table (HEAP) and an index-organized table (IOT). We'll consider three scenarios:

- Full table scan on small tables
- Primary key lookup on moderate tables
- Secondary index lookup on moderately sized tables

Full Scan on Small Tables

First we create our tables and indexes:

```
SQL> create table HEAP  
2   as select * from DUAL;
```

Table created.

```
SQL> create table IOT ( dummy primary key)  
2   organization index  
3   as select * from DUAL;
```

Table created.

Now we analyze both tables to ensure consistency in our results:


```
SQL> analyze table HEAP compute statistics;
```

Table analyzed.

```
SQL> analyze table IOT compute statistics;
```

Table analyzed.

Next we perform a preliminary run to massage the cache:

```
SQL> declare
2     x varchar2(1);
3     begin
4         for i in 1 .. 10000 loop
5             select dummy into x
6             from   HEAP;
7         end loop;
8     end;
9 /
```

PL/SQL procedure successfully completed.

```
SQL> declare
2     x varchar2(1);
3     begin
4         for i in 1 .. 10000 loop
5             select dummy into x
6             from   IOT;
7         end loop;
8     end;
9 /
```

PL/SQL procedure successfully completed.

We then take a snapshot of our statistics before we run our tests:

```
SQL> exec RUNSTATS_PKG.rs_start;
```

PL/SQL procedure successfully completed.

Now we run our lookup code for the HEAP table:

Setting Up

```
SQL> declare
  2     x varchar2(1);
  3     begin
  4     for i in 1 .. 10000 loop
  5         select dummy into x
  6         from   HEAP;
  7     end loop;
  8     end;
  9     /
```

PL/SQL procedure successfully completed.

And now another snapshot:

```
SQL> exec RUNSTATS_PKG.rs_middle
```

PL/SQL procedure successfully completed.

And then we run our lookup code for the IOT table:

```
SQL> declare
  2     x varchar2(1);
  3     begin
  4     for i in 1 .. 10000 loop
  5         select dummy into x
  6         from   IOT;
  7     end loop;
  8     end;
  9     /
```

PL/SQL procedure successfully completed.

Then we take our final snapshot and get our comparative statistics:

```
connor@ORATEST> exec RUNSTATS_PKG.rs_stop;
Run1 ran in 130 hsecs
Run2 ran in 74 hsecs
run 1 ran in 175.68% of the time
```

Name	Run1	Run2	Diff
LATCH.checkpoint queue latch	1	2	1
STAT...calls to kcmgas	1	0	-1

STAT...cleanouts and rollbacks	1	0	-1
STAT...immediate (CR) block cl	1	0	-1
STAT...parse time cpu	1	0	-1
<output cropped>...			
LATCH.library cache	20,211	20,089	-122
STAT...redo size	2,472	1,740	-732
STAT...table scan blocks gotte	10,000	0	-10,000
STAT...table scan rows gotten	10,000	0	-10,000
STAT...table scans (short tabl	10,000	0	-10,000
STAT...no work - consistent re	10,009	2	-10,007
STAT...buffer is not pinned co	10,014	3	-10,011
LATCH.undo global data	40,007	4	-40,003
STAT...calls to get snapshot s	50,011	10,002	-40,009
STAT...consistent gets	50,027	10,012	-40,015
STAT...db block gets	120,014	18	-119,996
STAT...session logical reads	170,041	10,030	-160,011
LATCH.cache buffers chains	340,125	20,113	-320,012

Run1 latches total versus runs -- difference and pct

Run1	Run2	Diff	Pct
400,570	40,285	-360,285	994.34%

PL/SQL procedure successfully completed.

Not only do we get faster execution times with the IOT, but we also get a massive reduction on the amount of latching performed in the database, suggesting that use of an IOT in this scenario would provide a much more scalable solution. For *small* table scans, IOTs are better because they don't have the overhead of reading the segment header block a number of times.

Note that the preceding test was run on an Oracle 8.1.7 database. If we repeat the test on an Oracle9i R2 database, we see the following:

Run1 ran in 145 hsecs

Run2 ran in 88 hsecs

run 1 ran in 164.77% of the time

Run1 latches total versus runs -- difference and pct

Run1	Run2	Diff	Pct
113,812	73,762	-40,050	154.30%

So the latching difference is much less marked, but the IOT is still worth considering for a full scan on small tables.

Primary Key Lookup on Moderate Tables

For this test we drop our existing HEAP and IOT tables and re-create them as follows:

```
create table HEAP ( r primary key, padding)
  as select rownum r, rpad(rownum,40) padding
  from all_objects;
```

```
create table IOT ( r primary key, padding)
  organization index
  as select rownum r, rpad(rownum,40)
  from all_objects;
```

The SQL in our lookup code simply changes from

```
select dummy into x
  from [HEAP | IOT];
```

to

```
select padding into x
  from [HEAP | IOT]
 where r = i;
```

With this modification, we perform the tests just as before. The following results were obtained on an 8.1.7 database:

```
Run1 ran in 101 hsecs
Run2 ran in 96 hsecs
run 1 ran in 105.21% of the time
```

```
Run1 latches total versus runs -- difference and pct
  Run1      Run2      Diff      Pct
  50,297    40,669    -9,628   123.67%
```

As we expected, the results here are closer, but the IOT still performs less latching and runs slightly faster than the HEAP.

Secondary Lookup on Moderate Tables

For this test we drop our existing HEAP and IOT tables and re-create them as follows:

```

create table HEAP ( r primary key, c , padding)
  as select rownum r, mod(rownum,5000), rpad(rownum,40) padding
  from all_objects;

create table IOT ( r primary key, c , padding)
  organization index
  as select rownum r, mod(rownum,5000), rpad(rownum,40) padding
  from all_objects;

create index HEAP_IX on HEAP ( c);
create index IOT_IX on IOT ( c);

```

The SQL in our lookup code becomes

```

select max(padding) into x
  from [HEAP|IOT]
  where c = i;

```

The following results were obtained on an 8.1.7 database:

```

Run1 ran in 93 hsecs
Run2 ran in 94 hsecs
run 1 ran in 98.94% of the time

```

Run1 latches total versus runs -- difference and pct

Run1	Run2	Diff	Pct
75,554	75,430	-124 1	00.16%

Here, the degree of latching is very similar, and the HEAP performs marginally the better of the two. IOTs don't have a ROWID, so to read via a secondary index, Oracle typically must read the secondary index and then read via the primary key.

Overall, we hope that this section has demonstrated how useful RUNSTATS can be in benchmarking a number of different solutions.

Package It All Up

VIRTUALLY EVERY APPLICATION development language has some concept of subprograms. Typically they are called “procedures,” “functions,” “routines,” or “sections,” but they all are indicative of a logical or modular grouping of code segments within a program. Similarly, PL/SQL, which is a derivative of Ada, also has the concept of subprograms, but the package mechanism is a wonderful extension to this model, as we will demonstrate in this chapter.

We’ll cover the functionality that packages offer over and above merely grouping program units together, and we’ll look at how this functionality makes your code more resilient to the changes that occur over time to the database structure. We will also look at why developers have steered away from packages in the past and consider the circumstances under which you may want to avoid using packages. Finally, we’ll look at a few of the packages that Oracle supplies with the database to demonstrate that it is always worthwhile to check out the available functionality before embarking on a custom solution.

Basic Benefits of Packages

The main reason packages are not more widely adopted is that users are unaware of the benefits they offer. Most of the applications that use PL/SQL are a mixture of procedures and functions, with the occasional package thrown in. In fact, many a training manual, book, and even the standard Oracle documentation imply that packages are simply a convenient way of grouping PL/SQL components together. The “typical” package example provided in training documentation is something along the lines of

```
create or replace
package emp_maint is
    procedure hire_emp(p_empno number, p_name varchar2);
    procedure fire_emp(p_empno);
    procedure raise_salary(p_empno number, p_salary number);
end;
/
```

The package body implements the underlying code for each of the modules in the package specification.

```

create or replace
package body emp_maint is
  procedure hire_emp(p_empno number, p_name varchar2) is
  begin
    insert into emp ...
  end;

  procedure fire_emp(p_empno) is
  begin
    delete from emp ...
  end;

  procedure raise_salary(p_empno number, p_salary number) is
  begin
    update emp ...
  end;
end;
/

```

There is nothing explicitly *wrong* with using packages in this way.¹ However, such examples give the impression that use of packages is merely a cataloguing convenience. However, with only a little editing of the example, we can highlight some of the true benefits of using packages.

```

create or replace
package emp_maint is
  cursor annual_review_candidates return emp%rowtype;
  v_last_empno_used number;
  procedure hire_emp(p_empno number, p_name varchar2);
  procedure fire_emp(p_empno number);
  procedure raise_salary(p_empno number, p_salary number);
  procedure hire_emp (p_empno number, p_name varchar2,
    p_hiredate date, p_sal number);
end;
/

```

```

create or replace
package body emp_maint is

  cursor annual_review_candidates return emp%rowtype is
    select * from emp where hiredate > trunc(sysdate,'YYYY');

```

1. In past versions of Oracle, if the resulting package was extremely large, there could be problems. See the “It’s a Package, Not a Library” section later in this chapter for more details.


```

v_pkg_execution_count number := 0;

procedure hire_emp(p_empno number, p_name varchar2) is
begin
    insert into emp ...
end;

procedure fire_emp(p_empno number) is
begin
    delete from emp ...
end;

procedure raise_salary(p_empno number, p_salary number) is
begin
    update emp ...
end;

procedure check_sal_limit(p_empno number) is
begin
    ...
end;

procedure hire_emp(p_empno number, p_name varchar2,
    p_hiredate date, p_sal number) is
begin
    check_sal_limit(p_empno);
    insert into emp..
end;

begin
    select empno into v_last_empno_used
from EMP
where hiredate = ( select max(hiredate)
    from emp )
and rownum = 1;
end;
/

```

In the following sections, we'll discuss the highlighted code in this amended package and demonstrate the true power of package use.

Package Overloading

You'll notice that within our revised `emp_maint` package, there are now *two* procedures named `hire_emp`. The first allows us to enter a new employee based on their employee number and name.

```
procedure hire_emp(p_empno number, p_name varchar2);
```

The second requires that we also supply the employee's date of hiring and initial salary.

```
procedure hire_emp(p_empno number, p_name varchar2,
                  p_hiredate date, p_sal number);
```

Packages allow multiple procedures and functions to have the same name, but each instance of the procedure can take a different parameter set. At execution time, the number of parameters passed and their datatypes will determine which version of `hire_emp` is to be executed.

