



P r o f e s s i o n a l E x p e r t i s e D i s t i l l e d

Advanced Oracle PL/SQL Developer's Guide

Second Edition

Master the advanced concepts of PL/SQL for professional-level certification and learn the new capabilities of Oracle Database 12c

Saurabh K. Gupta

[PACKT] enterprise 
PUBLISHING professional expertise distilled

Advanced Oracle PL/SQL Developer's Guide

Second Edition

Master the advanced concepts of PL/SQL for
professional-level certification and learn the new
capabilities of Oracle Database 12c

Saurabh K. Gupta



BIRMINGHAM - MUMBAI

Advanced Oracle PL/SQL Developer's Guide

Second Edition

Copyright © 2016 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: May 2012

Second edition: February 2016

Production reference: 1080216

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78528-480-9

www.packtpub.com

Credits

Author

Saurabh K. Gupta

Copy Editor

Stephen Copestake

Reviewers

Kamran Aghayev A

Patrick Barel

Nassyam Basha

Ramakrishna Kandula

Wisseem EL Khlifi

Sean Stacey

Davor Zelic

Project Coordinator

Shweta H Birwatkar

Proofreader

Safis Editing

Indexer

Monica Ajmera Mehta

Commissioning Editor

Priya Singh

Graphics

Abhinash Sahu

Acquisition Editor

Tushar Gupta

Production Coordinator

Nilesh R. Mohite

Content Development Editor

Arwa Manasawala

Cover Work

Nilesh R. Mohite

Technical Editor

Rohan Uttam Gosavi

About the Author

Saurabh K. Gupta is a seasoned database technologist with extensive experience in designing high performance and highly available database applications. His technology focus has been centered around Oracle Database architecture, Oracle Cloud platform, Database In-Memory, Database Consolidation, Multitenant, Exadata, Big Data, and Hadoop. He has authored the first edition of this book. He is an active speaker at technical conferences from Oracle Technology Network, IOUG Collaborate'15, AIOUG Sangam, and Tech Days. Connect with him on his twitter handle (or SAURABHKG) or through his technical blog www.sboracle.wordpress.com, with comments, suggestions, and feedback regarding this book.

About the Reviewers

Patrick Barel is a PL/SQL developer for AMIS Services (<http://www.amis.nl/>) in the Netherlands. Besides working with SQL and PL/SQL, he co-developed CodeGen together with Steven Feuerstein, and has written different plugins (<http://bar-solutions.com/>) for PL/SQL developer at (<http://www.allroundautomations.com/>). He publishes articles on AMIS Technology Blog (<http://technology.amis.nl/blog>) and on his own blog (<http://blog.bar-solutions.com>).

He has been a reviewer for several books including *Oracle PL/SQL Programming* by Steven Feuerstein. He has been an Oracle ACE since 2011.

Nassyam Basha is a database administrator and an Oracle ACE Director. He holds a master's degree in Computer Applications from the University of Madras. He is an Oracle 11g Certified Master and Exadata implementation specialist, and has good knowledge of Oracle technologies, such as Data Guard, RMAN, RAC, and Exadata. He actively participates in Oracle-related forums, such as OTN, where he has superhero status. He maintains an Oracle-technology-related blog (www.oracle-ckpt.com) and has coauthored *Oracle Data Guard 11gR2 administration beginners guide*, Packt Publishing. He actively writes many articles on OTN in various languages. He is a speaker at OTN, IOUG, and SANGAM, and he is the co-founder of Oraworld-team (www.oraworld-team.com). He is part of the AIOUG community on Twitter, where he occasionally expresses his views via the Twitter handle @AIOUG. He is currently working with Pythian as an Oracle database consultant.

Nassyam Basha has written *Oracle Data Guard 11gR2 Beginner's Guide*, Packt Publishing.

I want to thank the almighty Allah and my parents, Abdul Aleem and Rahimunnisa, for their support and blessings all the time – without them, nothing is possible. Special thanks to my wife and 9-month-old daughter Yashfeen Fathima, who've shared a lot of fun and crazy things with me while I worked on this book, and, as always, I would also like to thank my brother, Nawaz, and my cousins, for their great support. Finally, thanks to Saurabh Gupta for referring me as a technical reviewer, which was not an easy task for me, as this is my first assignment as a reviewer. He did a great job on this book.

Wissem El Khlifi is the first Oracle ACE in Spain and an Oracle Certified Professional DBA with over 12 years of IT experience.

He earned his Computer Science engineering degree from FST Tunisia, his master's degree in Computer Science from the UPC, Barcelona, and another master's degree in Big Data Science from the UPC, Barcelona.

His areas of interest are Linux System Administration, Oracle ERP and Databases (RAC and Dataguard), big data NoSQL database management, and big data analysis.

His career has included the roles of Oracle and Java analyst/programmer, Oracle DBA, architect, team leader, and big data scientist. He currently works as senior database and application engineer for Schneider Electric/APC.

He writes numerous articles on his website, <http://www.oracle-class.com>, and you can contact him via Twitter at @orawiss.

Davor Zelic is an IT professional with more than 15 years of experience in designing, developing, and implementing IT systems.

After getting his master's degree in Electrical Engineering, he began his professional career working with Oracle technology in the Croatian IT company TEB Informatika. For more than 10 years, Davor worked on IT projects related to road management, where he gained extensive experience working as an Oracle SQL, PLSQL, Forms, and Reports and Spatial developer. He has proved his knowledge by becoming an Oracle Certified Professional issued by the Oracle Corporation.

Apart from Oracle technology, Davor has gained expertise in design and development of geographic information systems for collection, storage, transformation, analysis, and visualization of geo-referenced data. He originally worked with Intergraph technology, but later his focus moved to open source GIS technologies, such as Geoserver and OpenLayers.

Davor currently works as a software architect at the IT department of Croatian Central Bank, designing software solutions for Croatian financial market data collection and analysis.

I want to thank my parents for the support that they gave me in choosing my educational path, which allowed me to find a job that is not just a routine, but also a source of satisfaction and constant challenge.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com, and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.



<https://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Instant updates on new Packt books

Get notified! Find out when new books are published by following @PacktEnterprise on Twitter or the *Packt Enterprise* Facebook page.

Table of Contents

Preface	xi
Chapter 1: Overview of PL/SQL Programming Concepts	1
Introduction to PL/SQL	2
PL/SQL program fundamentals	3
Cursors – an overview	5
The cursor execution cycle	5
Cursor attributes	6
Cursor FOR loop	8
Exception handling in PL/SQL	9
System-defined exceptions	9
User-defined exceptions	11
The RAISE_APPLICATION_ERROR procedure	13
Exception propagation	16
Creating stored procedures	17
Executing a procedure	19
Functions	20
Functions – execution methods	21
Restrictions on calling functions from SQL expressions	22
A PL/SQL package	23
Oracle Database 12c enhancements to PL/SQL subprograms	25
Managing database dependencies	26
Displaying the direct and indirect dependencies	26
Dependency metadata	27
Dependency issues and enhancements	27
Reviewing Oracle-supplied packages	28
Oracle SQL Developer	30
Oracle SQL Developer for DBA, Developers, and Application Architects	30
SQL Developer 4.0	31

Summary	32
Practice exercise	32
Chapter 2: Oracle 12c SQL and PL/SQL New Features	35
Database consolidation and the new Multitenant architecture	36
The Oracle Database 12c Multitenant architecture – features	38
Multitenant for Consolidation	39
Plug/unplug	39
Manage Many as One	40
Rapid provisioning	40
CDB Resource Management	41
Common users and local users	42
Oracle 12c SQL and PL/SQL new features	43
IDENTITY columns	43
Default column value to a sequence in Oracle 12c	45
The DEFAULT ON NULL clause	45
Support for 32K VARCHAR2	46
Row limiting using FETCH FIRST	47
Invisible columns	49
Temporal databases	51
In-Database Archiving	53
Defining a PL/SQL subprogram in the SELECT query and PRAGMA UDF	55
Test setup	57
Comparative analysis	60
The PL/SQL program unit white listing	61
Granting roles to PL/SQL program units	62
Test setup	63
Miscellaneous PL/SQL enhancements	67
The Oracle Database 12c (12.1.0.2) In-Memory option	67
The challenge	68
The problem statement and Oracle Database 12c In-Memory	68
Oracle Database 12c In-Memory option features	68
The Oracle Database 12c In-Memory Architecture	69
Controlling the In-Memory column store	70
The INMEMORY clause	70
Performance optimizations	72
In-Memory Advisor	72
Oracle Database In-Memory benefits	73
Summary	73
Chapter 3: Designing PL/SQL Code	75
Cursor structures	76
Cursor execution cycle	76
Cursor attributes	78

Implicit cursors	78
Explicit cursors	80
Cursor variables	83
Strong and weak ref cursor types	85
Working with cursor variables	86
SYS_REFCURSOR	87
Cursor variables as arguments	88
Cursor variables – restrictions	89
Cursor design considerations	89
Cursor design–guidelines	90
Implicit statement results in Oracle Database 12c	91
Subtypes	93
Subtype classification	93
Type compatibility with subtypes	95
Summary	95
Practice exercise	96
Chapter 4: Using Collections	99
Introduction to collections	100
Collection types	100
Associative arrays	101
Nested tables	104
Modify and drop a nested table object type	105
Design considerations of a nested table	105
Nested table storage	105
Nested table in an index - organized table	105
Nested table locators	106
Nested table as the schema object	106
Operations on a nested table type column	107
Nested table collection type in PL/SQL	110
Querying the nested table metadata	112
Nested table comparison functions	113
Multiset operations on nested tables	114
Varray	116
Varray as a schema object	117
Operations on varray type columns	119
Varray in PL/SQL	122
Comparing the collection types	122
Selecting the appropriate collection type	124
Oracle 12c enhancements to collections	124
PL/SQL collection methods	127
EXISTS	127
COUNT	128

LIMIT	129
FIRST and LAST	129
PRIOR and NEXT	130
EXTEND	131
TRIM	132
DELETE	133
Summary	135
Practice exercise	136
Chapter 5: Using Advanced Interface Methods	139
Overview of External Procedures	139
External Procedures	140
Components of external procedure execution flow	140
The extproc agent	141
The Library object	141
Callout and Callback	142
Call Specification	142
How an External Procedure executes	143
Environment setup	143
TNSNAMES.ora	144
EXTPROC.ora	144
Executing external C programs from PL/SQL	145
Securing External Procedures with Oracle Database 12c	149
Executing Java programs from PL/SQL	150
Loading a Java class into a database	150
Steps to execute a Java class from an Oracle PL/SQL unit	151
Summary	155
Practice exercise	155
Chapter 6: Virtual Private Database	159
Oracle Database Security overview	160
Fine-Grained Access Control	163
How FGAC works	163
Virtual Private Database	164
How does Virtual Private Database work?	164
Column-level Virtual Private Database	164
Virtual Private Database with Oracle Database 12c Multitenant	165
Virtual Private Database components	165
Application Context	165
Virtual Private Database policy function	168
Policy types	169
The DBMS_RLS package	169
Demonstration	170
Virtual Private Database features and best practices	177

Virtual Private Database metadata	178
Policy utilities—refresh and drop	179
Oracle Database 12c Security enhancements	180
Oracle Database 12c Data Redaction	181
Data Redaction exemptions and miscellaneous features	182
Data Redaction function types	182
Demonstration	183
The Data Redaction metadata	186
Summary	187
Practice exercise	187
Chapter 7: Oracle SecureFiles	191
Introduction to Large Objects	192
Classification of Large Object datatypes	193
Internal LOB	193
External LOB	194
LOB restrictions	194
LOB data types in Oracle	194
BLOB and CLOB	194
BFILE	195
Some more related stuff	195
The LOB locator	195
LOB instance initialization	196
The DBMS_LOB package	197
LOB usage notes	201
Oracle SecureFiles	201
Deduplication and compression	201
Encryption	202
File System Logging	203
Write Gather Cache	203
Free space management	203
BasicFiles and SecureFiles	204
The db_securefile parameter	204
Working with LOBs	205
LOB metadata	207
Enabling the advanced features of a SecureFile	209
Populating the LOB data	212
Temporary LOB operations	215
Managing temporary LOBs	215
Working with a temporary LOB	216
Migrating LONG to LOBs	218
Use the ALTER TABLE command	219
Using the TO_LOB function	219

Online Table Redefinition	220
Migrating BasicFiles to SecureFiles	220
Oracle Database 12c enhancements to SecureFiles	223
Summary	223
Practice exercise	224
Chapter 8: Tuning the PL/SQL Code	229
The PL/SQL Compiler	230
Subprogram inlining in PL/SQL	230
PRAGMA INLINE	231
PLSQL_OPTIMIZE_LEVEL	232
Native and interpreted compilation techniques	238
Oracle Database 11g Real Native Compilation	239
Selecting the appropriate compilation mode	240
Setting the compilation mode	240
Querying the compilation settings	241
Compiling a program unit for native or interpreted compilation	241
Recompiling a database for a PL/SQL native or interpreted compilation	243
Tuning PL/SQL code	247
Build secure applications using bind variables	247
Call parameters by reference	251
Avoiding an implicit data type conversion	252
Understanding the NOT NULL constraint	253
Selection of an appropriate numeric data type	254
Bulk processing in PL/SQL	257
BULK COLLECT	257
FORALL	262
Summary	270
Practice exercise	271
Chapter 9: Result Cache	275
Oracle Database 11g Result Cache	276
What is the Server Result Cache?	277
Configuring the Server Result Cache	278
Result Cache versus Buffer Cache	280
Result Cache versus Oracle 12c Database In-Memory	280
Result Cache versus In-Memory Database Cache	280
SQL query Result Cache	281
Monitoring the SQL Result Cache	284
Invalidation of the SQL Result Cache	288
Read consistency of the SQL Result Cache	289
Limitations	289

PL/SQL Function Result Cache	290
Does it sound similar to deterministic functions?	290
Differences between Result Cache and other caching techniques	291
Illustration	291
Monitoring the PL/SQL Result Cache	293
Invalidation of the PL/SQL Result Cache	294
Limitation	294
OCI Client results cache	295
The DBMS_RESULT_CACHE package	295
Displaying the result cache memory report	297
Oracle Database 12c enhancements to the PL/SQL function Result Cache	298
Result cache in Real Application Clusters	299
Summary	300
Practice exercise	301
Chapter 10: Analyzing, Profiling, and Tracing PL/SQL Code	305
A sample PL/SQL program	305
Tracking PL/SQL coding information	306
USER_ARGUMENTS	307
USER_OBJECTS	307
USER_OBJECT_SIZE	308
USER_SOURCE	308
USER_PROCEDURES	310
USER_PLSQL_OBJECT_SETTINGS and USER_STORED_SETTINGS	310
USER_DEPENDENCIES	312
The DBMS_DESCRIBE package	312
Tracking the program execution subprogram call stack	315
Tracking propagating exceptions in PL/SQL code	319
Determining identifier types and usages	323
USER_IDENTIFIERS	323
The PL/Scope tool	324
The PLScope_SETTINGS parameter	325
The DBMS_METADATA package	328
DBMS_METADATA data types and subprograms	328
Parameter requirements	331
The DBMS_METADATA transformation parameters and filters	331
Demonstration	333
Tracing PL/SQL programs using DBMS_TRACE	336
Installing the DBMS_TRACE package	337
DBMS_TRACE subprograms	337
Compiling a PL/SQL program for debugging	339