You may be thinking that a viable alternative would be to create a standalone procedure with the additional parameters coded as optional.

```
procedure hire_emp(p_empno number, p_name varchar2,
                  p_hiredate date default null, p_sal number default null);
```

However, you would then need to add code that caters to the case in which the `p_hiredate` or `p_sal` parameter value is supplied, but not *both*. Package overloading allows implicit enforcement of the parameters that must be supplied. Overloading also yields an effective mechanism for adding functionality to your applications in a low risk manner. Consider if you had deployed the first `hire_emp` procedure (the one that takes only employee name and employee number as parameters) throughout hundreds of programs within your enterprise. Say you wanted to build new programs that require new logic (where the additional salary and hire date parameters are allowed). Without overloading, you would need to find every occurrence of the existing `hire_emp` package to make appropriate code alterations. With overloading, the new four-parameter version can be added without difficulty—the new application programs will pick up the new version while existing programs will still successfully use the original two-parameter version.

Public and Private Package Variables

In our definition of `emp_maint`, we now have a public variable.

```
v_last_empno_used number;
```

And a private variable.

```
v_pkg_execution_count number := 0;
```

Packages give greater flexibility in defining the scope of the variables contained within them. Any variable declared in the package specification (in other words, in the `create package` section) is a *public* variable. Once instantiated, public variables maintain their values for the duration of an entire session.

NOTE Recall that a conventional local variable declared within a procedure and any value it may hold is lost as soon as the procedure is completed.

For example, if we called one of the procedures within the `emp_maint` package, which assigned a value of 1000 to the package variable `v_last_empno`, this variable and its value would remain accessible to any other procedures and programs within that connected session, even after the execution of the procedure is completed. Even though a session cannot see any other session's package variables, public package variables are often referred to as “global” variables because they exist for the duration of the session connection.

TIP If persistence is not desired, you can override this behavior by using the `serially_reusable pragma`.

A private package variable is any variable that is declared in the package body. In other words, a variable that is not defined locally to any one of the package body procedures. Private variables are available only to the other procedures within the package. Like public variables, private variables are persistent for the duration of a user session—their values are retained between package calls.

Public and private variables are ideal for implementing any functionality to record session level detail such as cumulative totals, counters, and the like.

Initialization

You may have noticed that the package body itself has its own PL/SQL block. This is the **initialization** section.

```

create or replace
package body emp_maint is
...
...
...
begin
  select empno into v_last_empno_used
  from EMP
  where hiredate = ( select max(hiredate)
                    from emp )
  and rownum = 1;
end;

```

When *any* component of a package is first referenced or executed, the code in the initialization section is processed only once. In our `emp_maint` example, we determine the employee number for the most recently hired person and store that data in `v_last_empno`, which is then available for the remainder of the session. This would not be possible with a standalone procedure because there is no persistence of variables once the procedure is completed.

Information Hiding

Details can be hidden within the package body. This is described as **encapsulation**. There are two such examples in the `emp_maint` package. The first is the `check_sal_limit` procedure.

```

procedure check_sal_limit(p_empno number) is
begin
  ...
end;

```

We can assume that the `check_sal_limit` procedure contains sensitive pay scale information so we do not want people to be aware of that source code. Furthermore, even with the source code hidden, it would not be appropriate for this routine to be called except from within the `hire_emp` procedure, when an initial salary is passed in for scrutiny. If `check_sal_limit` were a standalone procedure, controlling access to it would be extremely difficult. Having it defined only within the package body guarantees the security level we need—it cannot be run except by other routines within the package.

Similarly, the cursor `annual_review_candidates` shows that with packages, we can make the *usage* of a cursor available to the calling environment while keeping the *definition* of that cursor private. In the package specification, we simply declare the existence of the cursor.

```
cursor annual_review_candidates return emp%rowtype;
```

We then fully define it within the package body.

```
cursor annual_review_candidates return emp%rowtype is
  select * from emp where hiredate > trunc(sysdate,'YYYY');
```

Anyone with the appropriate privileges (that is, execution rights on the package) can get a list of the employees who are due for their annual review, but the underlying query is kept secure. The cursor can be used as per any normal cursor definition, not just within the `emp_maint` package. For example, a second procedure, designed for processing the annual review details, might get a list of annual review candidates.

```
procedure process_annual_review is
begin
  for i in emp_maint.annual_review_candidates loop
    ( processing )
  end loop;
end;
```

Notice that in order to take advantage of this facility, an explicit cursor is required. Believe it or not, this is one of the very few times when you *must* use explicit cursor definitions in PL/SQL. Of course, if there were no use for them at all, Oracle would not have invented them in the first place, but consult Chapter 3 for more information about why explicit cursors are perhaps overused in PL/SQL applications.

Standalone Procedures and the Dependency Crisis

The benefits obtained from information hiding, persistent public and private variables, and the initialization section are all inextricably linked to the separation of packages into public and private components. This separation also gives rise to the most significant advantage of packages over standalone procedures: insulation from what we call the “dependency crisis.”

Within Oracle (or any other environment), knowing the interrelationships between objects is vital for assessing the scope of impact when one or more of those objects is to be altered. Oracle exposes these relationships in the `USER_DEPENDENCIES` view, shown as follows:

```
SQL> desc user_dependencies
```

```
Name
-----
NAME
TYPE
REFERENCED_OWNER
REFERENCED_NAME
REFERENCED_TYPE
REFERENCED_LINK_NAME
SCHEMAID
DEPENDENCY_TYPE
```

For example, we can view the dependencies for our `emp_maint` package as follows:

```
SQL> select name, type,
2  referenced_name, referenced_type
3  from user_dependencies
4  where name = 'EMP_MAINT'
5  /
```

NAME	TYPE	REFERENCED_NAME	REFERENCED_TYPE
-----	-----	-----	-----
EMP_MAINT	PACKAGE	STANDARD	PACKAGE
EMP_MAINT	PACKAGE BODY	STANDARD	PACKAGE
EMP_MAINT	PACKAGE	EMP	TABLE
EMP_MAINT	PACKAGE BODY	EMP	TABLE
EMP_MAINT	PACKAGE BODY	EMP_MAINT	PACKAGE

8 rows selected.

Looking at each row in turn, we can see the following dependencies:

- The `emp_maint` *package* depends on the standard package (from which all PL/SQL is based).
- The `emp_maint` *package body* also depends on the standard package.
- The `emp_maint` *package* depends on the `emp` table (because the cursor `annual_review_candidates` refers to it).
- The `emp_maint` *package body* depends on `emp` table (because we insert/delete/update and query it).
- The `emp_maint` *package body* depends on its own package specification.

Thus the `USER_DEPENDENCIES` view consists of a hierarchical tree of parent-child pairs that describes the relationship between objects within the database. Hierarchical tables are typically queried using Oracle's `CONNECT-BY` clause, but check the following information before embarking down that route.

Querying Dependencies Using `CONNECT-BY` Clause

If you try to query `USER_DEPENDENCIES` using the `CONNECT-BY` clause in versions of Oracle prior to 9, you will typically get an SQL error because `CONNECT-BY` is not supported on joins. Even in version 9, elaborate hierarchical queries to `USER_DEPENDENCIES` typically run slowly because of the underlying complexity in the view definition. To overcome this, you can take advantage of the delivered script `$ORACLE_HOME/rdbms/admin/utldtree.sql` to populate a tree-structure of dependencies. For example, to see a tree of dependencies for the table `emp` in the `MCDONAC` schema, use the following:

```
SQL> exec deptree_fill('TABLE','MCDONAC','EMP');
```

```
PL/SQL procedure successfully completed.
```

```
SQL> select * from ideptree;
```

```
DEPENDENCIES
```

```
-----
```

```
TABLE MCDONAC.EMP
      PACKAGE BODY MCDONAC.PKG1
      PROCEDURE MCDONAC.P1
            PROCEDURE MCDONAC.P2
                  PROCEDURE MCDONAC.P3
                        PROCEDURE MCDONAC.P4
                              TRIGGER MCDONAC.EMP_DEPT_CNT_TRIGGER
```

To demonstrate the difference between procedures and packages in terms of the dependency tree within the database, we'll create a few standalone procedures that have some fairly basic dependencies. First we'll create procedure, `P1`, which will count the rows in table `emp`.

```
SQL> create or replace
2  procedure P1 is
3    v_cnt number;
4  begin
```

```
5  select count(*)
6  into   v_cnt
7  from   emp;
8  end;
9  /
```

Procedure created.

Next, we create three more procedures: P2, P3, and P4. Each one simply calls the previous one.

```
SQL> create or replace
2  procedure P2 is
3  begin
4    P1;
5  end;
6  /
```

Procedure created.

```
SQL> create or replace
2  procedure P3 is
3  begin
4    P2;
5  end;
6  /
```

Procedure created.

```
SQL> create or replace
2  procedure P4 is
3  begin
4    P3;
5  end;
6  /
```

Procedure created.

Now we can look at the dependency information from USER_DEPENDENCIES view.

```
SQL> select name, type,
2  referenced_name, referenced_type
3  from user_dependencies
4  /
```


NAME	TYPE	REFERENCED_NAME	REFERENCED_TYPE
-----	-----	-----	-----
P1	PROCEDURE	STANDARD	PACKAGE
P1	PROCEDURE	EMP	TABLE
P2	PROCEDURE	P1	PROCEDURE
P3	PROCEDURE	P2	PROCEDURE
P4	PROCEDURE	P3	PROCEDURE

This is exactly as we would expect: P1 depends on the emp table and STANDARD database package (which is the core component of PL/SQL itself). Procedure P2 depends on P1, P3 depends on P2, and P4 on P3. The importance of the dependency information is that it controls the scope of recompilation required when a parent object (in other words, an object that another object depends on) is altered. To further explain this, we'll make a simple alteration to P1 and gauge its impact on the other procedures. Initially, all our procedures have a status of VALID, as listed in the USER_OBJECTS dictionary view.

```
SQL> select object_name, status
  2  from user_objects
  3  where object_name in ('P1','P2','P3','P4');
```

OBJECT_NAME	STATUS
-----	-----
P4	VALID
P3	VALID
P2	VALID
P1	VALID

We now implement a simple change to the P1 procedure. Any change will do (that is, any change that involves re-creating P1). In this case, we add a clause to the SQL query so that we count only employees with a positive employee number.

```
SQL> create or replace
  2  procedure P1 is
  3  v_cnt number;
  4  begin
  5  select count(*)
  6  into v_cnt
  7  from emp
  8  where empno > 0;
  9  end;
10  /
```

Procedure created.

We then requery the `USER_OBJECTS` view to assess the impact of this change.

```
SQL> select object_name, status
       2  from user_objects
       3  where object_name in ('P1','P2','P3','P4');
```

OBJECT_NAME	STATUS
P4	INVALID
P3	INVALID
P2	INVALID
P1	VALID

Procedure P1 is fine (after all, we have just successfully compiled it), but *all* the procedures that are dependent on P1 now require recompilation. Oracle automatically attempts to recompile a procedure if it is called. For example, if we run the (currently invalid) procedure P3, it will work successfully.

```
SQL> exec p3;
```

PL/SQL procedure successfully completed.

Now we look at the status of each of our procedures after P3 was executed.

```
SQL> select object_name, status
       2  from user_objects
       3  where object_name in ('P1','P2','P3','P4');
```

OBJECT_NAME	STATUS
P4	INVALID
P3	VALID
P2	VALID
P1	VALID

We can see that procedure P2 must be recompiled before P3 can be recompiled and finally executed. P4 remains invalid because it is yet to be called. This sets you up for a maintenance nightmare—it could be days, weeks, or months before P4 is called again, and what if it refuses to compile? Informing the end users that there is a flaw in one of their programs that has been lying around undetected for months and an emergency fix is needed will not be a pleasant task.

One way to avoid this scenario is to write a script that cycles through `user_objects` repeatedly and manually recompiles all invalid objects. You could also use the `utl_recomp` package (available in version 10g) for a procedural mechanism.

Similarly, you might think that a little bit of recompilation here and there is not going to cause you any great problems, but consider a project of more realistic size in which you may have hundreds or thousands of interwoven PL/SQL modules. Let's extend the example to model a system that contains several hundred PL/SQL procedures. We'll start with a simple procedure, `P0`, that does nothing and so will not be dependent on anything except the core standard package.

```
SQL> create or replace
  2  procedure p0 is
  3  begin
  4    null;
  5  end;
  6  /
```

Procedure created.

Rather than write 100 individual procedure modules, we'll use a little bit of PL/SQL to dynamically generate more PL/SQL procedures. The following processing loop creates 50 procedures, named `prc_0001` to `prc_0050`, each of which simply calls `P0`.

```
SQL> declare
  2  x varchar2(32767);
  3  y varchar2(32767);
  4  begin
  5  for i in 1 .. 50 loop
  6    execute immediate
  7      'create or replace procedure prc_'||to_char(i,'fm0000')||
  8      ' is begin p0; end;';
```

The next processing loop creates 50 more procedures, named `prc_0051` to `prc_0100`. Each of these procedure calls each of the 50 procedures, `prc_0001` to `prc_0050`, which we created in the previous step.

```
  9  x := x || 'prc_'||to_char(i,'fm0000')||'; ';
 10  end loop;
 11  for i in 51 .. 100 loop
 12    execute immediate
 13      'create or replace procedure prc_'||to_char(i,'fm0000')||
 14      ' is begin '||x||' end;';
```

Finally, we create a single procedure, `prc_main`, which calls each of the 50 procedures `prc_0051` to `prc_0100`.

```

15     y := y || 'prc_' || to_char(i,'fm0000') || '; ';
16 end loop;
17 execute immediate
18     'create or replace procedure prc_main ' ||
19     ' is begin ' || y || ' end;';
20 end;
21 /

```

PL/SQL procedure successfully completed.

We have only 102 procedures, but we have already created a massive dependency chain.

```

SQL> select name, type,
2     referenced_name, referenced_type
3     from user_dependencies
4     where name like 'PRC____'

```

NAME	TYPE	REFERENCED_NAME	REFERENCED_TYPE
PRC_0066	PROCEDURE	PRC_0023	PROCEDURE
PRC_0067	PROCEDURE	PRC_0023	PROCEDURE
...			
...			

2600 rows selected.

All it takes is a small change to a highly nested standalone procedure to create a massive recompilation overhead. Let's change the `P0` procedure.

```

SQL> create or replace
2     procedure P0 is
3     x number;
4     begin
5     x := 1;
6     end;
7     /

```

Procedure created.

Because all other procedures rely on `P0` either directly or indirectly, every procedure has become invalid.

```
SQL> select object_name, status
       2  from user_Objects
       3  where object_name like 'PRC%'
       4  /
```

OBJECT_NAME	STATUS
-----	-----
PRC0001	INVALID
PRC0002	INVALID
PRC0003	INVALID
...	
PRC_MAIN	INVALID

101 rows selected.

In this extreme case, your entire PL/SQL project must be recompiled either manually or by letting Oracle do the work on an as used basis, as described earlier.

The Cost of Recompilation

Keep in mind that recompilation is resource intensive. We can monitor some session-level statistics that will compare the difference between running `prc_main` under normal conditions versus when the massive recompilation is required. To record the resource costs, we will take a before and after snapshot of the `V$MYSTATS`² view. The deltas between these two snapshots will show the various components of resource usage. We store our first snapshot in table `x1`.

```
SQL> create table x1 as select * from v$mystats;
```

Table created.

Now we run `prc_main`. Due to the change we just made to `P0`, all the `prc_xxxx` procedures will be recompiled on the fly.

```
SQL> exec prc_main;
```

PL/SQL procedure successfully completed.

Now we take a second snapshot of session-level statistics and store these in table `x2`.

2. See chapter 1 for instructions on how to create the `V$MYSTATS` view.

```
SQL> create table x2 as select * from v$mystats;
```

Table created.

Next, we re-run `prc_main`. All the dependent procedures (as well as `prc_main`) have now been recompiled, so this will be a normal execution.

```
SQL> exec prc_main;
```

PL/SQL procedure successfully completed.

We take a final snapshot of statistics in table `x3`.

```
SQL> create table x3 as select * from v$mystats;
```

Table created.

The difference between the statistics in tables `x2` and `x1` represents the resource cost for running `prc_main` with a full recompilation, and the difference between the statistics in tables `x3` and `x2` represents normal operation (no recompilation). The simple SQL shown in the following listing presents the data side-by-side.

```
SQL> select x2.name, x3.value-x2.value NORMAL_RUN,
2         x2.value-x1.value WITH_RECOMP
3   from x1, x2, x3
4  where x2.name = x1.name
5        and x3.name = x2.name
6        order by 3;
```

(some selected results shown)

NAME	NORMAL_	RUN WITH_RECOMP
-----	-----	-----
parse count (hard)	1	103
sorts (memory)	2	103
workarea executions - optimal	5	207
recursive cpu usage	2	1394
CPU used by this session	3	1401
parse time cpu	1	1439
parse time elapsed	1	1476
opened cursors cumulative	20	1535
enqueue releases	17	1635
enqueue requests	17	1635

parse count (total)	18	5068
consistent gets - examination	19	5284
execute count	22	10372
consistent gets	50	10853
redo entries	37	30539
db block gets	43	46627
recursive calls	273	57059
session logical reads	93	57480
db block changes	56	60250
table scan rows gotten	2	75550
redo size	7748	7017416

We will not address each row in isolation, but it is relatively easy to see that all the critical performance areas of an Oracle database (parsing, redo, CPU usage, and enqueue requests) take a significant hit under recompilation. Even Oracle has recognized the massive cost of recompilation by adding enhancements in version 10g whereby invalidations are minimized under certain scenarios, such as synonym manipulation.

NOTE Savvy administrators may have noticed the appearance of the new event, 10520, which is used when applying patchsets in versions 8i and 9i when the entire PL/SQL infrastructure is reloaded via the standard PL/SQL installation script `catproc.sql`. When event 10520 is set and a PL/SQL module is replaced, the incoming source is checked against the existing source in the data dictionary. If they are identical (that is, the source has not changed), compilation is skipped and thus invalidation of dependent objects does not occur. But this does not assist in the more general case in which a non-PL/SQL object (a table, view, and the like) is altered—all dependent PL/SQL modules will be marked for recompilation.

Breaking the Dependency Chain

The division between package specification and package body allows us to break the dependency linkages and avoid the need to worry about excessive recompilation problems. Let's return to our first simple example with four procedures: P1, P2, P3, and P4. We'll put each one in its own package and repeat the tests. Procedure P1 will be placed into package PKG1, procedure P2 into package PKG2 and so on.

```
SQL> create or replace package PKG1 is
  2   procedure P1; end;
  3   /
```

Package created.

```
SQL> create or replace package body PKG1 is
  2   procedure P1 is
  3     v_cnt number;
  4   begin
  5     select count(*)
  6     into v_cnt
  7     from emp;
  8   end;
  9 end;
 10 /
```

Package body created.

```
SQL> create or replace package PKG2 is
  2   procedure P2; end;
  3   /
```

Package created.

```
SQL> create or replace package body PKG2 is
  2   procedure P2 is
  3   begin
  4     PKG1.P1;
  5   end;
  6 end;
  7 /
```

Package body created.

```
SQL> create or replace package PKG3 is
  2   procedure P3; end;
  3   /
```

Package created.

```
SQL> create or replace package body PKG3 is
  2   procedure P3 is
  3   begin
```



```

4     PKG2.P2;
5     end;
6 end;
7 /

```

Package body created.

```

SQL> create or replace package PKG4 is
2     procedure P4; end;
3 /

```

Package created.

```

SQL> create or replace package body PKG4 is
2     procedure P4 is
3     begin
4         PKG3.P3;
5     end;
6 end;
7 /

```

Package body created.

Thus from a functional perspective, the four new packages perform identically to our original four procedures. Now we will make the same simple SQL modification to procedure P1 (now PKG1.P1) so that we count only employees with a positive employee number.

```

SQL> create or replace package body PKG1 is
2     procedure P1 is
3         v_cnt number;
4     begin
5         select count(*)
6         into   v_cnt
7         from   emp
8         where  empno > 0;
9     end;
10 end;
11 /

```

Package body created.