Viewing the PL/SQL trace information	340
Steps to trace PL/SQL program execution	341
Profiling PL/SQL code	342
The DBMS_HPROF package	343
Differences between DBMS_PROFILER and DBMS_HPROF	343
DBMS_HPROF subprograms	344
Collecting raw profile data	344
Interpreting the raw profiler data	345
Analyzing profiler data	346
Creating the profiler tables	346
Analyzing the profiler output	347
Querying the profiler tables	347
The plshprof utility	348
What do these reports reveal?	350
Summary	351
Practice exercise	351
Chapter 11: Safeguarding PL/SQL Code against SQL injection	353
What is SQL injection?	354
SQL injection targets	354
How to exploit the PL/SQL code?	355
Preventing SQL injection attacks	359
Sanitizing inputs using DBMS_ASSERT	360
Choose the right subprogram for the right identifier	361
DBMS_ASSERT – limitations	364
Use of bind variables to prevent injection attacks	365
Best practices to avoid SQL injection	366
Testing the code for SQL injection flaws	367
Test strategy	367
An effective code review	367
Static code analysis	368
Fuzz tools	368
Generating test cases	369
Summary	369
Practice exercise	369
Chapter 12: Working with Oracle SQL Developer	373
An overview of SQL Developer	374
Key differentiators	374
History and background	375
SQL Developer for Developers	376
SQL Developer for Database Administrators	376
SQLcl – The new SQL command line	377

Getting started with SQL Developer	379
Creating a database connection	380
Using the SQL worksheet	381
Core features of SQL Developer	383
Object Browser	383
PL/SQL Editor and Debugger	383
DBA Panel	384
Database Utilities	385
The Data Modeler	386
SQL Developer reports	386
Version control	387
The SQL Translation Framework	388
SQL Developer 4.0 and 4.1 New Features	388
Summary	390
Index	391

Preface

How many of us would believe that PL/SQL was introduced as a scripting language for executing a bunch of SQL scripts? Well, that's true. With the growing need to build computational logic and condition-based constructs, and to manage exception rules within databases, Oracle Corporation first released PL/SQL along with Oracle Database Version 6.0 with a limited set of capabilities. Within its capacity, PL/SQL was capable of creating program units that could not be stored inside the database. Eventually, Oracle's release in the application line, SQL *Forms version V3.0, included the PL/SQL engine and allowed developers to implement the application logic through procedures. Back then, PL/SQL used to be part of the transaction processing option in Oracle 6 and the procedural option in Oracle 7. Since the time of its ingenuous beginning, PL/SQL has matured immensely as a standard feature of Oracle Database. It has been enthusiastically received by the developer community, and the credit goes to its support for advanced elements such as modular programming, encapsulation, support for objects and collections, program overloading, native and dynamic SQL, and exception handling.

PL/SQL is loosely derived from Ada (named after Ada Lovelace, an English mathematician who is regarded as the first computer programmer), a high-level programming language, which complies with the advanced programming elements. Building a database backend for an application demands the ability to design the database architecture, skills to code complex business logics, and expertise in administering and protecting the database environment. One of the principal reasons why PL/SQL is a key enabler in the development phase is its tight integration with Oracle's SQL language. In addition to this, it provides a rich platform for implementing the business logic in the Oracle Database layer and store them as procedures or functions for subsequent use. As a procedural language, PL/SQL provides a diverse range of datatypes, iterative and control constructs, conditional statements, and exception handlers.

In a standard software development space, an Oracle database developer is expected to get involved in schema design; code business logics on the server side by using functions, procedures, or packages; implement action rules by using triggers; and support client-side programs in setting up the application interface. While building the server-side code, developers should understand that their code contributes to the application's performance and scalability. Language basics are expected to be resilient, but while building robust and secure applications using PL/SQL, developers must take advantage of best practices and try to use advanced language features. This book focuses on the advanced features of PL/SQL validated up to the latest Oracle Database 12c.

Learning by example has always been a well-attested approach for diving deep into a concept. This book will enable you to master the latest enhancements and new features of Oracle Database 12c. For efficient reading, you just have to be familiar with the PL/SQL fundamentals so that you can relate to the evolution of an advanced feature from its ever-expanding roots.

This book closely follows the outline of the Oracle University certification; that is, the Oracle Certified Advanced PL/SQL Developer Professional (1Z0-146) exam. One of the most sought after certifications in the developer community, the 1Z0-146 certification's objectives are quite comprehensive, and touch upon the various progressive areas of PL/SQL. To name a few, PL/SQL code performance, maintenance, bulk processing techniques, PL/SQL collections, security implementation, and the handling of large objects. For certification aspirants, this book will serve as a one-stop exam guide. At many stages, this book goes beyond the certification objectives and attempts to build a deep understanding of the concepts. Therefore, mid-level database developers will find this book a handy language reference and would be keen to have it on their bookshelves.

My last work on the same subject will remain close to my heart, but this one is straight from my experience. I hope that this book will help you improve your PL/SQL development skills and gain confidence in using advanced features, along with meticulous familiarization of Oracle Database 12c.

"The only real security that a man can have in the world is a reserve of knowledge, experience and ability"

- Henry Ford

What this book covers

Chapter 1, Overview of PL/SQL Programming Concepts, provides an overview of PL/SQL fundamentals. It refreshes the basic concepts, such as PL/SQL language features, the anonymous block structure, exception handling, and stored subprograms.

Chapter 2, Oracle 12c SQL and PL/SQL New Features, talks about the new features of Oracle Database 12c. It starts with the idea of consolidation of databases on a cloud and how the Oracle 12c Multitenant architecture addresses the requirements. It consolidates the new features in Oracle 12c SQL and PL/SQL, and explains each of them with examples. It will help you to feel the essence of Oracle Database 12c and understand what the driving wheel of innovation is. A section on the Oracle Database 12c In-memory option will familiarize you with the breakthrough feature in the analytics and warehouse space.

Chapter 3, Designing PL/SQL Code, primarily focuses on the PL/SQL cursor's design and handling. You will get to learn the basics of cursor design, cursor types and cursor variables, handling cursors in PL/SQL, and design guidelines. This chapter will also include the enhancements made by Oracle Database 12c with respect to cursors.

Chapter 4, Using Collections, introduces you to the world of collections; namely, associative arrays, nested tables, and varrays. Taking you all the way from their creation in SQL and PL/SQL to design considerations, this chapter makes you wise enough to choose the right collection type in a given situation. A section on Oracle Database 12c enhancements to collections introduces a very handy feature that will allow you to join a table and collection.

Chapter 5, Using Advanced Interface Methods, focuses on a powerful feature of PL/SQL: how to execute external procedures in PL/SQL. You will learn and understand the specifics of executing a C or Java program in PL/SQL as an external procedure through step-by-step demonstration. This chapter also mentions the Oracle Database 12c enhancement which allows you to secure external procedures through an additional safety net.

Chapter 6, Virtual Private Database, provides a detailed overview of the Oracle Database Security Defense-in-depth architecture and focuses on one of the developer-centric features, known as the Virtual Private Database. Oracle Database 12c security enhancements and a demonstration of data redaction will make you understand Oracle's security offerings.

Chapter 7, Oracle SecureFiles, provides a thorough understanding of handling large objects in Oracle and focuses on storage optimizations made by SecureFiles. Introduced in Oracle 11g, SecureFiles is the new storage mechanism that scores high on its advanced features, such as compression, encryption, and deduplication. This chapter also helps you with the recommended migration methods from older LOBs to SecureFiles.

Chapter 8, Tuning the PL/SQL Code, introduces the best practices for tuning PL/SQL code. It starts with the PL/SQL optimizer and rolls through the benefits of native compilation, PL/SQL code writing skills, and code evaluation design. This chapter includes the changes in Oracle 12c with respect to large object handling.

Chapter 9, Result Cache, explains the result caching feature in Oracle Database. It is a powerful caching mechanism that enhances the performance of SQL queries and PL/SQL functions that are repeatedly executed on the server. This chapter also discusses the enhancements made to the feature in Oracle Database 12c.

Chapter 10, Analyzing, Profiling, and Tracing PL/SQL Code, details the techniques used to analyze, profile, and trace PL/SQL code. If you are troubleshooting PL/SQL code for performance, you must learn the profiling and tracing techniques. In an enterprise application environment, these practices are vital weapons in a developer's arsenal.

Chapter 11, Safeguarding PL/SQL Code against SQL injection, describes ways to protect your PL/SQL from being attacked. A vulnerable piece of code is prone to malicious attacks and runs the risk of giving away sensitive information. Efficient code writing and proofing the code from external attacks can help to minimizing the attack surface area. In this chapter, you will learn the practices for safeguarding your code against external threats.

Chapter 12, Working with Oracle SQL Developer, describes the benefits of the Oracle SQL Developer for developers, database administrators, and architects. This chapter not only helps you get started with SQL Developer, but also helps you gain a better understanding of the new features of SQL Developer 4.0 and 4.1.

What you need for this book

If you are good with PL/SQL development basics, I'm sure you will enjoy reading this book. You will learn new ways to program efficiently in PL/SQL.

Who this book is for

This book is for Oracle developers who are responsible for database management. Readers are expected to have basic knowledge of the Oracle Database and the fundamentals of PL/SQL programming. Certification aspirants can use this book to prepare for the 1Z0-146 examination in order to become an Oracle Certified Professional in Advanced PL/SQL.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

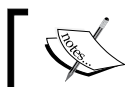
Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The modified `SELECT` query is then executed in the HR schema of the database."

A block of code is set as follows:

```
/*Create the stored procedure to set the context attribute*/
CREATE OR REPLACE PROCEDURE p_app_context (p_val VARCHAR2)
IS
BEGIN

    /*Create a namespace DEMO_CONTEXT*/
    DBMS_SESSION.SET_CONTEXT(
        NAMESPACE => 'DEMO_CONTEXT',
        ATTRIBUTE => 'COUNTRY',
        VALUE      => P_VAL);
END;
/
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "This is how interpreted compilation works. In the case of native compilation,, a sharable **dynamic linked library (DLL)** is generated instead of a machine code."



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Overview of PL/SQL Programming Concepts

Structured Query Language (SQL) is a language that has been widely accepted and adopted for accessing relational databases. This language allows users to perform database operations such as reading, creating, modifying, and deleting the data. Since the summer of 1970, when Dr. E.F. Codd published the paper *A Relational Model of Data for Large Shared Data Banks* for the ACM journal, the language has matured comprehensively as an industry standard. With its broad range of features and easy adaptation to enterprise environments, the SQL language has been typically regarded as the most reliable language for interacting with relational databases.

PL/SQL was developed in 1991 by Oracle Corporation as a procedural language extension to SQL. Its ability to integrate seamlessly with SQL makes it a powerful language to construct the data access layer and the rich procedural extensions help in translating business logic within the Oracle Database. This first chapter introduces you to the PL/SQL language and refreshes some of the key programming concepts. The chapter is outlined as follows:

- Introduction to PL/SQL
- Recapitulate procedures, functions, packages, and cursors
- Exception handling
- Object dependencies
- Major Oracle supplied packages
- Oracle Development tools—SQL Developer and SQL*Plus

Introduction to PL/SQL

PL/SQL stands for **Procedural Language-Structured Query Language(PL/SQL)**. It is part of the Oracle Database product, which means no separate installation is required. It is commonly used to translate business logic in the database and expose the program interface layer to the application. While SQL is purely a data access language that directly interacts with the database, PL/SQL is a programming language in which multiple SQLs and procedural statements can be grouped in a program unit. PL/SQL code is portable between Oracle Databases (subject to limitations imposed by versions). The built-in database optimizer refactors the code to improve the execution performance.

The advantages of PL/SQL as a language are as follows:

- PL/SQL supports all types of SQL statements, data types, static SQL, and dynamic SQL
- PL/SQL code runs on all platforms supported by the Oracle Database
- PL/SQL code performance can be improved by the use of bind variables in direct SQL queries
- PL/SQL supports the object-oriented model of the Oracle Database
- PL/SQL applications increase scalability by allowing multiple users to invoke the same program unit

Although it is not used to build user interfaces, it provides the opportunity to build robust, secure, and portable interface layers, which can be exposed to a high-level programming language. Some of the key faculties of PL/SQL (PL/SQL accomplishments) are listed here:

- **A procedural language:** A PL/SQL program can include a list of operations that can execute sequentially to get the desired result. Unlike SQL, which is just a declarative language, PL/SQL adds selective and iterative constructs to it.
- **Database programming language:** Server side programs run faster than the middle-tier programs. Code maintenance becomes easy as it needs to be re-written less frequently.
- **An integral language:** Application developers can easily integrate a PL/SQL program with other high-level programming interfaces such as Java, C++, or .NET. The PL/SQL procedures or subprograms can be invoked from client programs as executable statements.

PL/SQL program fundamentals

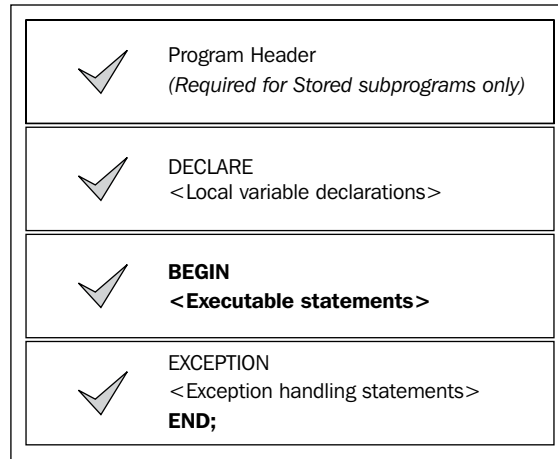
A well-written PL/SQL program should be able to answer the following fundamental questions:

- How do we handle an SQL execution in the program?
- How do we handle the procedural execution flow in the program?
- Does the program handle the exceptions?
- How do we maintain (trace and debug) the PL/SQL program code?

Well, there are multiple tips and techniques to standardize PL/SQL coding practices. But before we drill down to the programming skills, let us familiarize ourselves with the structure of a PL/SQL program. A PL/SQL program can be broken down into four sections. Each section carries a specific objective and must exist in the same sequence in a program. Let us have a brief look at the sections:

- **Header:** This is an optional section which is required for named blocks such as procedures, functions, and triggers. It contains the program name, the program's owner, and the parameter specification.
- **Declaration:** This is an optional section used to declare local variables, cursors, and local subprograms that are likely to be used in the program body. The `DECLARE` keyword indicates the beginning of the declaration section. The section can be skipped if the PL/SQL program uses no variables.
- **Execution:** This is the procedural section of the program and comprises the main program body and an exception section. The `BEGIN` and `END` keywords indicate the beginning and end of the program body. It must contain at least one executable statement. During block execution, these statements are parsed and sequentially executed by the PL/SQL engine.
- **Exception:** This is an optional section in the program body that contains a set of instructions as procedural statements, for various errors, that may occur in the program leading to abnormal termination. The program control lands into the exception section and the appropriate exception handler is executed. The `EXCEPTION` keyword indicates the start of the exception section.

The following block diagram shows the structure of a PL/SQL block:



A PL/SQL block is the elementary unit of a program that groups a set of procedural statements. Based on the sections included in a PL/SQL program unit, we can classify a program under following categories:

- **Anonymous PL/SQL block:** This is the simplest PL/SQL program that has no name, but has its DECLARE-BEGIN-END skeleton. It can either be run for current execution as standalone block or embedded locally within a PL/SQL program unit. An anonymous block cannot be stored in the database.
- **Named:** This block is a named PL/SQL routine that is stored persistently in the database as a schema object. It can be invoked either from a database session or by another program unit. A named PL/SQL program can be a function, procedure, trigger, or package.
- **Nested:** A block within another PL/SQL block forms a nested block structure.

So, let's get started with our first anonymous PL/SQL block. The block declares a string and displays it on screen. Note that each line in the program ends with a semi-colon and the block ends with a slash (/) for code execution.