When we look at the status of the four packages, we get a very interesting result.

```
SQL> select object_name, object_type, status
       2  from user_objects
       3  where object_name in ('PKG1','PKG2','PKG3','PKG4');
```

OBJECT_NAME	OBJECT_TYPE	STATUS
PKG4	PACKAGE	VALID
PKG4	PACKAGE BODY	VALID
PKG3	PACKAGE	VALID
PKG3	PACKAGE BODY	VALID
PKG2	PACKAGE	VALID
PKG2	PACKAGE BODY	VALID
PKG1	PACKAGE	VALID
PKG1	PACKAGE BODY	VALID

Everything is valid! There is no recompilation needed at all. Any stored program that references a package depends only upon the package *specification*, thus changes to the package *body* do not cause dependency chain violations. This simple separation, achieved by simply moving to packages for all your production code, provides insulation from massive dependency chains, which are of course typical in any project of reasonable size.

Avoiding a Single Dependency Point

Although having thorough control of the dependency hierarchy is an admirable goal, sometimes it can be difficult to avoid having one or two packages upon which everything in your application depends. One apposite example of this is the practice of storing common application constants in a single package for use by all other PL/SQL components.

CAUTION Too often this practice is based on the misplaced assumption that storing reference codes in a database table will somehow be inherently less efficient than using a PL/SQL variable. However, there are some valid reasons for maintaining a set of global variables within PL/SQL, such as storing application error codes and messages for use within all PL/SQL programs in your application.

A typical package of global variables could look something like this.

```

create or replace
package globals is
  g_gender_m constant char(1) := 'M';
  g_gender_f constant char(1) := 'F';
  g_error_msg_misc constant varchar2(80)
      := 'An unknown error has occurred';
  g_error_msg_gender constant varchar2(80)
      := 'Gender must be '|| g_gender_m|| ' or '|| g_gender_f;
  ...
  ...
end;

```

If you have a very well specified and controlled application environment, having a `globals` package of this nature can be a very efficient means of managing application constants. The emphasis here is on *well specified* because the likelihood of changes to the contents of the `globals` package over time should be quite low (or at least well managed and controlled). However, many projects don't have that luxury, and additions to the `globals` package can occur on an ad hoc basis.

Of course, the problem with frequent change to a package such as `globals` is that there is a high probability that every single PL/SQL module within your system will have a dependency on one or more of these global variables. After all, that is why you create such a package in the first place. To be able to reference the variables, they must be declared within the package *specification*, and thus the addition of a new global variable could easily invalidate *all* the application PL/SQL.

Solving this problem is not trivial. We still recommend that the best solution is one of the following:

- Ensure that the package is not being used where a database table would be more appropriate
- Increase the level of control and management so that global variable changes are made rarely

Let's explore some options you could possibly consider to lessen your exposure to a massive invalidation crisis. First, we'll create a `globals` package in the normal way so we can compare the conventional implementation with some variants.

```

SQL> create or replace
2 package globals is
3   g_1 constant number(5) := 1;
4   g_2 constant number(5) := 1;

```

```
5   g_3 constant number(5) := 1;
6   end;
7   /
```

Package created.

These global package variables will typically be referenced from other parts of the application, or they may even be used within SQL. To cater for each possibility, we create two functions that represent such typical usage of a global variable. First, a function that simply references one of the global variables.

```
SQL> create or replace
2   function use_g1 return number is
3   begin
4     return globals.g_1;
5   end;
6   /
```

Function created.

Now we create a second function that uses the global variables within an SQL statement.

```
SQL> create or replace
2   function use_g2 return number is
3   x number;
4   begin
5     select globals.g_3 into x
6     from dual
7     where globals.g_2 = globals.g_2;
8     return x;
9   end;
10  /
```

Function created.

The baseline benchmark test we will run calls each of the functions many times and records some execution time statistics. To do this, we will use the `DBMS_UTILITY.GET_TIME` function. This returns the number of centiseconds elapsed from some arbitrary epoch. By itself, this figure is useless. However, recording its value before and after a testing event allows us to observe the duration of that event to the nearest centisecond.

To test the access of a global variable, we will run the test a million times and record the elapsed time. To test the access of a global variable within a SQL

statement, we will run the test 50,000 times (to keep the total duration of the test down to an acceptable level).

```
SQL> declare
  2   d number;
  3   s number;
  4   begin
  5     s := dbms_utility.get_time;
  6     for i in 1 .. 1000000 loop
  7       d := use_g1;
  8     end loop;
  9     dbms_output.put_line((dbms_utility.get_time-s)/100||' seconds for usage');
 10    s := dbms_utility.get_time;
 11    for i in 1 .. 50000 loop
 12      d := use_g2;
 13    end loop;
 14    dbms_output.put_line((dbms_utility.get_time-s)/100||' seconds for usage in SQL');
 15  end;
 16  /
1.42 seconds for usage
5.61 seconds for usage in SQL
```

PL/SQL procedure successfully completed.

For the standard implementation, performance is very good. Let's add a new global variable.

```
SQL> create or replace
  2   package globals is
  3     g_1          number(5) := 1;
  4     g_2          number(5) := 1;
  5     g_3          number(5) := 1;
  6     g_new_global number(5) := 1;
  7   end;
  8   /
```

Package created.

As usual, we gauge the impact of this by querying the USER_OBJECTS view.

```
SQL> select object_name, status
  2   from user_objects
  3   where object_name in ('USE_G1', 'USE_G2');
```

OBJECT_NAME	STATUS
-----	-----
USE_G2	INVALID
USE_G1	INVALID

The functions that use the global variables have become invalid. What we would like to achieve is an alternative implementation that performs as well as the standard solution but is resilient to changes to the `globals` package.

Option 1: Use a Context

Contexts first appeared in version 8.1 and were part of the Virtual Private Database (VPD) feature.

TIP The VPD is covered in more detail in Chapter 11 of the Oracle *Application Developer Fundamentals Guide*

Contexts are used within VPD to provide values for a predetermined set of attributes that belong to a particular context definition. And of course, that is precisely what a set of global variables is, so we can explore the use of contexts to store global variables. Like a public package variable (or set of them), a context is bound to a particular session. (There are a number of extensions to the scope and usage of contexts within versions 9 and 10 of Oracle that we will not cover here.)

First, we create a context, `glob`, which is associated with the procedure, `set_values`, which we can then use to modify the attributes of the `glob` context.

NOTE You must have the `create any context` privilege to run this code.

```
SQL> create or replace
  2 context glob using pkg_security.set_values;
```

Context created.

Now we create the `set_values` procedure within package `pkg_security`. We simply use the `set_context` procedure in the built-in `dbms_session` package to define three attributes for our context: `g_1`, `g_2`, and `g_3`, with values of 1, 2, and 3 respectively. Thus the `glob` context holds the same name-value pairs as our previous `globals` package. Next, we create `pkg_security`.

```
SQL> create or replace
  2 package PKG_SECURITY is
  3 procedure set_values;
  4 end;
  5 /
```

Package created.

```
SQL> create or replace
  2 package body PKG_SECURITY is
  3 procedure set_values is
  4 begin
  5 dbms_session.set_context('glob','g_1',1);
  6 dbms_session.set_context('glob','g_2',2);
  7 dbms_session.set_context('glob','g_3',3);
  8 end;
  9 /
```

Package body created.

The `set_values` procedure must be executed before any reference to the `glob` context within will succeed. A logon trigger would be a prudent means of ensuring that the `globals` are available for any new database session.

We now create our standard two functions that use our context attributes. Again, the first one simply references an attribute value (via the `SYS_CONTEXT` function).

```
SQL> create or replace
  2 function use_g1 return number is
  3 begin
  4 return sys_context('glob','g_1');
  5 end;
  6 /
```

Function created.

The second function uses the value in a SQL statement.

```
SQL> create or replace
  2  function use_g2 return number is
  3    x number;
  4  begin
  5    select sys_context('glob','g_3') into x
  6    from dual
  7    where sys_context('glob','g_2') = sys_context('glob','g_2');
  8    return x;
  9  end;
10  /
```

Function created.

And now we will repeat the benchmark tests (after first setting our context values).

```
SQL> exec set_values
```

PL/SQL procedure successfully completed.

Our first attempt at testing this didn't finish within 15 minutes so we abandoned it and conducted a smaller test. We had to reduce the number of executions from 1,000,000 all the way down to 50,000.

```
SQL> declare
  2  d number;
  3  s number;
  4  begin
  5    s := dbms_utility.get_time;
  6    for i in 1 .. 50000 loop
  7      d := use_g1;
  8    end loop;
  9    dbms_output.put_line((dbms_utility.get_time-s)/100||' seconds for usage');
10    s := dbms_utility.get_time;
11    for i in 1 .. 50000 loop
12      d := use_g2;
13    end loop;
14    dbms_output.put_line((dbms_utility.get_time-s)/100||' seconds for usage in SQL');
15  end;
16  /
8.48 seconds for usage
7.1 seconds for usage in SQL
```

PL/SQL procedure successfully completed.

The performance is similar whether you use a global variable (and hence a context) in an SQL statement or the `globals` package. However, by simply referencing a global in a package, we suffer a large negative performance impact—the results suggest that this approach is more than 20 times slower.

A little investigation into the definition for the base PL/SQL package standard reveals the reason for the performance hit. In `$ORACLE_HOME/rdbms/admin/stdbody.sql`, the source reveals the following definition for `SYS_CONTEXT`:

```
function SYS_CONTEXT(namespace varchar2, attribute varchar2)
  return varchar2 is
  c varchar2(4000);
BEGIN
  select sys_context(namespace,attribute) into c from sys.dual;
  return c;
end;
```

It would appear that every call to `SYS_CONTEXT` from PL/SQL results in a recursive query to the database. This probably eliminates contexts as a source for (frequent use) global variables within PL/SQL because they are only performant when used from within SQL. And if SQL is the only place you are referencing globals, those globals should be stored in a database table!

On the positive side, using context provided insulation from dependency chain problems. Now we add a new name-value pair to hold a fourth global variable.

```
SQL> create or replace
 2  package body PKG_SECURITY is
 3  procedure set_values is
 4  begin
 5    dbms_session.set_context('glob','g_1',1);
 6    dbms_session.set_context('glob','g_2',2);
 7    dbms_session.set_context('glob','g_3',3);
 8    dbms_session.set_context('glob','g_new_global',3);
 9  end;
10  /
```

Package body created.

We can see that our functions have not been adversely affected:

```
SQL> select object_name, status
 2  from user_objects
```

```
3 where object_name in ('USE_G1','USE_G2');
```

OBJECT_NAME	STATUS
-----	-----
USE_G2	VALID
USE_G1	VALID

Option 2: Varchar2 Associative Arrays

Starting in version 9.2, Oracle allows associative arrays (formerly known as “index by” tables) to be indexed by a varchar2 datatype. Thus, a character string can be used to uniquely identify an entry within an array of values. This means that we can use the name of a global variable as the index into a PL/SQL table of global variable values. In its simplest form, we could create an associative array as a package specification variable. However, this would immediately create the dependency issues already covered, so we will create a function defined in the specification to return the appropriate value from the associative array, and hide the definition and population of array values within the package body. Our package specification thus simply contains the declaration of a function to return the global variable value.

```
SQL> create or replace
2 package new_globals is
3   function g(gname in varchar2) return number;
4 end;
5 /
```

Package created.

Our package body is where all the work takes place. We define a type, num_tab, which is indexed by a varchar2. This index will be the name of the global variable that we want to reference. Our function will return the value of the array entry indexed by the passed global variable name. We also take advantage of the initialization section of the package body so that all the global variable values are assigned only once—when the package is referenced for the first time.

```
SQL> create or replace
2 package body new_globals is
3
4   type num_tab is table of number
5     index by varchar2(30);
6
```

```

7   n num_tab;
8
9   function g(gname in varchar2) return number is
10  begin
11      return n(gname);
12  end;
13
14  begin
15      n('g_1') := 1;
16      n('g_2') := 1;
17      n('g_3') := 1;
18  end;
19  /

```

Package body created.

Now we rerun our benchmark. First we re-create our test functions to access the function within the `globals` package just created.

```

SQL> create or replace
2   function use_g1 return number is
3   begin
4       return new_globals.g('g_1');
5   end;
6   /

```

Function created.

```

SQL> create or replace
2   function use_g2 return number is
3   x number;
4   begin
5       select new_globals.g('g_3') into x
6       from dual
7       where new_globals.g('g_2') = new_globals.g('g_2');
8       return x;
9   end;
10  /

```

Function created.

Next we run our standard benchmark test to capture execution times. (We were able to use 1,000,000 executions when referencing the global variable.)

```

SQL> declare
  2   d number;
  3   s number;
  4   begin
  5     s := dbms_utility.get_time;
  6     for i in 1 .. 1000000 loop
  7       d := use_g1;
  8     end loop;
  9     dbms_output.put_line((dbms_utility.get_time-s)/100||' seconds for usage');
 10    s := dbms_utility.get_time;
 11    for i in 1 .. 50000 loop
 12      d := use_g2;
 13    end loop;
 14    dbms_output.put_line((dbms_utility.get_time-s)/100||' seconds for usage in SQL');
 15  end;
 16  /
4.11 seconds for usage
11.36 seconds for usage in SQL

```

PL/SQL procedure successfully completed.

A promising result! However, is our solution also resilient from dependency issues? Let's add another global variable to the initialization section in the package body.

```

SQL> create or replace
  2   package body new_globals is
  3
  4     type num_tab is table of number
  5       index by varchar2(30);
  6
  7     n num_tab;
  8
  9     function g(gname in varchar2) return number is
 10     begin
 11       return n(gname);
 12     end;
 13
 14   begin
 15     n('g_1') := 1;
 16     n('g_2') := 1;
 17     n('g_3') := 1;
 18     n('g_new_global') := 1;

```

```
19 end;
20 /
```

Package body created.

Next, look at the status of our testing functions.

```
SQL> select object_name, status
       2 from user_Objects
       3 where object_name in ('USE_G1','USE_G2');
```

OBJECT_NAME	STATUS
USE_G2	VALID
USE_G1	VALID

It appears that our dependency issues are covered as well. Is the associative array the answer to all our concerns? Possibly, but there are some drawbacks to the approach. Notice that in our simple example, all the global variables returned the same datatype (numeric). This is hardly likely to be the case in a more realistic scenario within a large application. There will be strings, dates, and possibly even raw data that may be referenced as global constants. This means that handling globals would require multiple associative arrays, one for each different datatype that the global values represent. Because packaged functions cannot be overloaded when they only differ by their return datatype, you would require a different function call for each datatype. For example, to handle numeric, character, and date datatypes, the global package may look like the following:

```
package extended_globals is
  function get_numeric(gname in varchar2) return number;
  function get_date(gname in varchar2) return date;
  function get_varchar2(gname in varchar2) return varchar2;
end;
```

Thus, any calling function must explicitly know the datatype of the global variable it is seeking. This might not be such a bad thing, but it introduces additional complexity into the implementation.

To summarize, there is no perfect solution to handling of globals within your application. It's a balancing act between your requirements—reducing the dependency issues versus maintaining adequate control over the ongoing additions and changes to those global variables. Ultimately as always, you should benchmark your alternatives carefully.

Enabling Recursion

Traversal of trees, management of linked list structures, backtracking algorithms, and many similar classical computing problems are often more easily coded with recursive techniques than a sequential coding approach. Although a full discussion on recursion is beyond the scope of this chapter, it is worth noting (within the context of packages) that using standalone procedures prohibit you from using any mutually recursive routines. They simply will not compile.

```
SQL> create or replace
  2 procedure A(p number) is
  3 begin
  4   if p < 5 then
  5     B(p+1);
  6   end if;
  7 end;
  8 /
```

Warning: Procedure created with compilation errors.

```
SQL> create or replace
  2 procedure B(p number) is
  3 begin
  4   if p < 5 then
  5     A(p+1);
  6   end if;
  7 end;
  8 /
```

Warning: Procedure created with compilation errors.

```
SQL> alter procedure B compile;
alter procedure B compile
*
ERROR at line 1:
ORA-04020: deadlock detected while trying to lock object MCDONAC.B
```

```
SQL> alter procedure A compile;
alter procedure A compile
*
ERROR at line 1:
ORA-04020: deadlock detected while trying to lock object MCDONAC.A
```

However, using packages allows forward references, thus opening the possibilities for recursion. If we embed the two procedures just shown within a package, mutual recursion becomes possible.

```
SQL> create package RECURSION is
  2   procedure A(p number);
  3   procedure B(p number);
  4 end;
  5 /
```

Package created.

```
SQL> create or replace
  2   package body RECURSION is
  3
  4   procedure A(p number) is
  5   begin
  6     if p < 5 then
  7       B(p+1);
  8     end if;
  9   end;
 10
 11  procedure B(p number) is
 12  begin
 13    if p < 5 then
 14      A(p+1);
 15    end if;
 16  end;
 17
 18  end;
 19 /
```

Package body created.

Why Have People Avoided Packages?

So far, everything we've said about packages is positive. You have all the facilities of standalone procedures plus the additional benefits of code encapsulation, insulation from change, greater control over the scope of variables, and initialization. There are no negatives with using packages. Is it just purely ignorance that has kept so many developers away from packages?

In early versions of Oracle, packages received some bad press for a variety of reasons, most of which were attributable to their usage and the way that this usage wreaked havoc with shared pool memory management on the server. Let's explore a couple of the poor usage habits that have historically dogged packages.

Ignorance of the Benefits of Separation

Most of the benefits of packages discussed in this chapter are a direct consequence of the separation between package specification and package body. These benefits are lost if the two are treated as one. Common mistakes in this regard are as follows:

- Recompiling both specification and body for any change that is made. If you have not changed the package specification, do *not* reload it into the database every time the package body is changed. Doing so eliminates the insulation from dependency.
- Storing the package specification and package body in a single text file. Look in the `$ORACLE_HOME/rdbms/admin` directory and you'll see that all the supplied Oracle packages are delivered in separate files—one file for the specification, one for the body.

In a perfect world, package specifications would never change. That has long been one of the primary goals for any language that supports encapsulation of the functionality offered to the calling environment.

It's a Package, Not a Library

Many developers were first exposed to packages (or PL/SQL, for that matter) through Oracle's client-server tools (Forms, Reports, and Graphics), all of which used a local PL/SQL engine as the runtime environment. The dependency tree benefits of packages were not particularly well documented and in any event, early releases of Forms required that every PL/SQL unit in a module be recompiled (even if it was already valid) before an executable `fm` file was produced. Therefore, it didn't really matter how many dependencies a PL/SQL unit had—they all had to be compiled each time anyway.

Therefore, many developers treated packages as a convenience more than anything else, merely as a means for grouping logically related PL/SQL program units. This didn't cause major problems on the client software development environments, but it was an approach that was carried over into the server-based

PL/SQL platform. As such, it was common to see enormous packages with generic names such as `common`, `tools` or `utils`, each serving as a catch-all for hundreds of small routines.

The problem with this approach was that in early releases of Oracle 7, packages (or any PL/SQL program unit) were loaded entirely into memory when any reference was made to one of their components, and that memory had to be a single contiguous chunk.

Because the shared pool is typically made up of a large number of small pieces of memory (see the following sidebar), loading a large package into a contiguous memory space could easily put a system under enormous strain. Hundreds or thousands of objects would need to be flushed out of the shared pool in least-recently used order to make space for the package. Imagine a large bucket filled to the brim with tennis balls numbered from 1 to 1,000, and you must find space for an incoming basketball by removing tennis balls in numerical order—that should give you an idea of the amount of work required in shared pool memory management to load a single package.

Compounding the problem, there was no guarantee of success in this operation because you could conceivably flush out all chunks of memory that were freeable and still not have a big enough chunk of free space because you may not have been flushing out *contiguous* chunks of memory. Systems could easily be brought to a standstill while this operation took place.

As a consequence, many developers and DBAs began to equate packages with the ubiquitous error message “ORA-4031:unable to allocate bytes of shared memory.”

More About the Shared Pool

Although it is outside the scope of this book to go into detail about the makeup of the shared pool, the following query (which you must run while connected as SYSDBA) will demonstrate that for normal use within a typical database, the shared pool is comprised of a large number of small chunks of memory.

```
SQL> select bytes chunk_size, count(*) no_of_chunks from
2  (
3  select power(10,trunc(ln(ksmchsiz)/ln(10)))||' to '||
4  (power(10,trunc(ln(ksmchsiz)/ln(10))+1)-1) chunk_size
5  from x$kmsmp
6  where ksmchcls != 'perm' )
7  group by bytes
8  /
```

CHUNK_SIZE	NO_OF_CHUNKS
-----	-----
10 to 99	2487
100 to 999	16548
1000 to 9999	4573
10000 to 99999	15
100000 to 999999	18
1000000 to 9999999	7

This query shows the number of free and in-use memory chunks in the shared pool within a logarithmic distribution.