```
/*Enable the Serveroutput to display block messages*/
SET SERVEROUTPUT ON
```



The SERVEROUTPUT parameter is a SQL*Plus variable that enables the printing of DBMS_OUTPUT messages from a PL/SQL block.

```
/*Start the PL/SQL block*/
```

```

DECLARE
/*Declare a local variable and initialize with a default value*/
  L_STR VARCHAR2(50) := 'I am new to PL/SQL';
BEGIN
/*Print the result*/
  DBMS_OUTPUT.PUT_LINE('I Said - ' || L_STR);
END;
/
I Said - I am new to PL/SQL

```

PL/SQL procedure successfully completed.

Cursors – an overview

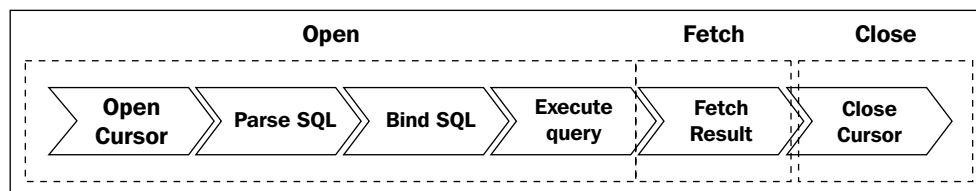
Writing SQL in PL/SQL is one of the critical parts of database programming. All SQL statements embedded within a PL/SQL block are executed as a cursor. A cursor is a private memory area, temporarily allocated in the session's **User Global Area (UGA)**, that is used for processing SQL statements. The private memory stores the result set retrieved from the SQL execution and cursor attributes. Cursors can be classified as **implicit** and **explicit** cursors.

Oracle creates an implicit cursor for all the SQL statements included in the executable section of a PL/SQL block. In this case, the cursor lifecycle is maintained by the Oracle Database.

For explicit cursors, the execution cycle can be controlled by the user. Database developers can explicitly declare an implicit cursor under the `DECLARE` section along with a `SELECT` query.

The cursor execution cycle

A cursor moves through the following stages during execution. Note that, in the case of an implicit cursor, all the steps are carried out by the Oracle Database. Let's take a quick look at the execution stages `OPEN`, `FETCH`, and `CLOSE`.




- The **OPEN** stage allocates the context area in the session's **User Global Area** for performing SQL processing. The SQL processing starts with parsing and binding, followed by statement execution. In the case of the **SELECT** query, the record pointer points to the first record in the result set.
- The **FETCH** stage pulls the data from the query result set. If the result set is a multi-record set, the record pointer moves incrementally with every fetch. The fetch stage is alive until the last record is reached in the result set.
- The **CLOSE** stage closes the cursor, flushes the context area, and releases the memory back to the UGA.

Cursor attributes

Cursor attributes hold the information about the cursor processing at each stage of its execution:

- **%ROWCOUNT**: Number of rows fetched until the last fetch or impacted by the last DML operation. Applicable for **SELECT** as well as DML statements.
- **%ISOPEN**: Boolean **TRUE** if the cursor is still open, if not **FALSE**. For an implicit cursor, this attribute is always **FALSE**.
- **%FOUND**: Boolean **TRUE**, if the fetch operation switches and points to a record; if not, **FALSE**.
- **%NOTFOUND**: Boolean **FALSE** when the cursor pointer switches but does not point to a record in the result set.

 **%ISOPEN** is the only cursor attribute that is accessible outside the cursor execution cycle.

The following program uses the cursor attributes **%ISOPEN**, **%NOTFOUND**, and **%ROWCOUNT** to fetch the data from the **EMP** table and display it:

```
/*Enable the SERVEROUTPUT to display block messages*/
SET SERVEROUTPUT ON

/*Start the PL/SQL Block*/
DECLARE

/*Declare a cursor to select employees data*/
    CURSOR C_EMP IS
        SELECT EMPNO,ENAME
        FROM EMP;
    L_EMPNO EMP.EMPNO%TYPE;
```

```
L_ENAME EMP.ENAME%TYPE;
BEGIN

/*Check if the cursor is already open*/
IF NOT C_EMP%ISOPEN THEN
    DBMS_OUTPUT.PUT_LINE('***Displaying Employee Info***');
END IF;

/*Open the cursor and iterate in a loop*/
OPEN C_EMP;
LOOP

/*Fetch the cursor data into local variables*/
FETCH C_EMP INTO L_EMPNO, L_ENAME;
EXIT WHEN C_EMP%NOTFOUND;

/*Display the employee information*/
    DBMS_OUTPUT.PUT_LINE(chr(10)||'Display Information for
employee:'||C_EMP%ROWCOUNT);
    DBMS_OUTPUT.PUT_LINE('Employee Id:'||L_EMPNO);
    DBMS_OUTPUT.PUT_LINE('Employee Name:'||L_ENAME);
END LOOP;
END;
/

***Displaying Employee Info***

Display Information for employee:1
Employee Id:7369
Employee Name:SMITH

Display Information for employee:2
Employee Id:7499
Employee Name:ALLEN

Display Information for employee:3
Employee Id:7521
Employee Name:WARD

Display Information for employee:4
Employee Id:7566
Employee Name:JONES
...
PL/SQL procedure successfully completed.
```

Cursor FOR loop

Looping through all the records of a cursor object can be facilitated with the use of the FOR loop. A FOR loop opening a cursor directly is known as a CURSOR FOR loop. The usage of the CURSOR FOR loop reduces the overhead of manually specifying the OPEN, FETCH, and CLOSE stages of a cursor.

The CURSOR FOR loop will best compact the code when working with multi-row explicit cursors. The following PL/SQL block demonstrates the purpose:

```
/*Enable the SERVEROUTPUT parameter to print the results in the
environment*/
SET SERVEROUTPUT ON

/*Start the PL/SQL block*/
DECLARE
/*Declare an explicit cursor to select employee information*/
    CURSOR CUR_EMP IS
        SELECT ename, sal
        FROM emp;
BEGIN
/*FOR Loop uses the cursor CUR_EMP directly*/
    FOR EMP IN CUR_EMP
    LOOP
/*Display message*/
        DBMS_OUTPUT.PUT_LINE(EMP.ename||' earns '||EMP.sal||' per month');
    END LOOP;
END;
/

SMITH earns 800 per month
ALLEN earns 1600 per month
WARD earns 1250 per month
JONES earns 2975 per month
MARTIN earns 1250 per month
BLAKE earns 2850 per month
CLARK earns 2450 per month
SCOTT earns 3000 per month
KING earns 5000 per month
TURNER earns 1500 per month
ADAMS earns 1100 per month
JAMES earns 950 per month
FORD earns 3000 per month
MILLER earns 1300 per month
```

PL/SQL procedure successfully completed.

Note that, with the `CURSOR FOR` loop, you do not need to declare the block variables to capture the cursor columns. The `CURSOR FOR` loop index implicitly acts as a record of the cursor type. Also, you do not need to explicitly open or close the cursor in the PL/SQL program.

Exception handling in PL/SQL

If a program shows an unusual and unexpected flow during runtime, which might result in abnormal termination of the program, the situation is said to be an exception. Such errors must be trapped and handled in the `EXCEPTION` section of the PL/SQL block. The exception handlers can suppress the abnormal termination with an alternative and secured action.

Exception handling is one of the important steps of database programming. Unhandled exceptions can result in unplanned application outages, impact business continuity, and frustrate end users.

There are two types of exceptions—system-defined and user-defined. While the Oracle Database implicitly raises a system-defined exception, a user-defined exception is explicitly declared and raised within the program unit.

In addition, Oracle provides two utility functions, `SQLCODE` and `SQLERRM`, to retrieve the error code and message for the most recent exception.

System-defined exceptions

As the name implies, system-defined exceptions are defined and maintained implicitly by the Oracle Database. They are defined in the Oracle `STANDARD` package. Whenever an exception occurs inside a program, the database picks up the appropriate exception from the available list. All system-defined exceptions are associated with a negative error code (except 1 to 100) and a short name, which is used while specifying the exception handlers.

For example, the following PL/SQL program includes a `SELECT` statement to select details of employee 8376. It raises `NO_DATA_FOUND` exception because employee id 8376 doesn't exist.

```
SET SERVEROUTPUT ON

/*Declare the PL/SQL block */
DECLARE
    L_ENAME VARCHAR2 (100);
    L_SAL NUMBER;
    L_EMPID NUMBER := 8376;
```

```
BEGIN

/*Write a SELECT statement */
  SELECT ENAME, SAL
  INTO L_ENAME, L_SAL
  FROM EMP
  WHERE EMPNO = L_EMPID;
END;
/
```

```
DECLARE
*
ERROR at line 1:
ORA-01403: no data found
ORA-06512: at line 8
```

Let us rewrite the preceding PL/SQL block to include an EXCEPTION section and handle the NO_DATA_FOUND exception:

```
/*Enable the SERVEROUTPUT parameter to print the results in the
environment*/
SET SERVEROUTPUT ON

/*Start the PL/SQL block*/
DECLARE
  /*Declare the local variables*/
  L_ENAME VARCHAR2 (100);
  L_SAL NUMBER;
  L_EMPID NUMBER := 8376;
BEGIN
  /*SELECT statement to fetch the name and salary details of the
employee*/
  SELECT ENAME, SAL
  INTO L_ENAME, L_SAL
  FROM EMP
  WHERE EMPNO = L_EMPID;
EXCEPTION
  /*Exception Handler */
  WHEN NO_DATA_FOUND THEN
    /*Display an informative message*/
    DBMS_OUTPUT.PUT_LINE ('No Employee exists with the id '||L_EMPID);
END;
/
```

No Employee exists with the id 8376

PL/SQL procedure successfully completed.

The following table lists some of the commonly used system-defined exceptions along with their short name and ORA error code:

Error	Named exception	Comments (raised when:)
ORA-00001	DUP_VAL_ON_INDEX	Duplicate value exists
ORA-01001	INVALID_CURSOR	Cursor is invalid
ORA-01012	NOT_LOGGED_ON	User is not logged in
ORA-01017	LOGIN_DENIED	System error occurred
ORA-01403	NO_DATA_FOUND	The query returns no data
ORA-01422	TOO_MANY_ROWS	A single row query returns multiple rows
ORA-01476	ZERO_DIVIDE	An attempt was made to divide a number by zero
ORA-01722	INVALID_NUMBER	The number is invalid
ORA-06504	ROWTYPE_MISMATCH	Mismatch occurred in row type
ORA-06511	CURSOR_ALREADY_OPEN	Cursor is already open
ORA-06531	COLLECTION_IS_NULL	Working with NULL collection
ORA-06532	SUBSCRIPT_OUTSIDE_LIMIT	Collection index out of range
ORA-06533	SUBSCRIPT_BEYOND_COUNT	Collection index out of count

User-defined exceptions

Oracle allows users to create custom exceptions, specify names, associate error codes, and raise statements in line with the implementation logic. If PL/SQL applications are required to standardize the exception handling, not just to control the abnormal program flow but also to alter the program execution logic, you need to use user-defined exceptions. The user-defined exceptions are raised in the `BEGIN..END` section of the block using the `RAISE` statement.

There are three ways of declaring user-defined exceptions:

- Declare the `EXCEPTION` type variable in the declaration section. Raise it explicitly in the program body using the `RAISE` statement. Handle it in the `EXCEPTION` section. Note that no error code is involved here.
- Declare the `EXCEPTION` variable and associate it with a standard error number using `PRAGMA EXCEPTION_INIT`.



A **Pragma** is a directive to the compiler to manipulate the behavior of the program unit during compilation, and not at the time of execution.

PRAGMA EXCEPTION_INIT can also be used to map an exception to a non-predefined exception. These are standard errors from Oracle but not defined as PL/SQL exceptions.

- Use the RAISE_APPLICATION_ERROR to declare a dedicated error number and error message.

The following PL/SQL block declares a user-defined exception and raises it in the program body:

```
/*Enable the SERVEROUTPUT parameter to print the results in the
environment*/
SET SERVEROUTPUT ON

/*Declare a bind variable M_DIVISOR*/
VARIABLE M_DIVISOR NUMBER;

/*Declare a bind variable M_DIVIDEND*/
VARIABLE M_DIVIDEND NUMBER;

/*Assign value to M_DIVISOR as zero*/
EXEC :M_DIVISOR := 0;

PL/SQL procedure successfully completed.

/*Assign value to M_DIVIDEND as 10/
EXEC :M_DIVIDEND := 10;

PL/SQL procedure successfully completed.

/*Start the PL/SQL block*/
DECLARE
    /*Declare the local variables and initialize with the bind
    variables*/
    L_DIVISOR NUMBER := :M_DIVISOR;
    L_DIVIDEND NUMBER := :M_DIVIDEND;
    L_QUOT NUMBER;
    /*Declare an exception variable*/
    NOCASE EXCEPTION;
BEGIN
    /*Raise the exception if Divisor is equal to zero*/
```

```

        IF L_DIVISOR = 0 THEN
            RAISE NOCASE;
        END IF;
        L_QUOT := L_DIVIDEND/L_DIVISOR;
        DBMS_OUTPUT.PUT_LINE('The result : '||L_QUOT);
    EXCEPTION
        /*Exception handler for NOCASE exception*/
        WHEN NOCASE THEN
            DBMS_OUTPUT.PUT_LINE('Divisor cannot be equal to zero');
    END;
    /
    Divisor cannot be equal to zero

    PL/SQL procedure successfully completed.

    /*Assign a non zero value to M_DIVISOR*/
    EXEC :M_DIVISOR := 2;

    PL/SQL procedure successfully completed.