Since version 7.3 of Oracle, the problems associated with loading PL/SQL units into the shared pool have largely been resolved. The management of PL/SQL program units can be likened to demand paging; that is, the execution code for a PL/SQL program unit is loaded on an on-demand basis and in chunk sizes no larger than 4k.

This can be demonstrated by creating a large package and seeing whether the entire package code is loaded if just a small routine from that package is executed. We will create a package where the source code is comprised of 300 small procedures.

```
package MEMTEST is
  procedure X1;
  procedure X2;
  ...
  procedure X300;
end;
```

Each procedure assigns the number 1 to a numeric variable, y. Rather than type the source by hand, we will use some PL/SQL to dynamically create the large package and its body.

```
SQL> declare
  2   x varchar2(32767);
  3   begin
  4     for i in 1 .. 300 loop
  5       x := x || ' procedure X' || i || ';';
  6     end loop;
  7     execute immediate
  8       'create or replace package MEMTEST is ' || x || ' end;';
  9     x := replace(x, ',', ' is y number; begin y := 1; end;');
```

```

10  execute immediate
11    'create or replace package body MEMTEST is '||x||' end;';
12  end;
13  /

```

PL/SQL procedure successfully completed.

Now we'll flush the shared pool to clear out any existing evidence of the memtest package, execute just one of the procedures in the package, and recheck the contents on the shared pool.

```
SQL> alter system flush shared_pool;
```

System altered.

```
SQL> exec memtest.x1;
```

PL/SQL procedure successfully completed.

Next we look at the V\$SGASTAT view to determine the amount of memory in use for PL/SQL code (the procedure we just executed).

```
SQL> select * from v$sgastat
  2 where name like 'PL/SQL MPCODE';

```

POOL	NAME	BYTES
shared pool	PL/SQL MPCODE	80980

Approximately 80K are in use. That may seem like a lot, but remember that to run any PL/SQL program, components of the standard package (which defines PL/SQL itself) must also be loaded. If our memtest package code was loaded entirely into memory, presumably any references to other procedures in package memtest should already be present. However, as we execute different procedures, we can see the amount of code loaded into shared pool memory increases with each execution. Next we execute procedure X2 within the memtest package.

```
SQL> exec memtest.x2;
```

PL/SQL procedure successfully completed.

```
SQL> select * from v$sgastat
  2 where name like 'PL/SQL MPCODE';

```

POOL	NAME	BYTES
shared pool	PL/SQL MPCODE	85612

```
SQL> exec memtest.x10;
```

PL/SQL procedure successfully completed.

```
SQL> select * from v$sgastat
  2  where name like 'PL/SQL MPCODE';
```

POOL	NAME	BYTES
shared pool	PL/SQL MPCODE	90652

```
SQL> exec memtest.x100;
```

PL/SQL procedure successfully completed.

```
SQL> select * from v$sgastat
  2  where name like 'PL/SQL MPCODE';
```

POOL	NAME	BYTES
shared pool	PL/SQL MPCODE	95284

We stress that just because large packages do not place the system under the memory and latching strain that they have in the past, it does not imply that we should create systems comprised of just a few massive packages. Each version of Oracle tends to raise the limits on the maximum allowed size of a PL/SQL unit, but we should not forget the other maxims of maintainability, modularity, and perhaps most importantly, common sense, when composing our packages.

When Not to Use Packages

There is one scenario for which a package is not the preferred solution for a PL/SQL stored program: any PL/SQL function that will be a candidate for a function-based index. In these cases, the protection from the dependency chain that packages afford can cause problems. Let's demonstrate the issue with an example. First, we will create and populate a table on which we will create a function-based index.

```
SQL> create table x ( x number, y varchar2(30));
```

Table created.

```
SQL> insert into x
  2 select rownum, 'xxxxx' || rownum
  3 from SRC
  4 where rownum < 10000;
```

9999 rows created.

We will be indexing the table, x, with a function-based index where that function is defined within a package. The next step is to create a package to hold such a function. For a function to be used within a function-based index, it must be defined as deterministic. Doing this will tell Oracle that this function will always return a consistent (in other words, unchanging) result for multiple executions with a specific parameter value.

```
SQL> create or replace
  2 package FBI is
  3   function ix(p varchar2) return varchar2 deterministic;
  4 end;
  5 /
```

Package created.

Our function will be very simple—it will return whatever value is passed to it.

```
SQL> create or replace
  2 package body FBI is
  3   function ix(p varchar2) return varchar2
  4     deterministic is
  5     begin
  6       return p;
  7     end;
  8 end;
  9 /
```

Package body created.

Now we can use this package function to build an index on our table, x. For the optimizer to use this (or any) function-based index, we also need to collect optimizer statistics for the table and index once created.

```
SQL> create index x1 on x ( FBI.ix(y));
```

Index created.

```
SQL> analyze table x estimate statistics
```

Table analyzed.

To prove that the index will be used, we can enable the SQL*Plus autotrace facility to display execution plans for any SQL that we run.

```
SQL> set autotrace on
```

```
SQL> select *
  2  from x
  3  where FBI.ix(y) = 'xxxxx123';
```

```
X          Y
-----  -----
123      xxxxx123
```

Execution Plan

```
-----
 0      SELECT STATEMENT Optimizer=CHOOSE (Cost=2 Card=1 Bytes=12)
 1      0      TABLE ACCESS (BY INDEX ROWID) OF 'X' (Cost=2 Card=1 Bytes=12)
 2      1      INDEX (RANGE SCAN) OF 'X1' (NON-UNIQUE) (Cost=1 Card=1)
```

No problems so far. The function-based index is being used as anticipated. Obviously, if we were to change the package function, we would need to rebuild our index to pick up the new values being returned from the function. Nevertheless, one of the selling points of packages is that to change the underlying implementation of one of its components, only the *package body* needs to be changed. We'll alter just the package body to append the value abc to the return value.

```
SQL> create or replace
  2  package body p1 is
  3    function ix(p varchar2) return varchar2
  4      deterministic is
  5    begin
  6      return p||'abc';
  7    end;
  8  end;
  9  /
```

Package body created.

Oracle is *not* aware that the function code has been altered because the dependency chain is based on the package specification, not the body. As a result, the values stored in the index are effectively “out of sync” with the function upon which it is based. This could lead to the nasty situation of getting different results from a query dependent on how the optimizer decides to run the query. If we re-run the SQL from above, and we allow the optimizer to use the index as per the previous execution plan, we obtain a row in the result.

```
SQL> select *
  2  from x
  3  where p1.ix(y) = 'xxxxx123';
```

X	Y
-----	-----
123	xxxxx123

The fact that we have obtained a row reveals the problem here—there is no row in the table for which column Y has a trailing abc, so no rows should be returned. If the same query is re-executed but we force the optimizer to not use the index, then we get the correct result, namely, no rows being returned:

```
SQL> select /*+ FULL(x) */ *
  2  from x
  3  where p1.ix(y) = 'xxxxx123';
```

no rows selected

Until the index is rebuilt, it will yield incorrect results. Of course, if the package *specification* had also been re-created, the function-based index would have been marked as unusable (that is, the `funcidx_status` column in `user_indexes` would be set to `disabled`), and the index would need to be rebuilt. But in an application development environment where package bodies *only* are being altered and recompiled (to ensure dependency problems are avoided), it is risky to assume that someone will take the time to check whether any functions in that package are used within indexes. For this reason, PL/SQL functions used in indexes are best created as standalone routines.

Delivered Packages

Another positive aspect that will result from embracing packages throughout your applications is that the development team may start to focus on the func-

tionality that is delivered *for free* with the PL/SQL packages that are supplied with the Oracle software.

Take a moment to run a query to see the PL/SQL packages that are delivered under the SYS account and you will get some startling results. The following query was performed on a version 9.2 database:

```
SQL> select object_name
       2  from dba_objects
       3  where owner = 'SYS'
       4  and object_type = 'PACKAGE';
```

```
OBJECT_NAME
-----
CONNECTIONINTERFACE
CURSORMANAGERINTERFACE
DATABASEINTERFACE
DATAPROVIDERINTERFACE
DATATYPEIDCONSTANTS
DBMSOBJG
DBMSOBJG2
DBMSOBJGWRAPPER
DBMSOBJG_DP
DBMSZEXP_SYSPKGGRNT
DBMS_ALERT
DBMS_APPCTX
DBMS_APPLICATION_INFO
...
XSLPROCESSOR
XSLPROCESSORCOVER
XSLSTYLESHEETCOVER

357 rows selected.
```

Over 300 packages! Even though not all of them are intended for developers, the ones that are for developers are very well documented in the standard Oracle documentation. One of the manuals we always head to first when a new version of Oracle is released is the *Supplied PL/SQL Packages and Types Reference*. Within any company, certain functionality often pops out as being common across several Oracle projects. When that functionality becomes common across many companies, Oracle will “jump on the bandwagon” and deliver it as a native feature in the database. Each new release of Oracle delivers more PL/SQL built-in functions that you do not (and should not) have to build for yourself.

A definitive guide to all the packages delivered by Oracle is easily a book in its own right. In this book, we simply use these built-in packages as and where appropriate. For example, in the *Security Packages* chapter, we use the built-in `dbms_fga` package for fine-grained auditing. In the *Web Packages* chapter, we use `utl_http` to make HTTP requests directly from an Oracle database, and in the *Debugging* chapter, we investigate various built-in packages for effective debugging (alongside our own custom solution).

In the next section, we will look at a few delivered package solutions because they highlight two very important points:

- They are solutions to common requirements. We have seen developers spend hours (or days or weeks) building solutions because they didn't know that Oracle supplied a package that was perfect for the job.
- If you take the time to explore the supplied packages, you can often discover effective alternative uses for them that Oracle might not have originally intended.

Code Path Tracing Made Easy

Being able to debug your application code is vital to the success of any project. For many 3GL languages, the developer is responsible for tracking where the code is currently executing. The most typical solution to this requirement is the implementation of a stack. For example, a solution of this type in PL/SQL could look as follows:

```
SQL> create or replace
2 package stack is
```

We define an associative array to represent a stack of modules.

```
3 type module_list is
4 table of varchar2(80)
5 index by binary_integer;
6
```

We then define the standard operations that we perform on a stack—pushing an item onto the stack and popping an item off the top of the stack.

```
7 procedure push(module_name varchar2);
8 procedure pop;
9 procedure show_stack;
```

```
10
11 end;
12 /
```

Package created.