    /*Re-execute the block */
    SQL> /
    The result : 5

    PL/SQL procedure successfully completed.

```

The RAISE_APPLICATION_ERROR procedure

The `RAISE_APPLICATION_ERROR` is an Oracle-supplied procedure that raises a user-defined exception with a custom exception message. The exception can be optionally pre-defined in the declarative section of the PL/SQL.

The syntax for the `RAISE_APPLICATION_ERROR` procedure is as follows:

```

RAISE_APPLICATION_ERROR (error_number, error_message[, {TRUE |
FALSE}])

```

In this syntax, the `error_number` parameter is a mandatory parameter with the error value ranging between 20000 to 20999. `error_message` is the user-defined message that appears along with the exception. The last parameter is an optional argument that is used to add the exception error code to the current error stack.

The following PL/SQL program lists the employees who have joined the organization after the given date. The program must raise an exception if the date of joining is before the given date. The block uses `RAISE_APPLICATION_ERROR` to raise the exception with an error code 20005, and an appropriate error message appears on the screen:

```
/*Enable the SERVEROUTPUT parameter to print the results in the
environment*/
SET SERVEROUTPUT ON

/*Start the PL/SQL block */
DECLARE

/*Declare the birth date */
    L_DOB_MON DATE := '01-DEC-1981';

/*Declare a cursor to filter employees who were hired on birthday
month*/
    CURSOR C IS
        SELECT empno, ename, hiredate
        FROM emp;
BEGIN
    FOR I IN C
    LOOP

        /*Raise exception, if birthdate is later than the hiredate */
        IF i.hiredate < l_dob_mon THEN
            RAISE_APPLICATION_ERROR (-20005,'Hiredate earlier than the
given date!! Check for another employee');
        ELSE
            DBMS_OUTPUT.PUT_LINE(i.ename||'was hired on'||i.hiredate);
        END IF;
    END LOOP;
END;
/

*
ERROR at line 1:
ORA-20005: Hiredate earlier than the given date!! Check for another
employee
ORA-06512: at line 11
```

In the preceding example, note that the exception name is not used to create the exception handler. Just after the exception is raised through `RAISE_APPLICATION_ERROR`, the program is terminated.

If you wish to have a specific exception handler for the exceptions raised through `RAISE_APPLICATION_ERROR`, you must declare the exception in the declarative section and associate the error number using `PRAGMA EXCEPTION_INIT`. Check the following PL/SQL program:

```

/*Enable the SERVEROUTPUT parameter to print the results in the
environment*/
SQL> SET SERVEROUTPUT ON

/*Start the PL/SQL block */
DECLARE

/*Declare the birth date */
    l_dob_mon DATE := '01-DEC-1981';

/*Declare the exception variable */
    INVALID_EMP_DATES EXCEPTION;
    PRAGMA EXCEPTION_INIT(INVALID_EMP_DATES,-20005);

/*Declare a cursor to filter employees who were hired on birthday
month*/
    CURSOR C IS
        SELECT ename, deptno, hiredate
        FROM emp;
BEGIN
    FOR I IN C
    LOOP
        /*Raise exception, if birthdate is later than the hiredate */
        IF i.hiredate < l_dob_mon THEN
            RAISE INVALID_EMP_DATES;
        ELSE
            DBMS_OUTPUT.PUT_LINE(i.ename||'was hired on'||i.hiredate);
        END IF;
    END LOOP;
EXCEPTION
    WHEN INVALID_EMP_DATES THEN
        DBMS_OUTPUT.PUT_LINE(SQLERRM||'Hiredate earlier than the given
date!! Check for another employee');

```

```
END;  
/
```

```
ORA-20005: Hiredate earlier than the given date!! Check for another  
employee
```

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

Exception propagation

Until now, we have seen that, as soon as the exception is raised in the procedural section of a PL/SQL block, the control jumps to the exception section and chooses the appropriate exception handler. The non-existence of the exception handler may lead to the abnormal termination of the program.

In the case of nested PL/SQL blocks, if the exception is raised in an inner block, the program control flows down to the exception section of the inner block. If the inner block handles the exception, it is executed and the program control returns to the next executable statement in the outer block.

If the inner block does not handle the exception, the program control continues to search for the appropriate handler and propagates to the exception section of the outer block. Yes, the execution of the outer block is skipped and the program control lands straight in to the exception section. The program control will continue to propagate the unhandled exception in the outer blocks until the appropriate one is found and handled.

For example, the following PL/SQL program contains a child block within the parent block:

```
/*Parent block*/  
DECLARE  
...  
BEGIN  
    /*Outer block executable statements*/  
    ...  
    /*Child Block*/  
    DECLARE  
    ...  
    BEGIN  
        ...  
        /*Inner block executable statements*/  
        ...
```

```
EXCEPTION
    /*Inner block exception handlers*/
END;
...
/*Outer block executable statements*/
EXCEPTION
/*Outer block exception handlers*/
END;
```

If the exception is raised in one of the `/*Inner block executable statements*/`, the control flows to `/*Inner block exception handlers*/`. If the appropriate exception handler is not found, it propagates straight to the `/*Outer block exception handlers*/` and execution of `/*Outer block executable statements*/` is skipped.

When working with nested PL/SQL blocks, developers must be cautious while coding exception handling logic. The exception propagation should be thoroughly tested to build fail-proof applications.

Creating stored procedures

A **procedure** is a derivative of a PL/SQL block that has a name and is stored persistently within the database. It is the schema object that is primarily used to implement business logic on the server side. A procedure promotes a modular programming technique by breaking down complex logic into simple routines.

The key features of stored procedures are:

- A procedure must be invoked from the executable section of a PL/SQL block as a procedural statement. You can also execute it directly from SQLPLUS using the `EXECUTE` statement. Note that a procedure can not be called from a `SELECT` statement.
- A procedure can optionally accept parameters in `IN`, `OUT`, or `IN OUT` mode.
- A procedure cannot return a value. The only way for a procedure to return a value is through `OUT` parameters, but not through the `RETURN [value]` statement. The `RETURN` statement in a procedure is used to skip the further execution of the program and exit control.

The following table differentiates between the IN, OUT, and IN OUT parameters:

IN	OUT	IN OUT
Default parameter mode	Has to be explicitly defined	Has to be explicitly defined
Parameter's value is passed to the program from the calling environment	Parameter returns a value back to the calling environment	Parameter may pass a value from the calling environment to the program or return value to the calling environment
Parameters are passed by reference	Parameters are passed by value	Parameters are passed by value
May be a constant, literal, or initialized variable	Uninitialized variable	Initialized variable
Can hold default value	Default value cannot be assigned	Default value cannot be assigned

The syntax for a procedure is as follows:

```
CREATE [OR REPLACE] PROCEDURE [Procedure Name] [Parameter List]
[AUTHID DEFINER | CURRENT_USER]
IS
    [Declaration Statements]
BEGIN
    [Executable Statements]
EXCEPTION
    [Exception handlers]
END [Procedure Name];
```

The following standalone procedure converts the case of the input string from lower case to upper case:

```
/*Create a procedure to change case of a string */
CREATE OR REPLACE PROCEDURE P_TO_UPPER (P_STR VARCHAR2)
IS
/*Declare the local variables*/
    L_STR VARCHAR2(50);
BEGIN
/*Convert the case using UPPER function*/
    L_STR := UPPER(P_STR);
/*Display the output with appropriate message*/
    DBMS_OUTPUT.PUT_LINE('Input string in Upper case : '||L_STR);
END;
/
```

Procedure created.

Executing a procedure

A procedure can either be executed from SQL*Plus or a PL/SQL block. The P_TO_UPPER procedure can be executed from SQL*Plus.

The following code shows the execution of the procedure from SQL*Plus (note that the parameter is passed using bind variable):

```
/*Enable the SERVEROUTPUT parameter to print the results in the
environment*/
```

```
SQL> SET SERVEROUTPUT ON
```

```
/*Declare a session variable for the input*/
```

```
SQL> VARIABLE M_STR VARCHAR2(50);
```

```
/*Assign a test value to the session variable*/
```

```
SQL> EXECUTE :M_STR := 'My first PLSQL procedure';
```

PL/SQL procedure successfully completed.

```
/*Call the procedure P_TO_UPPER*/
```

```
SQL> EXECUTE P_TO_UPPER(:M_STR);
```

Input string in Upper case : MY FIRST PLSQL PROCEDURE

PL/SQL procedure successfully completed.

The P_TO_UPPER procedure can be called as a procedural statement within an anonymous PL/SQL block:

```
/*Enable the SERVEROUTPUT parameter to print the results in the
environment*/
```

```
SQL> SET SERVEROUTPUT ON
```

```
/*Start a PL/SQL block*/
```

```
SQL> BEGIN
```

```
    /*Call the P_TO_UPPER procedure*/
```

```
        P_TO_UPPER ('My first PLSQL procedure');
```

```
    END;
```

```
    /
```

Input string in Upper case : MY FIRST PLSQL PROCEDURE

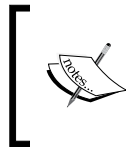
PL/SQL procedure successfully completed.

Functions

Similar to a stored procedure, a **function** is a named derivative of a PL/SQL block that is physically stored within the Oracle database schema.

The key features of stored functions are as follows:

- A function can accept parameters in all three modes (IN, OUT, and IN OUT) and mandatorily returns a value.
- Functions can be called in SQL statements (SELECT and DMLs). Such functions must accept only IN parameters of valid SQL types. Alternatively, a function can also be invoked from SELECT statements if the function body obeys the database purity rules.
- If the function is called from an SQL statement, its return type should be a valid SQL data type. If the function is invoked from PL/SQL, the return type should be a valid PL/SQL type.



Starting from Oracle Database 12c, PL/SQL—only data types can cross the PL/SQL to SQL interface. A PL/SQL anonymous block can invoke a PL/SQL subprogram with parameters of BOOLEAN or a packaged collection type.

The syntax for a function is as follows:

```
CREATE [OR REPLACE] FUNCTION [Function Name] [Parameter List]
RETURN [Data type]
[AUTHID DEFINER | CURRENT_USER]
[DETERMINISTIC | PARALLEL_ENABLED | PIPELINED]
[RESULT_CACHE [RELIES_ON (table name)]]
IS
    [Declaration Statements]
BEGIN
    [Executable Statements]
    RETURN [Value]
EXCEPTION
    [Exception handlers]
END [Function Name];
```

Let us create a standalone function, F_GET_DOUBLE, which accepts a numeric parameter and returns its double:

```
/*Create the function F_GET_DOUBLE*/
CREATE OR REPLACE FUNCTION F_GET_DOUBLE (P_NUM NUMBER)
RETURN NUMBER    /*Specify the return data type*/
```

```
IS

/*Declare the local variable*/
  L_NUM NUMBER;
BEGIN

/*Calculate the double of the given number*/
  L_NUM := P_NUM * 2;

/*Return the calculated value*/
  RETURN L_NUM;
END;
/

Function created.
```

Functions – execution methods

Functions can either be called from a SQL*Plus environment or invoked from a PL/SQL program as a procedural statement.

The function F_GET_DOUBLE can be executed in the SQL* Plus command prompt as follows. As the function returns an output, you must declare a session variable and capture the function result in the variable.

```
/*Enable the SERVEROUTPUT parameter to print the results in the
environment*/
SET SERVEROUTPUT ON

/*Declare a session variable M_NUM to hold the function output*/
VARIABLE M_NUM NUMBER;

/*Function is executed and output is assigned to the session
variable*/
EXECUTE :M_NUM := F_GET_DOUBLE(10);

PL/SQL procedure successfully completed.

/*Print the session variable M_NUM*/
PRINT M_NUM

M_NUM
-----
20
```


The F_GET_DOUBLE function can be called from an anonymous block or a standalone subprogram.

```
/*Enable the SERVEROUTPUT parameter to print the results in the
environment*/
SET SERVEROUTPUT ON

DECLARE
    M_NUM NUMBER;
BEGIN
    M_NUM := F_GET_DOUBLE(10);
    DBMS_OUTPUT.PUT_LINE('Doubled the input value as : '||M_NUM);
END;
/
Doubled the input value as : 20

PL/SQL procedure successfully completed.
```

Restrictions on calling functions from SQL expressions

Unlike procedures, a stored function can be called from a SELECT statement, provided it does not violate the database purity levels. The rules are as follows:

- A function called from a SELECT statement cannot contain DML statements
- A function called from an UPDATE or DELETE statement on a table cannot query (SELECT) or perform transactions (DMLs) on the same table
- A function called from an SQL expression cannot contain TCL (COMMIT or ROLLBACK) commands or DDL (CREATE or ALTER) commands

The F_GET_DOUBLE function can easily be embedded within a SELECT statement as it respects all the preceding rules:

```
/*Invoke the function F_GET_DOUBLE from SELECT statement*/
SQL> SELECT F_GET_DOUBLE(10) FROM DUAL;

F_GET_DOUBLE(10)
-----
                20
```



In the Oracle Database, DUAL is a table owned by the SYS user, which has a single row and a single column, DUMMY, of VARCHAR2 (1) type. It was first designed by Charles Weiss while working with internal views to duplicate a row. The DUAL table is created by default during the creation of the data dictionary with a single row whose value is X. All database users, other than SYS, use its public synonym to select the value of pseudo columns such as USER, SYSDATE, NEXTVAL, or CURRVAL. Oracle 10g considerably improved the performance implications of the DUAL table through a fast dual-access mechanism.

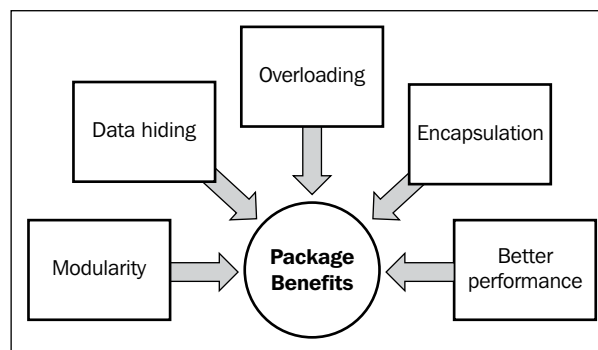
A PL/SQL package

A PL/SQL package encapsulates multiple PL/SQL constructs under a single unit. The PL/SQL constructs can be subprograms, cursors, variables, and exceptions. As a schema object, a PL/SQL package demonstrates the principles of logic hiding, encapsulation, and subprogram overloading.



Standalone subprograms cannot be overloaded. Only packaged subprograms can be overloaded by their signatures.

The following diagram shows the advantages of a package:




A package has two components – the package specification and package body. While the package specification contains the prototype of public constructs, the package body contains the definition of public as well as private (local) constructs.

The characteristics of the package specification are as follows:

- It is the mandatory component of the package. A package cannot exist without its specification.

- It contains the prototypes of public constructs. The prototype is a forward declaration of the constructs that includes the declaration, header specification and signature information terminated by a semicolon. The subprograms constructs, once prototyped, should be defined in the package body section. The package specification cannot contain an executable section.
- These member constructs are visible within and outside the package. They can be invoked from outside the package by the privileged users.

 The public constructs of a package are accessed as
[PACKAGE NAME] . [CONSTRUCT].

- Valid package constructs can be PL/SQL types, variables, exceptions, procedures, and functions.
- If the package specification contains variables, they are implicitly initialized to NULL by Oracle

The characteristics of the package body are as follows:

- The package body contains the definition of the subprograms that were declared in the package specification.
- The package body can optionally contain local constructs. The accessibility of the local constructs is limited to the package body only.
- The package body is an optional component; a package can exist in a database schema without its package body.

The syntax for creating a package is as follows:

```
CREATE [OR REPLACE] PACKAGE [NAME] IS
    [PRAGMA]
    [PUBLIC CONSTRUCTS]
END;

CREATE [OR REPLACE] PACKAGE BODY [NAME] IS
    [LOCAL CONSTRUCTS]
    [SUBPROGRAM DEFINITION]
    [BEGIN]
END;
```

Note the optional BEGIN section in the package body. It is optional, but gets executed only the first time the package is referenced. It is used to initialize global variables.

A package can be compiled with its specification component alone. In such cases, packaged program units cannot be invoked as their executable logic has not been defined yet.

The compilation of a package with a specification and body ensures the concurrency between the program units prototyped in the specification and the program units defined in the package body. All packaged program units are compiled in a single package compilation. If a package is compiled with errors, it is created as an invalid object in the database schema. You can query the `STATUS` column to check the current status of an object in the `USER_OBJECTS`, `ALL_OBJECTS`, or `DBA_OBJECTS` dictionary views.

Oracle Database 12c enhancements to PL/SQL subprograms

Oracle Database Release 12c includes a number of PL/SQL feature enhancements. These enhancements are focused on improving the usability of PL/SQL as a language. Although the next chapter will discuss many more new features in detail, it is worthwhile to mention a few of them that are exclusively related to Oracle PL/SQL subprograms.

- **Defining PL/SQL subprograms in the `SELECT` statement:** Although PL/SQL allows the invoking of a function from the `SELECT` statement, the context switch from SQL to the PL/SQL engine degraded the performance. Oracle 12c allows creating PL/SQL units in the `WITH` clause of a subquery and using it in the `SELECT` statement. The new approach to calling functions in SQL statements enhances the performance as there is no context switching across the engines. In addition, these functions are not stored in the database schema.
- **Granting roles to program units:** One of the challenges in PL/SQL before Oracle 12c was that a program unit had to be created with definers rights, if it was intended to be executed by all users. A user with a lower set of privileges could perform the unauthorized changes. With Oracle 12c, granting roles to PL/SQL program units adds a levels of safety. You can now create program units with invoker's rights and control the privileges, which are required to run the program, through a role.
- **Protecting PL/SQL unit access through the `ACCESSIBLE BY` clause:** With Oracle 12c, you can restrict access to a PL/SQL unit by unauthorized programs. A subprogram (a procedure, function or a package) can optionally include an `ACCESSIBLE BY` clause to define a **white list** of PL/SQL program units that can invoke it.

Managing database dependencies

PL/SQL program units, as well as other database objects such as views, may refer to other database objects in their procedural section. The calling program unit is said to be dependent on the called program units (known as referenced objects). If EMP and DEPT are the base tables used in creating a view V_EMP_REP, then the view is dependent on EMP and DEPT.



A sequence can always be a referenced object. A package body is always a dependent object.

Database dependency can be classified as **direct** or **indirect**. Consider three objects – P, M, and N. If object P references object M and object M references object N, then P is directly dependent on M and indirectly dependent on N.

Displaying the direct and indirect dependencies

The dependency matrix is automatically generated and maintained within the Oracle Database. The status of an object is the basis of dependency among the objects. The status of an object can be queried from the USER_OBJECTS (or ALL_OBJECTS or DBA_OBJECTS) dictionary view. The following query queries the status of the function F_GET_DOUBLE:

```
/*Check the status of the function F_GET_DOUBLE*/
SELECT status
FROM user_objects
WHERE object_name='F_GET_DOUBLE'
/

STATUS
-----
VALID
```

The system views DEPTREE and IDEPTREE capture the necessary information about the direct and indirect dependencies. Database administrators can create the views by running the script \$ORACLE_HOME\RDBMS\ADMIN\utldtree.sql.

The execution steps for the script are as follows:

1. Login as SYSDBA in SQL Developer or SQL*Plus.
2. Copy the complete path and script name (prefixed with @).

3. Execute the script (with *F9*).
4. Query the `DEPTREE` and `IDEPTREE` views to verify their creation.

The script creates the `DEPTREE_TEMPTAB` table and the `DEPTREE_FILL` procedure. The `DEPTREE_FILL` procedure can be executed to populate the dependency details of an object.

```
/*Populate the dependency matrix for the function F_GET_DOUBLE*/
SQL> EXEC DEPTREE_FILL('FUNCTION','SCOTT','F_GET_DOUBLE');
```

PL/SQL procedure successfully completed.

Note that the first parameter of the `DEPTREE_FILL` procedure is the object type, the second is the owner, and the third is the object name.

The `DEPTREE` and `IDEPTREE` views can now be queried to view the dependency information.

Dependency metadata

Oracle provides the data dictionary views (`USER_DEPENDENCIES`, `ALL_DEPENDENCIES`, and `DBA_DEPENDENCIES`) to view the complete dependency metrics shared by an object. Besides the dependent object's list, it also lists its referencing object name and owner.

The following screenshot shows the structure of the dictionary view `DBA_DEPENDENCIES`:

```
SQL> DESC DBA_DEPENDENCIES
```

Name	Null?	Type
OWNER	NOT NULL	VARCHAR2(30)
NAME	NOT NULL	VARCHAR2(30)
TYPE		VARCHAR2(18)
REFERENCED_OWNER		VARCHAR2(30)
REFERENCED_NAME		VARCHAR2(64)
REFERENCED_TYPE		VARCHAR2(18)
REFERENCED_LINK_NAME		VARCHAR2(128)
DEPENDENCY_TYPE		VARCHAR2(4)

Dependency issues and enhancements

In line with the conventional dependency phenomenon, the status validity of the dependent object depends upon the status of the referenced object. So, if the definition of the referenced object is altered, the dependent object is marked `INVALID` in the `USER_OBJECTS` view. Although object recompilation can easily solve the problem, the object invalidations may impact the application flow.

Oracle 11g introduced **Fine Grained Dependency Tracking (FGD)** to modify the dependency principle as follows. If the alteration in the referenced object does not affect the dependent object, the dependent object will remain in the VALID state. For instance, if a view is created with a fixed set of columns of a table and the table is altered to add a new column, the view will remain in a VALID state.

Reviewing Oracle-supplied packages

Oracle-supplied packages exist as prebuilt programs in the database as wrapper code. These packages not only help database developers work on extended functionalities but also reduce writing extensive and complex code. The use of the Oracle-supplied API is always recommended as it improves code standardization.

The scripts for these packages are available in the `$ORACLE_HOME\RDBMS\ADMIN\` folder. All packages reside on the database server. Public synonyms are available (or can be created too) for these packages so that the packages are accessible to the database users. Oracle 12c adds multiple packages to the Oracle-supplied PL/SQL package list. The latest additions provide PL/SQL interfaces for the new functionalities that have been added to the database release.

Some of the important Oracle-supplied packages are listed as follows:

- `DBMS_ALERT`: This package is used for the notification of database events.
- `DBMS_LOCK`: This package is used for managing lock operations (lock, conversion, and release) in PL/SQL applications.
- `DBMS_SESSION`: This package is used to set session level preferences from PL/SQL programs (similar to `ALTER SESSION`).
- `DBMS_OUTPUT`: This package is one of the most frequently used built-ins for buffering data messages and displaying debug information.
- `DBMS_HTTP`: This package is used for HTTP callouts.
- `UTL_FILE`: This package is used for reading, writing, and performing other file operations on the server.
- `UTL_MAIL`: This package is used to compose and send mails.
- `DBMS_SCHEDULER`: This package is used for scheduling execution of stored procedures at a given time.
- `DBMS_PARALLEL_EXECUTE`: This package can be used to execute a user-defined task in parallel. If the PL/SQL block is running a large update on a table, the package can be used to enable the parallel execution of the task by splitting it into chunks.

- **DBMS_PRIVILEGE_CAPTURE:** This package is introduced in Oracle Database 12c to set a policy in order to capture the usage of privileges (object and system) in respect to users. It helps in controlling excess privileges for users.
- **DBMS_REDACT:** This package is introduced in Oracle Database 12c to create redaction policies, in order to mask data based on user authorization.
- **DBMS_RESOURCE_MANAGER:** This package is used to create consumer groups, directives, and resource manager plans for containers as well as a pluggable database. The resource manager plan determines the resources allocated to a pluggable database, a schema, or a task.
- **DBMS_DATAPUMP:** This package is used to move data, metadata, or both from one database to another. The source and target databases can be on different platforms.
- **DBMS_PDB:** This package is introduced in Oracle Database 12c to generate or analyze the integration properties of a pluggable database for unplug/plug operations.
- **DBMS_SQL:** This package enables dynamic SQL in PL/SQL. You can run DML or DDL statements using `DBMS_SQL` from a PL/SQL block.
- **DBMS_REDEFINITION:** This package is used to perform online redefinition tasks for tables.
- **DBMS_UTILITY:** This package is used to accomplish many utility operations such as analyze object, compile schema, get dependency, resolve a given name, validate database objects, format call and error stack, expand SQL text, and retrieve current database version.

Based on the objective to be achieved, the packages can be categorized as follows:

- **Standard application development:** Many of the DBMS packages provide an application interface for database features. For example, `DBMS_REDACT` provides an interface to create and manage redaction policies. Similarly, `DBMS_UTILITY` provides subprograms to retrieve instance or database information, call stacks and error stacks, and analyze and validate objects. The `DBMS_OUTPUT` package is one of the packages most frequently used to display text messages. It can be efficiently used for tracing and debugging purposes. Accessing and writing operating system files was made possible through `UTL_FILE`.
- **General usage and application administration:** Oracle has many packages for monitoring applications and users. Statistics generation, load history, and space management are the key objectives accomplished by these packages. `DBMS_UTILITY` comprises subprograms for general usage.
- **Internal support packages:** Oracle maintains these packages for its own use.

- **Transaction processing packages:** Oracle provides utility packages that enable the monitoring of transaction stages. Though they are rarely used, they can efficiently ensure transparent and smooth transactions. For example, `DBMS_TRANSACTION` is used to access SQL transactions from stored subprograms. `DBMS_PARALLEL_EXECUTE` executes a large update in parallel by splitting it into small tasks.

Oracle SQL Developer

Oracle SQL Developer is a **Graphical User Interface (GUI)** integrated development environment from Oracle Corporation. Oracle SQL Developer is a free tool that simplifies PL/SQL code development, the database design, and modeling and administration for standalone and cloud deployments. This tool enhances the user's experience by maximizing productivity and extensibility.

SQL Developer is a Java-based cross-platform tool that can run on Linux, Windows, and Mac OS X.

Oracle SQL Developer supports Oracle Database versions 10g, 11g, and 12c. With Oracle Database 12c Multitenant architecture, SQL Developer extends full support for various multitenant operations. In addition to the support for Oracle Databases, the tool also allows users to connect to non-Oracle Databases including MySQL, Microsoft SQL Server, Microsoft Access, Sybase, and Teradata. The extensible framework enables users to develop custom extensions in order to address specific requirements.

SQL Developer was first released in March, 2006. The initial release of SQL Developer included basic features such as a schema browser, an SQL worksheet to run SQL and invoke SQL scripts, a PL/SQL editor, code debugging, and running basic reports. Since then, the tool has grown immensely and matured over the years. At the time of writing, the latest release of SQL Developer is 4.1.

Oracle SQL Developer for DBA, Developers, and Application Architects

Oracle SQL Developer provides powerful features and interfaces for code development, administrative activities, and data modeling. It offers rich editors for developers who work with SQL, PL/SQL, and stored program units. SQL Developer can run SQL queries, monitor performance through execution plans and SQL tuning, and prepare scripts. Database developers can build PL/SQL applications, debug them, and perform run-time testing.

With SQL Developer 3.0, the DBA panel in the tool added the functionality to perform common DBA tasks and activities. The new feature additions such as the data miner, data modeler, database navigator, and DBMS scheduler provide a range of administrative functionalities in the tool. Database administrators can perform core administration tasks such as export/import through data pump, **RMAN**, user and role management, and resource management. SQL Developer integrates seamlessly with Oracle APEX, thereby allowing **APEX** developers to browse, deploy, and export applications.

With the most recent version, more DBA activities have been incorporated to make it more feature-rich and complete.

For database architects, SQL Developer offers a data modeling solution with **SQL Developer Data Modeler (SDDM)**. The SDDM enables architects to create data flow diagrams, design versioning via subversion, import designer repositories, and most importantly perform logical, relational, and physical data modeling.

SQL Developer 4.0

With the release of Oracle SQL Developer 4.0 and onwards, the tool follows Oracle's data management strategy by supporting Oracle Database 12c Multitenant option, cloud deployments, Oracle NoSQL and JSON. In addition, it includes a brand new command line interface utility to enhance user experience. With native support for Oracle Rest Data Services in Oracle Database 12c, Web application developers can work with SQL Developer to create and alter services. Database administrators can now run ASH, AWR, and ADDM reports from the performance page in the DBA panel. SQL Developer 4.0's new features can be summarized as follows:

- Support for Oracle Database 12c Multitenant architecture
- Support for Oracle NoSQL Database
- Support for database products such as **TimesTen**, **Data Miner**, **XML DB**, and Spatial and Graphs
- Support for Java 7
- Querying JSON data in relational format
- New instance viewer enables the monitoring of wait events, storage, log switches, and database processes

Chapter 12, Working with Oracle SQL Developer, focuses on various features of the Oracle SQL Developer tool.

Summary

Over the years, PL/SQL has matured a great deal and has produced a vast library of objects, features, and standards. This chapter focused on giving a quick summary of PL/SQL programming. It assumed readers were familiar with database programming concepts and provided enough substance to get them ready for the forthcoming chapters. It would have been a tough call to build a glossary of PL/SQL objects in a single chapter.

We started with an overview of PL/SQL fundamentals, block structure, and exception handling. Additionally, this chapter threw light on cursor handling in PL/SQL, the `CURSOR FOR` loop, and schema objects such as procedures, functions, and packages. In forthcoming chapters, we will focus on the key faculties of the PL/SQL language as well as Oracle Database 12c features.

Practice exercise

- Which of the following features are not available in SQL Developer?
 1. Query builder.
 2. Database export and import.
 3. Database backup and recovery functions.
 4. Code Subversion repository.
- For a function to be called from a SQL expression, which of the following conditions should it obey?
 1. A function in the `SELECT` statement should not contain DML statements.
 2. The function should return a value.
 3. A function in the `UPDATE` or `DELETE` statement should not query the same table.
 4. A function called from a SQL expression cannot contain TCL (`COMMIT` or `ROLLBACK`) commands or DDL (`CREATE` or `ALTER`) commands.
- The following query is executed in the SCOTT schema:

```
SELECT NAME, referenced_owner, referenced_name
FROM all_dependencies
WHERE owner = USER
AND referenced_type IN ('TABLE', 'VIEW')
AND referenced_owner IN ('SYS')
ORDER BY owner, NAME, referenced_owner, referenced_name;
```

Which statement is true about the output of this query?

1. It displays the schema objects, created by the user ORADEV, that use a table or view owned by SYS.
 2. An exception occurs as user SCOTT has insufficient privileges to access ALL_DEPENDENCIES view.
 3. It displays all PL/SQL code objects that reference a table or view directly for all the users in the database.
 4. It displays only those PL/SQL code objects created by the user OE that reference a table or view created by the user SYS.
- Which of the following is true about PL/SQL blocks?
 1. Exception is a mandatory section without which an anonymous PL/SQL block fails to compile.
 2. Bind variables cannot be referred inside a PL/SQL block.
 3. The scope and visibility of the variables declared in the declarative section of the block are within the current block only.
 4. The RAISE_APPLICATION_ERROR procedure maps a predefined error message to a customized error code.
 - From the following options, identify the ways of defining exceptions:
 1. Declare an EXCEPTION variable and raise it using the RAISE statement.
 2. Use PRAGMA EXCEPTION_INIT to associate a customized exception message to a pre-defined oracle error number.
 3. Declare an EXCEPTION variable and use it in RAISE_APPLICATION_ERROR.
 4. Use RAISE_APPLICATION_ERROR to create a dynamic exception at any stage within the executable or exception section of a PL/SQL block.
 - Choose the differences between procedures and functions:
 1. A function must mandatorily return a value, while a procedure may or may not.
 2. A function can be called from a SQL query, while a procedure can never be invoked from SQL.
 3. A function can accept parameters passed by a value, while a procedure can accept parameters passed by reference only.
 4. A standalone function can be overloaded but a procedure cannot.

- Examine the values of the cursor attribute for the following query and pick the attribute with the wrong value:

```
BEGIN
...
SELECT ENAME, SAL
  INTO L_ENAME, L_SAL
  FROM EMPLOYEES
 WHERE EMPID = 7900;
...
END;
```

1. SQL%ROWCOUNT = 1
2. SQL%ISOPEN = FALSE
3. SQL%FOUND = FALSE
4. SQL%NOTFOUND = FALSE

2

Oracle 12c SQL and PL/SQL New Features

Oracle released Oracle Database 12c in July 2013. From a technology standpoint, it was an important product release as the focus was consolidation of databases on public and private cloud infrastructures. Oracle Database 12c introduces the new **Multitenant architecture** that allows multiple databases to run as a tenant within a single database. The new design assures tenant isolation and security, and enhances manageability of consolidated databases, Oracle Database 12c provides more than 500 new features including multitenant architecture and many others related to security, high availability, and performance. The chapter covers the key features introduced in the latest product release.

Oracle Database 12c has made considerable improvements to the **SQL** and **PL/SQL** languages. The language enhancements in Oracle Database 12c focus on support for ANSI SQL standards, effortless code writing and migration from non-Oracle to Oracle compliant code. This chapter discusses many such enhancements and features of Oracle Database 12c.

The outline of the chapter is as follows:

- The Multitenant architecture
- SQL new features
- PL/SQL new features
- Oracle Database 12c In-Memory option

Database consolidation and the new Multitenant architecture

Consolidation is the key enabler for moving databases to on-cloud models. An efficient consolidation strategy can provide elastic sharing of resources and maximize resource utilization in a consolidated stack. A database hosted on a public or private cloud must guarantee tenant isolation and security. In addition to the elementary requirements of cloud deployments, database provisioning on the cloud should be quick and easy.

Oracle Database 12c introduces a multitenant architecture that meets the challenges of cloud deployments. The new tenant-based architecture allows one or more application databases (known as **pluggable databases**) to run within a single database (known as the container database). Each pluggable database is completely shielded from all other pluggable databases running within the same container database. The architecture provides a unique mechanism to segregate the system and application metadata at the container and pluggable level respectively. In a single-tenant architecture, a root container can have just one pluggable database. In a multitenant architecture, a root container can have more than one pluggable database. While single-tenant is free of cost and available in Oracle Database Standard Edition, multitenant can be licensed in Oracle Database Enterprise Edition only.

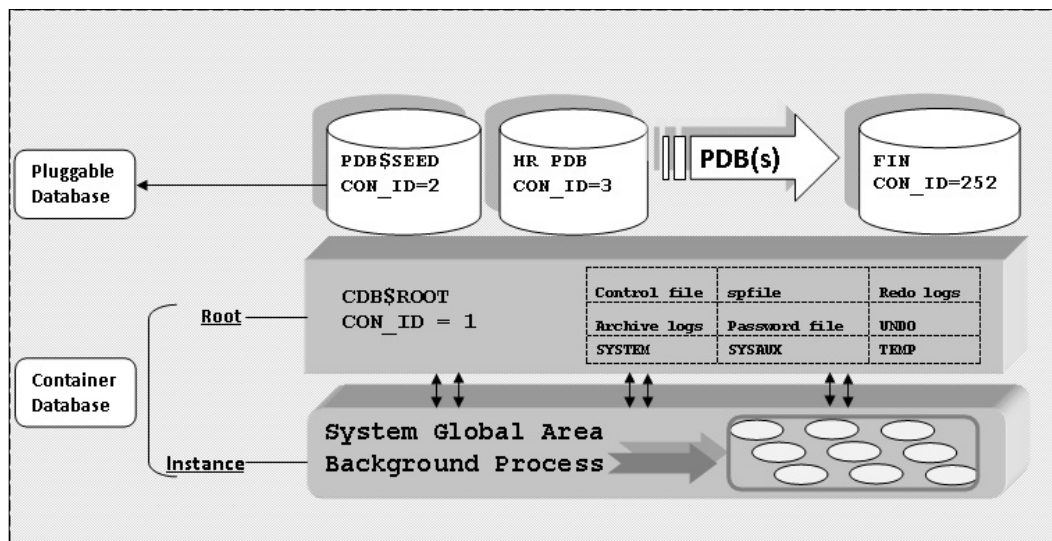
Starting with Oracle Database 12.1.0.2, the non-tenant or standalone architecture of Oracle Database is deprecated which means that it will not be further enhanced.

Let us get familiarized with the terms of the new world – the container and pluggable databases.

An Oracle 12c **container database (CDB)** provides the instance that can be shared by multiple databases. A CDB instance has the memory and a set of background processes. From the server subsystem, the container database is the only database visible.

A **pluggable database (PDB)** is similar to a pre 12c database that serves as an application database backend. A pluggable database contains the application tablespaces.

The following diagram shows the Oracle Database 12c multitenant architecture:



A container database may contain one or more pluggable database containers. With reference to the preceding architecture, let us drill into the specifics of new components.

A container database (CDB) consists of one root container (known as **CDB\$ROOT**), one seed pluggable database (known as **PDB\$SEED**), and multiple pluggable databases. Similar to the previous releases, a container database has an instance and set of files. The database instance is of the container, which means that the **System Global Area (SGA)** is common for all the pluggable databases. Also, there is a single copy of background processes at the container instance level only, and not replicated for each PDB.

A pluggable database is a database that stores the application data. As of Oracle Database 12c Release 1, a multitenant database can have a maximum of 252 pluggable databases. A PDB service is created at the time of provisioning that runs within the CDB service and gets auto-registered with the CDB listener. Since a PDB runs as a secured service within the container database administration service, it cannot be authenticated by the server operating system. However, you can create a user-defined service using **DEMS_SERVICE** for OS authentication and TNS connection.

The **redo** logs and **archive** logs are at the container level. Every time a pluggable database has to make a redo entry, the request is tagged with the PDB identity. The identity in this case is nothing but the container identifier (`CON_ID`). Each container in a container database is assigned a unique container id. The root has the container id 1; the seed PDB has 2 and each pluggable database is assigned a container id in a sequential fashion. The container id is quite a significant element as the common redo is logically virtualized by annotating each and every entry with the container id. All the dictionary views and metadata views have an additional column `CON_ID` to identify the container to which the information belongs.

The control file and server parameter file are at the root container level. Starting with Oracle 12c, there are two categories of parameters – CDB modifiable and PDB modifiable. The `V$PARAMETER` dictionary view has an additional column `ISPDB_MODIFIABLE`. The value of the column is `N` for CDB-modifiable and `Y` for PDB-modifiable parameters.

The `SYSTEM` and `SYSAUX` tablespaces at the root container level store the system metadata that is specific just to the Oracle Database. The application metadata for each pluggable database is stored in each PDB's respective `SYSTEM` and `SYSAUX` tablespaces. The `UNDO` tablespace is at the root container level. Once again, similar to redo, each undo entry is tagged with the container identifier or `CON_ID`. With one undo for the entire container, you might be interested in evaluating the performance implications of the system. Well, undo management in multitenant consolidation is not too different from schema consolidation. The redo and undo entries are indexed by the container ids, thus speeding up the concurrent record access for a specific PDB.

Although the multitenant architecture shows the `TEMP` tablespace at the container level, but it can be created for each of the pluggable database too. Users can create tablespaces within the pluggable databases as usual, whenever required.

The Oracle Database 12c Multitenant architecture – features

Having mentioned the challenges of consolidation and cloud deployments, let us take a look at the capabilities enabled by the multitenant architecture.

Multitenant for Consolidation

In the past, enterprises have been following multiple approaches to consolidate databases to achieve tenant isolation and manageability. You could have an enormous physical server and then extract virtualized homes for each application. The virtualized home is the application's world of operation and shares the server subsystem. However the overhead of managing the heterogeneous pieces can be a potential pain point. The consolidation density is limited by the fact that the memory allocated to a virtualized home remains intact whether or not it is getting used.

You can create multiple databases on a server for each application. An aggregation of databases is possible until the server memory is exhausted. Keep in mind that each database instance creates its own copy of background processes, which holds the CPU cycles, thus reducing the consolidation density.

One of the most efficient approaches before Oracle 12c was to adopt schema-based consolidation. You can achieve the highest consolidation density as multiple schemas are part of a single database. The downside of schema-based consolidation is that tenant isolation was not guaranteed, while security and manageability were a big concern.

The multitenant approach in Oracle Database 12c allows the effective sharing of server resources, the operating system, and even the database. The advanced sharing of resources makes it suitable to implement and deploy on the cloud. It guarantees the tenant isolation as each pluggable database appears remote to other pluggable databases sharing the same container. A common SGA and a single set of background processes optimally utilize the server memory, thus maximizing consolidation. Management capabilities add value to the new architecture in Oracle 12c.

Plug/unplug

A pluggable database can be unplugged from the current container and plugged into another compatible container. Data mobility across the containers becomes easier and quicker as you just have to work with the PDB metadata and not move the application data. The unplug operation captures the PDB lineage in an XML manifest file, allowing a PDB to be plugged into another container database.

The feature comes in handy in data center operations when the databases are expected to move quickly to different service levels, without impacting business continuity. Another very important use case of the plug/unplug feature is when you are required to upgrade or patch subsets of PDBs in a container. You can unplug a PDB from a 12.1 container and plug it in a Oracle 12.x container database.

Manage Many as One

The multitenant architecture enables the "manage many as one" capability. Organizations who are adopting a Multitenant architecture, are expected to see a significant reduction in operational expenses by managing multiple databases as one. Here is the list of multitenant operations that apply to all the pluggable databases:

- Backing up the container database backs up the root container including all the pluggable databases. However, point in time recovery at the pluggable database level is possible.



You can also backup just the CDB\$ROOT or a particular PDB

- In a data guard setup, all pluggable databases are auto-discovered at the standby site. Therefore, a high availability of all PDBs is maintained by implementing the data guard at the container level.



In Oracle Database 12.1.0.2, you can provision a PDB on a primary site but disable its recovery at the standby site by specifying STANDBYS=NONE

- Upgrading and patching the container database upgrades or patches all the pluggable databases. However, if you are required to upgrade or patch a subset of pluggable databases, you can unplug and plug into a different container of a higher release or patch set.

Rapid provisioning

Pluggable databases can be quickly provisioned either locally within a container or from a remote container. PDB provisioning doesn't involve copying the system metadata or the creation of background processes, thereby speeding up the creation process.

A pluggable database can be provisioned in different ways as listed here:

- A fresh PDB from seed (PDB\$SEED) — creating a brand new pluggable database from the seed PDB is a mere copy of `SYSTEM` and `SYSAUX` files from the seed location to the target PDB location on the database server.
- Clone/copy an existing PDB — an existing PDB can be cloned locally within the current container. Operationally, cloning a PDB is copying the files to a new PDB location. The PDB remote cloning feature is available from Oracle Database 12c (12.1.0.2).

- Snapshot Cloning – Oracle Database 12c supports snapshot cloning of pluggable databases on copy-on-write file systems. By virtue of the copy-on-write feature, snapshot cloning is an extremely fast method of creating copies of pluggable databases.

Oracle Database 12.1.0.2 introduced new enhancements to PDB cloning in a multitenant container database. The enhancements are briefly described here:

- Schema consolidation to PDB-based consolidation – you can create a new pluggable database by specifying just the tablespaces to be available in the new PDB. You can specify tablespaces in the `USER_TABLESPACE` clause at the time of PDB creation. Pre 12c databases, which earlier used the schema consolidation approach, will benefit from this new feature while moving to Oracle 12c Multitenant.
- Metadata-only clone – You can clone a PDB data model, and not its data, by specifying `NO DATA` at the time of cloning.
- Remote clone non-CDB as PDB – You can directly clone an Oracle 12c non-CDB as the PDB in a multitenant container database.

CDB Resource Management

Resource management is an essential exercise in a multitenant container database that hosts multiple application databases, with a common SGA and background process. With multiple pluggable databases, scenarios may arise where databases may compete for server resources. The request congestion might impact the performance of critical databases, thus impacting business SLAs. The **Database Resource Manager (DBRM)** feature has been enhanced in Oracle 12c to extend support to the multitenant architecture.

In a multitenant environment, you can create CDB plan directives to manage the allocation of the CPU as shares, and parallel execution slaves across the multiple pluggable databases. In the event of resource congestion, pluggable databases follow the CDB resource plan to prioritize the requests.

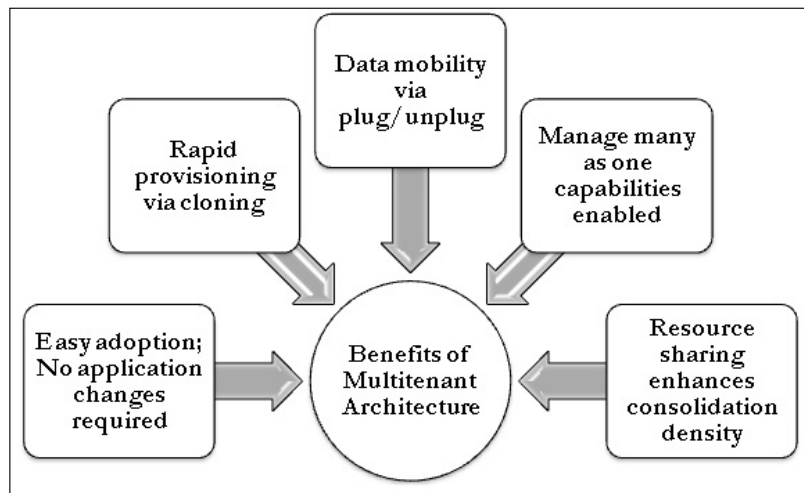
Within a pluggable database, you can create the usual database resource management plans to control the consumption of the available resources across consumer groups.

Common users and local users

Starting with Oracle 12c Multitenant architecture, there will be two families of users, namely **common users** and **local users**. The container DBA creates a common user in the root container. Local users are the ones that are visible and operational within a single pluggable database. All the users until Oracle Database 11g will fall under the category of local users.

Here are the important points regarding common and local users:

- All common users must start with C##. SYS is an exception.
- Pluggable databases must grant `CONNECT` and `CREATE SESSION` privilege to the common user to allow access to the container.
- The static views `CDB_USERS` and `DBA_USERS` have a new column `COMMON` to differentiate common and local users.
- A common user can be used to execute a generic script across multiple containers (using `catcon.pl`).
- With Oracle 12c (12.1.0.2), common users can query common objects in root the (`CDB$ROOT`) from multiple containers using the `CONTAINERS` clause.



Oracle 12c SQL and PL/SQL new features

SQL is the most widely used data access language while PL/SQL is a popular language that can integrate seamlessly with SQL commands. The biggest benefit of running PL/SQL is that the code processing happens natively within the Oracle Database. In the past, there have been debates and discussions on server side programming while the client invokes the PL/SQL routines to perform a task. The server side programming approach has many benefits. It reduces the network round trips between the client and the database. It reduces the code size and eases the code portability because PL/SQL can run on all platforms, wherever Oracle Database is supported.

Oracle Database 12c introduces many language features and enhancements that focus on SQL to PL/SQL integration, code migration, and ANSI compliance. This section discusses the SQL and PL/SQL new features in Oracle Database 12c.

IDENTITY columns

Oracle Database 12c introduces identity columns in SQL in compliance with the **American National Standard Institute (ANSI)** SQL standard. A table column, marked as `IDENTITY`, automatically generates an incremental numeric value at the time of record creation.

Before the release of Oracle 12c, developers had to create an additional sequence in the schema and assign its value to the column through a trigger or in a PL/SQL block. The new feature simplifies code writing and benefits the migration of a non-Oracle database to Oracle.

The following script declares an identity column in the table `T_ID_COL`:

```
/*Create a table for demonstration purpose*/
CREATE TABLE t_id_col
(id    NUMBER GENERATED AS IDENTITY,
 name VARCHAR2(20))
/
```

The identity column metadata can be queried from the dictionary views `USER_TAB_COLS` and `USER_TAB_IDENTITY_COLS`. Note that Oracle implicitly creates a sequence to generate the number values for the column. However, Oracle allows the configuration of the sequence attributes of an identity column. The custom sequence configuration is listed under `IDENTITY_OPTIONS` in `USER_TAB_IDENTITY_COLS` view:

```
/*Query identity column information in USER_TAB_COLS*/
SELECT column_name, data_default, user_generated, identity_column
FROM user_tab_cols
WHERE table_name='T_ID_COL'
```


Let us check the data in the table. Note the identity column values:

```
/*Query the table*/
SELECT id, name FROM t_id_col
/
```

```

ID NAME
-----
1 Allen
2 Matthew
3 Peter
```

The sequence created under the covers for identity columns is tightly coupled with the column. If a user tries to insert a user-defined input for the identity column, the operation throws an exception `ORA-32795`:

```
INSERT INTO t_id_col VALUES (7, 'Steyn');
insert into t_id_col values (7, 'Steyn')
*
ERROR at line 1:
ORA-32795: cannot insert into a generated always identity column
```

Default column value to a sequence in Oracle 12c

Oracle Database 12c allows developers to default a column directly to a sequence-generated value. The `DEFAULT` clause of a table column can be assigned to `SEQUENCE.CURRVAL` or `SEQUENCE.NEXTVAL`. The feature will be useful while migrating non-Oracle data definitions to Oracle.

The `DEFAULT ON NULL` clause

Starting with Oracle Database 12c, a column can be assigned a default non-null value whenever the user tries to insert `NULL` into the column. The default value will be specified in the `DEFAULT` clause of the column with a new `ON NULL` extension.

Note that the `DEFAULT ON NULL` cannot be used with an object type column.

The following script creates a table `t_def_cols`. A column `ID` has been defaulted to a sequence while the column `DOJ` will always have a non-null value:

```
/*Create a sequence*/
CREATE SEQUENCE seq START WITH 100 INCREMENT BY 10
/
```



```
/*Create a table with a column defaulted to the sequence value*/
CREATE TABLE t_def_cols
( id number default seq.nextval primary key,
  name varchar2(30),
  doj date default on null '01-Jan-2000'
)
/
```

The following PL/SQL block inserts the test data:

```
/*Insert the test data in the table*/
BEGIN
  INSERT INTO t_def_cols (name, doj) values ('KATE', '27-FEB-2001');
  INSERT INTO t_def_cols (name, doj) values ('NANCY', '17-JUN-1998');
  INSERT INTO t_def_cols (name, doj) values ('LANCE', '03-JAN-2004');
  INSERT INTO t_def_cols (name) values ('MARY');
  COMMIT;
END;
/
```

Query the table and check the values for the ID and DOJ columns. ID gets the value from the sequence SEQ while DOJ for MARY has been defaulted to 01-JAN-2000.

```
/*Query the table to verify sequence and default on null values*/
SELECT * FROM t_def_cols
/
```

ID	NAME	DOJ
100	KATE	27-FEB-01
110	NANCY	17-JUN-98
120	LANCE	03-JAN-04
130	MARY	01-JAN-00

Support for 32K VARCHAR2

Oracle Database 12c supports the VARCHAR2, NVARCHAR2, and RAW datatypes up to 32,767 bytes in size. The previous maximum limit for the VARCHAR2 (and NVARCHAR2) and RAW datatypes was 4,000 bytes and 2,000 bytes respectively. The support for extended string datatypes will benefit non-Oracle to Oracle migrations.

The feature can be controlled using the initialization parameter `MAX_STRING_SIZE`. It accepts two values:

- `STANDARD` (*default*) – The maximum size prior to the release of Oracle Database 12c will apply.
- `EXTENDED` – The new size limit for string datatypes apply. Note that after the parameter is set to `EXTENDED`, the setting cannot be rolled back.

The steps to increase the maximum string size in a database are:

1. Restart the database in `UPGRADE` mode. In the case of a pluggable database, the PDB must be opened in `MIGRATE` mode.
2. Use the `ALTER SYSTEM` command to set `MAX_STRING_SIZE` to `EXTENDED`.
3. As `SYSDBA`, execute the `$ORACLE_HOME/rdbms/admin/utl32k.sql` script. The script is used to increase the maximum size limit of `VARCHAR2`, `NVARCHAR2`, and `RAW` wherever required.
4. Restart the database in `NORMAL` mode.
5. As `SYSDBA`, execute `utlrp.sql` to recompile the schema objects with invalid status.

The points to be considered while working with the 32k support for string types are:

- `COMPATIBLE` must be 12.0.0.0
- After the parameter is set to `EXTENDED`, the parameter cannot be rolled back to `STANDARD`
- In RAC environments, all the instances of the database comply with the setting of `MAX_STRING_SIZE`

Row limiting using `FETCH FIRST`

For Top-N queries, Oracle Database 12c introduces a new clause, `FETCH FIRST`, to simplify the code and comply with ANSI SQL standard guidelines. The clause is used to limit the number of rows returned by a query. The new clause can be used in conjunction with `ORDER BY` to retrieve **Top-N** results.

The row limiting clause can be used with the `FOR UPDATE` clause in an SQL query. In the case of a materialized view, the defining query should not contain the `FETCH` clause.

Another new clause, `OFFSET`, can be used to skip the records from the top or middle, before limiting the number of rows. For consistent results, the offset value must be a positive number, less than the total number of rows returned by the query. For all other offset values, the value is counted as zero.

Keywords with the `FETCH FIRST` clause are:

- `FIRST | NEXT`—Specify `FIRST` to begin row limiting from the top. Use `NEXT` with `OFFSET` to skip certain rows.
- `ROWS | PERCENT`—Specify the size of the result set as a fixed number of rows or percentage of total number of rows returned by the query.
- `ONLY | WITH TIES`—Use `ONLY` to fix the size of the result set, irrespective of duplicate sort keys. If you want all records with matching sort keys, specify `WITH TIES`.

The following query demonstrates the use of the `FETCH FIRST` and `OFFSET` clauses in Top-N queries:

```
/*Create the test table*/
CREATE TABLE t_fetch_first
(empno VARCHAR2(30),
 deptno NUMBER,
 sal NUMBER,
 hiredate DATE)
/
```

The following PL/SQL block inserts sample data for testing:

```
/*Insert the test data in T_FETCH_FIRST table*/
BEGIN
  INSERT INTO t_fetch_first VALUES (101, 10, 1500, '01-FEB-2011');
  INSERT INTO t_fetch_first VALUES (102, 20, 1100, '15-JUN-2001');
  INSERT INTO t_fetch_first VALUES (103, 20, 1300, '20-JUN-2000');
  INSERT INTO t_fetch_first VALUES (104, 30, 1550, '30-DEC-2001');
  INSERT INTO t_fetch_first VALUES (105, 10, 1200, '11-JUL-2012');
  INSERT INTO t_fetch_first VALUES (106, 30, 1400, '16-AUG-2004');
  INSERT INTO t_fetch_first VALUES (107, 20, 1350, '05-JAN-2007');
  INSERT INTO t_fetch_first VALUES (108, 20, 1000, '18-JAN-2009');
  COMMIT;
END;
/
```

The `SELECT` query pulls in the top-5 rows when sorted by their salary:

```
/*Query to list top-5 employees by salary*/
SELECT *
FROM t_fetch_first
ORDER BY sal DESC
FETCH FIRST 5 ROWS ONLY
/
```

EMPNO	DEPTNO	SAL	HIREDATE
104	30	1550	30-DEC-01
101	10	1500	01-FEB-11
106	30	1400	16-AUG-04
107	20	1350	05-JAN-07
103	20	1300	20-JUN-00

The SELECT query lists the top 25% of employees (2) when sorted by their hiredate:

```
/*Query to list top-25% employees by hiredate*/
SELECT *
FROM t_fetch_first
ORDER BY hiredate FETCH FIRST 25 PERCENT ROW ONLY
/
```

EMPNO	DEPTNO	SAL	HIREDATE
103	20	1300	20-JUN-00
102	20	1100	15-JUN-01

The SELECT query skips the first five employees and displays the next two—the 6th and 7th employee data:

```
/*Query to list 2 employees after skipping first 5 employees*/
SELECT *
FROM t_fetch_first
ORDER BY SAL DESC
OFFSET 5 ROWS FETCH NEXT 2 ROWS ONLY
/
```

Invisible columns

Oracle Database 12c supports invisible columns, which implies that a user can control the visibility of a column. A column marked invisible does not appear in the following operations:

- SELECT * FROM queries on the table
- SQL*Plus DESCRIBE command
- Local records of %ROWTYPE
- **Oracle Call Interface (OCI)** description

A column can be made invisible by specifying the `INVISIBLE` clause against the column. Columns of all types (*except user-defined types*), including virtual columns, can be marked invisible, provided the tables are not temporary tables, external tables, or clustered. The `SELECT` statement can explicitly select an invisible column. Similarly, the `INSERT` statement will not insert values in an invisible column unless explicitly specified.

Furthermore, a table can be partitioned based on an invisible column. A column retains its nullity feature even after it is made invisible. An invisible column can be made visible, but the ordering of the column in the table may change.

In the following script, the column `NICKNAME` is set as invisible in the table `t_inv_col`:

```
/*Create a table to demonstrate invisible columns*/
CREATE TABLE t_inv_col
(id NUMBER,
 name VARCHAR2(30),
 nickname VARCHAR2 (10) INVISIBLE,
 dob DATE
)
/
```

The information about the invisible columns can be found in `user_tab_cols`. Note that the invisible column is marked as hidden:

```
/*Query the USER_TAB_COLS for metadata information*/
SELECT column_id,
       column_name,
       hidden_column
FROM   user_tab_cols
WHERE  table_name = 'T_INV_COL'
ORDER BY column_id
/
```

COLUMN_ID	COLUMN_NAME	HID
1	ID	NO
2	NAME	NO
3	DOB	NO
	NICKNAME	YES

Hidden columns are different from invisible columns. Invisible columns can be made visible and vice versa, but hidden columns cannot be made visible.

If we try to make the NICKNAME visible and NAME invisible, observe the change in column ordering:

```

/*Script to change visibility of NICKNAME column*/
ALTER TABLE t_inv_col MODIFY nickname VISIBLE
/
/*Script to change visibility of NAME column*/
ALTER TABLE t_inv_col MODIFY name INVISIBLE
/
/*Query the USER_TAB_COLS for metadata information*/
SELECT column_id,
       column_name,
       hidden_column
FROM   user_tab_cols
WHERE  table_name = 'T_INV_COL'
ORDER BY column_id
/

```

COLUMN_ID	COLUMN_NAME	HID
1	ID	NO
2	DOB	NO
3	NICKNAME	NO
	NAME	YES

Temporal databases

Temporal databases were released as a new feature in **ANSI SQL:2011**. The term temporal data can be understood as a piece of information that can be associated with a period within which the information is valid. Before the feature was included in Oracle Database 12c, data whose validity is linked with a time period had to be handled either by the application or using multiple predicates in the queries. Oracle 12c partially inherits the feature from the ANSI SQL:2011 standard to support the entities whose business validity can be bracketed with a time dimension.

The temporal database feature in Oracle Database 12c is different from the total recall feature in Oracle Database 11g. The total recall feature records the transaction time of the data in the database to secure the transaction validity and not the functional validity. For example, an investment scheme is active between January to December. The date recorded in the database at the time of data loading is the transaction timestamp.



Starting from Oracle 12c, the **Total Recall** feature has been rebranded as **Flashback Data Archive** and has been made available for all versions of Oracle Database.

The valid time temporal feature can be enabled for a table by adding a time dimension using the `PERIOD FOR` clause on the date or timestamp columns of the table. The following script creates a table `t_tmp_db` with valid time temporal:

```
/*Create table with valid time temporal*/
CREATE TABLE t_tmp_db(
  id NUMBER,
  name VARCHAR2(30),
  policy_no VARCHAR2(50),
  policy_term number,
  pol_st_date date,
  pol_end_date date,
  PERIOD FOR pol_valid_time (pol_st_date, pol_end_date))
/
```

Create some sample data in the table:

```
/*Insert test data in the table*/
BEGIN
  INSERT INTO t_tmp_db
  VALUES (100, 'Packt', 'PACKT_POL1', 1, '01-JAN-2015', '31-DEC-2015');
  INSERT INTO t_tmp_db
  VALUES (110, 'Packt', 'PACKT_POL2', 2, '01-JAN-2015', '30-JUN-2015');
  INSERT INTO t_tmp_db
  VALUES (120, 'Packt', 'PACKT_POL3', 3, '01-JUL-2015', '31-DEC-2015');
  COMMIT;
END;
/
```

Let us set the current time period window using `DBMS_FLASHBACK_ARCHIVE`. Grant the `EXECUTE` privilege on the package to the `scott` user.