The underlying stack operations are trivial to implement with associative arrays because the array itself maintains a count of the entries within it.

```
SQL> create or replace
  2 package body stack is
  3
  4   m module_list;
  5
  6 procedure push(module_name varchar2) is
  7 begin
  8   m(m.count+1) := module_name;
  9 end;
10
11 procedure pop is
12 begin
13   m.delete(m.count);
14 end;
15
16 procedure show_stack is
17 begin
18   for i in 1 .. m.count loop
19     dbms_output.put_line(rpad('-',i,'-')||m(i));
20   end loop;
21 end;
22
23 end;
24 /
```

Package body created.

Now that we have a generic stack facility within PL/SQL, we can use it within our applications programs to manage tracking our execution paths. As we enter a new procedure, we add its name to the stack, and when we exit the procedure, we remove its name from the stack. To demonstrate, the following three procedures use the stack to record the execution of each procedure:

```
SQL> create or replace
  2 procedure p1 is
```

```

3  begin
4    stack.push('P1');
5    stack.show_stack;
6    stack.pop;
7  end;
8  /

```

Procedure created.

```

SQL> create or replace
2  procedure p2 is
3  begin
4    stack.push('P2');
5    p1;
6    stack.pop;
7  end;
8  /

```

Procedure created.

```

SQL> create or replace
2  procedure p3 is
3  begin
4    stack.push('P3');
5    p2;
6    stack.pop;
7  end;
8  /

```

Procedure created.

Now, when we execute procedure P3, the call to `show_stack` within procedure P1 outputs the stack of procedure calls.

```

SQL> exec p3;
-P3
--P2
---P1

```

PL/SQL procedure successfully completed.

There are two major problems with this. First, every PL/SQL program that we want to participate in the stack tracing must be coded so that it pushes its name onto the stack at its entry point, and also removes its name from the stack

for every possible exit point in the procedure. Every possible error must be accounted for, and all it takes is a single PL/SQL module to miss a call to the stack management for the stack to be corrupted. Second, and this is a more fundamental point, the functionality is already provided by Oracle! We can recode procedure P1 as the following:

```
SQL> create or replace
  2 procedure p1 is
  3 begin
  4   dbms_output.put_line(
  5     substr(dbms_utility.format_call_stack,1,255));
  6 end;
  7 /
```

Procedure created.

The `FORMAT_CALL_STACK` function within the delivered `dbms_utility` package returns the call stack as a character string. No special stack package is required, and there is no requirement to have special code within every single PL/SQL program developed. Executing the revised P3 procedure demonstrates the output from `DBMS_UTILITY.FORMAT_CALL_STACK`.

```
SQL> exec p3;
----- PL/SQL Call Stack -----
  object          line  object
  handle          number name
700000003f56250      3  procedure SCOTT.P1
700000003f4c3c0      4  procedure SCOTT.P2
700000003f47e38      4  procedure SCOTT.P3
700000003f44e98      1  anonymous block
```

We also get the added benefits of the actual line number at which each PL/SQL module called another. Admittedly, a little string manipulation would be required to parse the output string, but this is small price to pay for such a convenient facility.

And while you are developing debugging utilities that take advantage of the `DBMS_UTILITY.FORMAT_CALL_STACK` function, you can get more benefits from other built-in packages, such as the `dbms_application_info` package, which we'll discuss in detail in the *Debugging* chapter.

Other Useful Routines

There are a number of other useful tools within the `dbms_utility` package that allow you to obtain information about your system architecture. We have already seen how the `DBMS_UTILITY.GET_TIME` function can return the number of centiseconds from arbitrary epoch. Let's take a quick look at a few others.

Dependency Information

This procedure (which does not appear in the documentation) provides a neat listing of all the dependency information associated with a database object.

```
SQL> exec dbms_utility.get_dependency('TABLE',user,'EMP');
-
DEPENDENCIES ON SCOTT.EMP
-----
*TABLE SCOTT.EMP()
*  PACKAGE BODY SCOTT.PKG()
-
DEPENDENCIES ON SCOTT.EMP
-----
*TABLE SCOTT.EMP()
*  PACKAGE BODY SCOTT.PKG()

PL/SQL procedure successfully completed.
```

Platform and Version Information

These programs return the version and operating system port for the database. For example, on a laptop running version 9204 of the database, the following tests were performed:

```
SQL> variable b1 varchar2(64)
SQL> variable b2 varchar2(64)
SQL> exec dbms_utility.db_version(version=>b1,compatibility=>b2);

PL/SQL procedure successfully completed.

SQL> print b1
```

B1

```
-----
9.2.0.4.0
```

```
SQL> print b2
```

B2

```
-----
9.2.0.0.0
```

```
SQL> select dbms_utility.port_string from dual;
```

```
PORT_STRING
```

```
-----
IBMPC/WIN_NT-8.1.0
```

From these results, it would appear that the port string is useful to determine on the platform, but certainly version information should not be trusted from the resultant string.

Retrieving DDL

Suppose you have a table (or any database object) in the database for which you would like to see the underlying DDL. This may sound like a simple enough task but unfortunately, given the increasing flexibility and functionality that arrives with each release of Oracle, it is actually more difficult than you might imagine.

Many developers or administrators have tackled this problem by using scripts to probe the data dictionary. In fact, a quick search on Google reveals a myriad of such solutions, one of which is reproduced as follows. This sample is designed to regenerate the create table statement used for each of the tables within the current schema.

```
SELECT DECODE(T1.COLUMN_ID,1,'CREATE TABLE ' ||
T1.TABLE_NAME || ' (' || ' ') A,
T1.COLUMN_NAME B, T1.DATA_TYPE ||
DECODE(T1.DATA_TYPE, 'VARCHAR2', '(' ||
TO_CHAR(T1.DATA_LENGTH)||')', 'NUMBER','(' ||
TO_CHAR(T1.DATA_PRECISION)||
', ' || TO_CHAR(T1.DATA_SCALE)||')',
'CHAR','(' || TO_CHAR(T1.DATA_LENGTH)||')') ||
DECODE(T1.COLUMN_ID,MAX(T2.COLUMN_ID), ');',' ','') C
FROM USER_TAB_COLUMNS T1, USER_TAB_COLUMNS T2
```

```

WHERE T1.TABLE_NAME = T2.TABLE_NAME
GROUP BY T1.COLUMN_ID, T1.TABLE_NAME, T1.DATA_TYPE,
         T1.DATA_LENGTH, T1.DATA_SCALE, T1.COLUMN_NAME, T1.DATA_PRECISION
ORDER BY T1.COLUMN_ID;

```

Such a script does generate some meaningful output, but the resultant DDL misses a number of important elements such as the storage clause, the constraints, the object datatypes, the handling of index-organized, clustered, or partitioned tables, and many other elements that may be used for a table definition. It's a good starting point, but certainly not a definitive or complete solution. Maybe the best bet is therefore to head over to an Oracle site and get an official version of such a script. The following script was found on the Metalink support site.

```

DECLARE
  cursor cur0 is
    select table_name, TABLESPACE_NAME, PCT_FREE, PCT_USED,
           INI_TRANS, MAX_TRANS,
           INITIAL_EXTENT, NEXT_EXTENT, PCT_INCREASE
    from   user_tables
    order by table_name;

  cursor cur1 (t_name varchar2) is
    select table_name, column_name, data_type, data_length,
           data_precision, data_scale, nullable
    from   user_tab_columns
    where  table_name = t_name
    order by column_name;

  tab_name      varchar2(40);
  tabsp_name    varchar2(40);
  mpct_free     number;
  mpct_used     number;
  mini_trans    number;
  mmax_trans    number;
  mini_ext      number;
  mnext_ext     number;
  mpct_inc      number;
  col_name      varchar2(40);
  ct            number := 0;
  line_ct       number := 0;

BEGIN
  delete from AAA;

```

```

for cur0_rec in cur0 loop
    tab_name := cur0_rec.table_name;
    tabsp_name := cur0_rec.TABLESPACE_NAME;
    mpct_free := cur0_rec.PCT_FREE ;
    mpct_used := cur0_rec.PCT_USED;
    mini_trans := cur0_rec.INI_TRANS;
    mmax_trans := cur0_rec.MAX_TRANS;
    mini_ext := cur0_rec.INITIAL_EXTENT ;
    mnext_ext := cur0_rec.NEXT_EXTENT ;
    mpct_inc := cur0_rec.PCT_INCREASE ;
    insert into AAA(f1, line)
    values(line_ct, 'create table ' || tab_name || '(' );
    line_ct := line_ct + 1;

ct := 0;
for cur1_rec in cur1(tab_name) loop
    ct := ct + 1;
    if ct = 1 then
        insert into AAA(f1, line )
        values (line_ct ,
            ' ' || cur1_rec.column_name || ' ' ||
            cur1_rec.data_type ||
            decode(cur1_rec.data_type, 'VARCHAR2', '(' ||
            to_char(cur1_rec.data_length) || ')',
            'NUMBER', decode(cur1_rec.data_precision, null, '',
            '(' || to_char(cur1_rec.data_precision) ||
            decode(cur1_rec.data_scale, null, ')', ', ' ||
            to_char(cur1_rec.data_scale) || ') ' ) ), '' ) ||
            decode(cur1_rec.nullable, 'Y', '', ' NOT NULL') );
        line_ct := line_ct + 1;
    else
        insert into AAA(f1, line)
        values (line_ct, ' , ' || cur1_rec.column_name || ' ' ||
            cur1_rec.data_type ||
            decode(cur1_rec.data_type, 'VARCHAR2', '(' ||
            to_char(cur1_rec.data_length) || ')',
            'NUMBER', decode(cur1_rec.data_precision, null, '',
            '(' || to_char(cur1_rec.data_precision) ||
            decode(cur1_rec.data_scale, null, ')', ', ' ||
            to_char(cur1_rec.data_scale) || ') ' ) ), '' ) ||
            decode(cur1_rec.nullable, 'Y', '', ' NOT NULL') );
        line_ct := line_ct + 1;
    end if;

```



```

end loop;
insert into AAA (f1, line )
values(line_ct, ' ');
line_ct := line_ct + 1;
insert into AAA (f1, line )
values (line_ct, ' PCTFREE ' || mpct_free || ' PCTUSED ' || mpct_used );
    line_ct := line_ct + 1;
insert into AAA (f1, line )
values (line_ct, ' INITRANS ' || mini_trans || ' MAXTRANS ' || mmax_trans);
insert into AAA (f1, line )
values (line_ct, ' TABLESPACE ' || tabsp_name );
line_ct := line_ct + 1;
insert into AAA (f1, line )
values(line_ct, ' STORAGE ( INITIAL ' || mini_ext || ' NEXT '
        || mnext_ext || ' PCTINCREASE ' || mpct_inc || ');' );
    line_ct := line_ct + 1;
insert into AAA (f1, line )
values(line_ct, 'REM -----NEXT TABLE ----- ');
line_ct := line_ct + 1;
commit;
end loop;
END;
/

```

Although this script looks impressive and yields more of the finer details, it still doesn't cover everything. Storage clauses are taken care of, but the moment a new feature arrives, the script will need to be corrected. For example, the script has no support for generating the DDL for a partitioned table.

Once again, taking the time to look for a predelivered package yields the best solution. A simple call to the GET_DDL function within the dbms_metadata package returns the DDL as a character large object (CLOB). For example, to obtain the DDL for the emp table within the SCOTT schema, we need only issue a simple select statement, shown as follows.

```

SQL> select dbms_metadata.get_ddl('TABLE','EMP','SCOTT')
2 from dual;

CREATE TABLE "SCOTT"."EMP"
(
    "EMPNO" NUMBER(4,0),

    "ENAME" VARCHAR2(10),
    ...
    CONSTRAINT "PK_EMP" PRIMARY KEY ("EMPNO")

```

```

USING INDEX PCTFREE 10 INITRANS 2 MAXTRANS 255
STORAGE(...) TABLESPACE "USERS" ENABLE,
    CONSTRAINT "FK_DEPTNO" FOREIGN KEY ("DEPTNO")
        REFERENCES "SCOTT"."DEPT" ("DEPTNO") ENABLE
) PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255
NOCOMPRESS LOGGING STORAGE(...) TABLESPACE "USERS"

```

It is that simple. There are also a number of transformations that you can make to the DDL that is returned. For example, if you want to omit the storage parameters, you can simply set the appropriate transformation variable as follows:

```

begin
  DBMS_METADATA.SET_TRANSFORM_PARAM(
    DBMS_METADATA.SESSION_TRANSFORM,
    'STORAGE',
    false);
end;

```

The DDL is not limited to tables. You can get the DDL for just about anything in the database. For example, the user definitions are easily retrieved with the following:

```

SQL> select dbms_metadata.get_ddl('USER', 'SCOTT')
       2 from dual;

```

```

CREATE USER "SCOTT"
IDENTIFIED BY VALUES 'F894844C34402B67'
DEFAULT TABLESPACE "USERS"
TEMPORARY TABLESPACE "TEMP"

```

See the *PL/SQL Supplied Packages Guide* for a complete list of the different objects for which you can retrieve the defining DDL. It's far easier than writing a suite of scripts to do the task.

An Interesting Use for DBMS_ROWID

Most developers are aware of the `dbms_rowid` package, which consists of a number of utilities to manipulate rowids, but many may think it is no big deal. After all, rowid manipulation is something that is rarely if ever required in the application development life cycle. However, something that *is* part of application development is resolving locking (or *enqueue*) problems, and the `dbms_rowid` package can be very useful in assisting here. To demonstrate, let's consider two separate

sessions, each attempting to remove rows from the emp table. The first session adds 10 percent to the salaries of all our long-term employees (anyone who was hired before September 19, 1981, in this case).

```
SQL> update emp
  2 set sal = sal * 1.1
  3 where hiredate < to_date('08/SEP/1981','DD/MON/YYYY');
```

6 rows updated.

Our second session closes department 20, and all employees in this department are relocated to department 30.

```
SQL> update emp set deptno = 30 where deptno = 20;
(waiting...)
```

We have a locking problem but because we are updating multiple rows, it is not immediately apparent which row(s) are causing the problem. This is where `dbms_rowid` becomes very useful. First, we examine `V$SESSION` view, which identifies the explicit row that is causing the problem. The row prefixed columns are populated when a session is waiting for an enqueue, so we can check for those sessions that are waiting on an enqueue on the emp table.

```
SQL> select row_wait_obj#, row_wait_file#,
  2 row_wait_block#, row_wait_row#
  3 from v$session
  4 where row_wait_obj# in (
  5 select data_object_id
  6 from user_objects
  7 where object_name = 'EMP');
```

ROW_WAIT_OBJ#	ROW_WAIT_FILE#	ROW_WAIT_BLOCK#	ROW_WAIT_ROW#
49627	7	138	0

Armed with this information, we can now use the `dbms_rowid` package to isolate precisely the row that is causing the problem.

```
SQL> select *
  2 from emp
  3 where rowid = dbms_rowid.rowid_create(1,49627,7,138,0);
```

EMPNO	ENAME	MGR	HIREDATE	SAL	DEPTNO
7369	SMITH	7902	17/DEC/80	880	20

Of course, it would take very little effort for a DBA to set this kind of activity in a simple script so that developers could always easily locate rows involved in an enqueue problem. This is much better than developers having to take the more drastic action of terminating their database sessions until the problem disappears.

Background Tasks

Consider the case in which a database task (for example, report generation) may take 30 seconds to complete. We don't want to lock up the requesting user's terminal/PC while that request is executed; we would like the user to be able to request the report and then continue on with other tasks. A commonly implemented (but generally flawed) approach to solving this problem is to use database pipes. Because of the length of the code, we won't include the entire solution here, but we will summarize the pseudo-code for such a solution.

Foreground Process:

1. Pack a pipe message with the task to be run in the background.
2. Send the pipe message in pipe P1.
3. Wait for a return message on a different pipe P2 to ensure the sent message was received.
4. Return control to the user.

Background Processor:

1. Wait for a message on pipe P1.
2. When a message is received, remove it from pipe P1.
3. Send acknowledgment on pipe P2.
4. Run commands as specified in the message (or have some sort of indication of what was done).
5. Resume waiting on pipe P1.

A number of complexities in this solution that are not immediately apparent from the algorithm just shown are:

- You need to ensure that the background processor is running.
- You need to ensure that foreground processes don't wait forever in case the background processor failed or is busy on another task.
- Pipes are nontransactional, so if the foreground process requests a background task as part of a transaction, the task will execute even if the foreground process later issues a rollback.

If the last item above is a “back breaker,” the resolution is not trivial. You might consider using transactional database alerts, but task requests can be dropped because the implementation of database alerts only guarantees that one (not all) message is received by a background processor that has registered interest.

Of course a better solution is to use a database table that stores the requests, and have the background processor read from this table. Even this approach requires some careful forethought. For example, if we want to increase the number of concurrent background processors, we will need to build in some locking intelligence to ensure that no task is executed either more than once or not at all.

We are amazed at the number of sites where such a solution exists, painstakingly built and tested from scratch by a development team when once again, *the functionality is already delivered for free*. The Job Queue facility has been available since version 7 but it is under utilized because it is mistakenly considered to be purely for tasks that run on a fixed schedule. Using a job queue to perform a background task is as simple as setting the following foreground task:

```
dbms_job.submit(:job, 'task_to_be_performed;');
```

That's all there is to it! The task request is automatically transactional—if you issue a rollback, the job will not be run. You can have up to 36 background processors in version 8, and 1,000 in version 9. You can add, subtract, stop, and start the background processors at any time without affecting a foreground process's ability to submit tasks. Any errors are logged in trace files, and you get some restartability features if a job fails.

Launching processes asynchronously with transactional processes is a fantastic means of adding functionality to your database. For example, you could use job queues to delete a file only when a transaction is actually committed. First we create a table, `list_of_files`, which contains the filename and directory where that file is located.

```
SQL> create table LIST_OF_FILES (
2   fname varchar2(30),
3   fdir  varchar2(30)) ;
```

Table created.

```
SQL> insert into LIST_OF_FILES
  2   values ('demo.dat','/tmp');
```

1 row created.

We can now create a trigger on that table that will submit a job to run the `utl_file.fremove` procedure (which deletes files) when a user deletes a row. Because jobs are transactional, the deletion of the file will take place only if the user commits the action.

```
SQL> create or replace
  2   trigger DEL_FILE
  3   before delete on LIST_OF_FILES
  4   for each row
  5   declare
  6     j number;
  7   begin
  8     dbms_job.submit(j,'utl_file.fremove(''||
  9                   :old.fdir||'', ''||:old.fname||'');');
 10   end;
 11   /
```

Trigger created.

Let's see this in action. We inserted a row previously for the file `/tmp/demo.dat`. We can see that this file exists on our system.

```
SQL> !ls /tmp/demo.dat
/tmp/demo.dat
```

We now delete this row from the table.

```
SQL> delete from list_of_files;
```

1 row deleted.

If we had deleted the file immediately from within the trigger, we would have been stuck if the user issued a rollback. The row would be reinstated to the table, but the file would be gone. But because the deletion was wrapped as a database job, it does not start until we commit. We can see that our file still exists.

```
SQL> !ls /tmp/demo.dat  
/tmp/demo.dat
```

When the changes are committed, the background job will execute shortly thereafter and the file will be removed.

```
SQL> commit;
```

```
Commit complete.
```

```
SQL> !ls /tmp/demo.dat  
ls: 0653-341 The file /tmp/demo.dat does not exist.
```

NOTE You can see additional examples of the usefulness of `DBMS_JOB` in Chapter 6, *Triggers*.

The list of functionality delivered for free goes on and on. We stress again, when a new version of Oracle is released, take the time to check out what PL/SQL packages are provided. They might save you a bundle of time and energy.

Summary

We recommend that every PL/SQL program you write (except the function-based index example discussed earlier) be defined as a package. Even if a routine appears to be entirely self-contained and thus unlikely to be affected by dependency or recompilation issues, the one fact that never changes in the application development business is that applications change. If a self-contained program performs a useful function, it's probably a case of *when*, not *if*, someone will write a program that calls it, and thus introduce the more complicated dependencies that make packages worth their weight in gold. The extra effort of just a few extra keystrokes to code a package instead of a standalone procedure or function is well worth it.