```
/*Connect to sysdba to grant execute privilege to scott*/
conn sys/oracle as sysdba
GRANT EXECUTE ON dbms_flashback_archive to scott
/
```

Grant succeeded.

```
/*Connect to scott*/
conn scott/tiger
```

```
/*Set the valid time period as CURRENT*/
EXEC DBMS_FLASHBACK_ARCHIVE.ENABLE_AT_VALID_TIME('CURRENT');
```

PL/SQL procedure successfully completed.

Setting the valid time period as `CURRENT` means that all the tables with a valid time temporal will only list the rows that are valid with respect to today's date. You can set the valid time to a particular date too.

```
/*Query the table*/
SELECT * from t_tmp_db
/
```

ID	POLICY_NO	POL_ST_DATE	POL_END_DATE
100	PACKT_POL1	01-JAN-15	31-DEC-15
110	PACKT_POL2	01-JAN-15	30-JUN-15

[



Due to a dependency on the current date, the result may vary when the reader runs the preceding queries.

]

The query lists only those policies that are active as of March 2015. Since, the third policy starts in July 2015, it is currently not active.

In-Database Archiving

Oracle Database 12c introduces In-Database Archiving to archive the low priority data in a table. The inactive data remains in the database but is not visible to the application.

You can mark old data for archival, which is not actively required in the application except for regulatory purposes. Although the archived data is not visible to the application, it is available for querying and manipulation. In addition, the archived data can be compressed to improve backup performance.

A table can be enabled by specifying the `ROW ARCHIVAL` clause at the table level, which adds a hidden column `ORA_ARCHIVE_STATE` to the table structure. The column value must be updated to mark a row for archival. For example:

```
/*Create a table with row archiving*/
CREATE TABLE t_row_arch(
  x number,
  y number,
  z number) ROW ARCHIVAL
/
```


When we query the table structure in the USER_TAB_COLS view, we find an additional hidden column, which Oracle implicitly adds to the table:

```
/*Query the columns information from user_tab_cols view*/
SELECT column_id,column_name,data_type, hidden_column
FROM user_tab_cols
WHERE table_name='T_ROW_ARCH'
/
```

COLUMN_ID	COLUMN_NAME	DATA_TYPE	HID
	ORA_ARCHIVE_STATE	VARCHAR2	YES
1	X	NUMBER	NO
2	Y	NUMBER	NO
3	Z	NUMBER	NO

Let us create test data in the table:

```
/Insert test data in the table*/
BEGIN
  INSERT INTO t_row_arch VALUES (10,20,30);
  INSERT INTO t_row_arch VALUES (11,22,33);
  INSERT INTO t_row_arch VALUES (21,32,43);
  INSERT INTO t_row_arch VALUES (51,82,13);
  commit;
END;
/
```

For testing purpose, let us archive the rows in the table where x > 50 by updating the ora_archive_state column:

```
/*Update ORA_ARCHIVE_STATE column in the table*/
UPDATE t_row_arch
SET ora_archive_state = 1
WHERE x > 50
/
COMMIT
/
```

By default, the session displays only the active records from an archival-enabled table:

```
/*Query the table*/
SELECT *
FROM t_row_arch
/
```

X	Y	Z

10	20	30
11	22	33
21	32	43

If you wish to display all the records, change the session setting:

```
/*Change the session parameter to display the archived records*/
ALTER SESSION SET ROW ARCHIVAL VISIBILITY = ALL
/
```

Session altered.

```
/*Query the table*/
SELECT *
FROM t_row_arch
/
```

X	Y	Z
10	20	30
11	22	33
21	32	43
51	82	13

Defining a PL/SQL subprogram in the SELECT query and PRAGMA UDF

Oracle Database 12c includes two new features to enhance the performance of functions when called from `SELECT` statements. With Oracle 12c, a PL/SQL subprogram can be created inline with the `SELECT` query in the `WITH` clause declaration. The function created in the `WITH` clause subquery is not stored in the database schema and is available for use only in the current query. Since a procedure created in the `WITH` clause cannot be called from the `SELECT` query, it can be called in the function created in the declaration section. The feature can be very handy in read-only databases where the developers were not able to create PL/SQL wrappers.

Oracle Database 12c adds the new `PRAGMA UDF` to create a standalone function with the same objective.

Earlier, the `SELECT` queries could invoke a PL/SQL function, provided the function didn't change the database purity state. The query performance would degrade because of the context switch from SQL to the PL/SQL engine (and vice versa) and the different memory representations of data type in the processing engines.

In the following example, the function `fun_with_plsql` calculates the annual compensation of an employee

```
/*Create a function in WITH clause declaration*/
WITH FUNCTION fun_with_plsql (p_sal NUMBER) RETURN NUMBER IS
BEGIN
    RETURN (p_sal * 12);
END;
SELECT ename, deptno, fun_with_plsql (sal) "annual_sal"
FROM emp
/
```

ENAME	DEPTNO	annual_sal
-----	-----	-----
SMITH	20	9600
ALLEN	30	19200
WARD	30	15000
JONES	20	35700
MARTIN	30	15000
BLAKE	30	34200
CLARK	10	29400
SCOTT	20	36000
KING	10	60000
TURNER	30	18000
ADAMS	20	13200
JAMES	30	11400
FORD	20	36000
MILLER	10	15600

14 rows selected.



If the query containing the `WITH` clause declaration is not a top-level statement, then the top level statement must use the `WITH_PLSQL` hint. The hint is used if `INSERT`, `UPDATE`, or `DELETE` statements are trying to use a `SELECT` with a `WITH` clause definition. Failure to include the hint results in an exception `ORA-32034: unsupported use of WITH clause`.

A function can be created with the `PRAGMA UDF` to inform the compiler that the function is always called in a `SELECT` statement. Note that the standalone function created in the following code carries the same name as the one in the last example. The local `WITH` clause declaration takes precedence over the standalone function in the schema.

```

/*Create a function with PRAGMA UDF*/
CREATE OR REPLACE FUNCTION fun_with_plsql (p_sal NUMBER)
RETURN NUMBER IS
PRAGMA UDF;
BEGIN
RETURN (p_sal *12);
END;
/

```

Since the objective of the feature is performance, let us go ahead with a case study to compare the performance when using a standalone function, a PRAGMA UDF function, and a WITH clause declared function.

Test setup

The exercise uses a test table with 1 million rows, loaded with random data.

```

/*Create a table for performance test study*/
CREATE TABLE t_fun_plsql
(id number,
str varchar2(30))
/
/*Generate and load random data in the table*/
INSERT /*+APPEND*/ INTO t_fun_plsql
SELECT ROWNUM, DBMS_RANDOM.STRING('X', 20)
FROM dual
CONNECT BY LEVEL <= 1000000
/
COMMIT
/

```

- **Case 1:** Create a PL/SQL standalone function as it used to be until Oracle Database 12c. The function counts the numbers in the `str` column of the table.

```

/*Create a standalone function without Oracle 12c enhancements*/
CREATE OR REPLACE FUNCTION f_count_num (p_str VARCHAR2)
RETURN PLS_INTEGER IS
BEGIN
RETURN (REGEXP_COUNT(p_str, '\d'));
END;
/

```

The PL/SQL block measures the elapsed and CPU time when working with a pre-Oracle 12c standalone function. These numbers will serve as the baseline for our case study.

```
/*Set server output on to display messages*/
SET SERVEROUTPUT ON
/*Anonymous block to measure performance of a standalone
function*/
DECLARE
  l_el_time PLS_INTEGER;
  l_cpu_time PLS_INTEGER;
  CURSOR c1 IS
    SELECT f_count_num (str) FROM t_fun_plsql;
  TYPE t_tab_rec IS TABLE OF PLS_INTEGER;
  l_tab t_tab_rec;
BEGIN
  l_el_time := DBMS_UTILITY.GET_TIME ();
  l_cpu_time := DBMS_UTILITY.GET_CPU_TIME ();
  OPEN c1;
  FETCH c1 BULK COLLECT INTO l_tab;
  CLOSE c1;
  DBMS_OUTPUT.PUT_LINE ('Case 1: Performance of a standalone
function');
  DBMS_OUTPUT.PUT_LINE ('Total elapsed time:'||to_char(DBMS_
UTILITY.GET_TIME () - l_el_time));
  DBMS_OUTPUT.PUT_LINE ('Total CPU time:'||to_char(DBMS_UTILITY.
GET_CPU_TIME () - l_cpu_time));
END;
/
```

Performance of a standalone function:

```
Total elapsed time:1559
Total CPU time:1366
```

PL/SQL procedure successfully completed.

- **Case 2:** Create a PL/SQL function using PRAGMA UDF to count the numbers in the str column.

```
/*Create the function with PRAGMA UDF*/
CREATE OR REPLACE FUNCTION f_count_num_pragma (p_str VARCHAR2)
RETURN PLS_INTEGER IS
  PRAGMA UDF;
BEGIN
  RETURN (REGEXP_COUNT(p_str, '\d'));
END;
/
```

Let us now check the performance of the PRAGMA UDF function using the following PL/SQL block.

```

/*Set server output on to display messages*/
SET SERVEROUTPUT ON
/*Anonymous block to measure performance of a PRAGMA UDF
function*/
DECLARE
    l_el_time PLS_INTEGER;
    l_cpu_time PLS_INTEGER;
    CURSOR c1 IS
        SELECT f_count_num_pragma (str) FROM t_fun_plsql;
    TYPE t_tab_rec IS TABLE OF PLS_INTEGER;
    l_tab t_tab_rec;
BEGIN
    l_el_time := DBMS_UTILITY.GET_TIME ();
    l_cpu_time := DBMS_UTILITY.GET_CPU_TIME ();
    OPEN c1;
    FETCH c1 BULK COLLECT INTO l_tab;
    CLOSE c1;
    DBMS_OUTPUT.PUT_LINE ('Case 2: Performance of a PRAGMA UDF
function');
    DBMS_OUTPUT.PUT_LINE ('Total elapsed time: ' || to_char(DBMS_
UTILITY.GET_TIME () - l_el_time));
    DBMS_OUTPUT.PUT_LINE ('Total CPU time: ' || to_char(DBMS_UTILITY.
GET_CPU_TIME () - l_cpu_time));
END;
/

```

Performance of a PRAGMA UDF function:

Total elapsed time:664

Total CPU time:582

PL/SQL procedure successfully completed.

- **Case 3:** The following PL/SQL block dynamically executes the function in the WITH clause subquery. Note that, unlike other SELECT statements, a SELECT query with a WITH clause declaration cannot be executed statically in the body of a PL/SQL block.

```

/*Set server output on to display messages*/
SET SERVEROUTPUT ON
/*Anonymous block to measure performance of inline function*/
DECLARE
    l_el_time PLS_INTEGER;
    l_cpu_time PLS_INTEGER;
    l_sql VARCHAR2(32767);

```

```
c1 sys_refcursor;
TYPE t_tab_rec IS TABLE OF PLS_INTEGER;
l_tab t_tab_rec;
BEGIN
  l_el_time := DBMS_UTILITY.get_time;
  l_cpu_time := DBMS_UTILITY.get_cpu_time;
  l_sql := 'WITH FUNCTION f_count_num_with (p_str VARCHAR2)
           RETURN NUMBER IS
           BEGIN
             RETURN (REGEXP_COUNT(p_str, ''||'\'||'\d'||'\'))';
           END;
           SELECT f_count_num_with(str) FROM t_fun_plsql';
  OPEN c1 FOR l_sql;
  FETCH c1 bulk collect INTO l_tab;
  CLOSE c1;
  DBMS_OUTPUT.PUT_LINE ('Case 3: Performance of an inline
function');
  DBMS_OUTPUT.PUT_LINE ('Total elapsed time: '||to_char(DBMS_
UTILITY.GET_TIME () - l_el_time));
  DBMS_OUTPUT.PUT_LINE ('Total CPU time: '||to_char(DBMS_UTILITY.
GET_CPU_TIME () - l_cpu_time));
END;
/
```

Performance of an inline function:

Total elapsed time:830

Total CPU time:718

PL/SQL procedure successfully completed.

Comparative analysis

Comparing the results from the preceding three cases, it's clear that the Oracle 12c flavor of PL/SQL functions out-performs the pre-12c standalone function by a high margin. From the following matrix, it is apparent that the usage of the `PRAGMA UDF` or `WITH` clause declaration enhances the code performance by (roughly) a factor of 2.

Case Description	Elapsed Time	CPU time	Performance gain factor by CPU time
Standalone PL/SQL function in pre-Oracle 12c database	1559	1336	1x
Standalone PL/SQL PRAGMA UDF function in Oracle 12c	664	582	2.3x
Function created in WITH clause declaration in Oracle 12c	830	718	1.9x

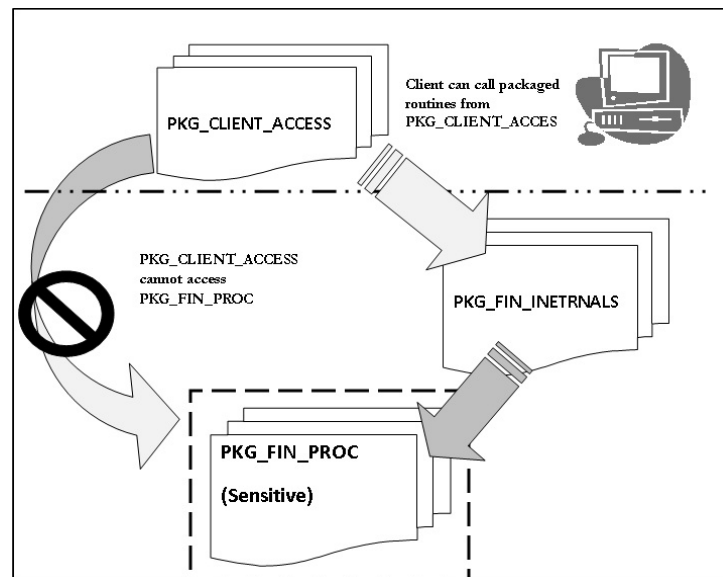


Note that the numbers may slightly differ in the reader's testing environment but you should be able to draw the same conclusion by comparing them.

The PL/SQL program unit white listing


Prior to Oracle 12c, a standalone or packaged PL/SQL unit could be invoked by all other programs in the session's schema. Oracle Database 12c allows users to prevent unauthorized access to PL/SQL program units. You can now specify the list of whitelist program units that can invoke a particular program. The PL/SQL program header or the package specification can specify the list of program units in the `ACCESSIBLE BY` clause in the program header. All other program units, including cross-schema references (even SYS owned objects), trying to access a protected subprogram will receive an exception, PLS-00904: insufficient privileges to access object [object name].

The feature can be very useful in an extremely sensitive development environment. Suppose, a package `PKG_FIN_PROC` contains the sensitive implementation routines for financial institutions, the packaged subprograms are called by another PL/SQL package `PKG_FIN_INTERNALS`. The API layer exposes a fixed list of programs through a public API called `PKG_CLIENT_ACCESS`. In order to restrict access to the packaged routines in `PKG_FIN_PROC`, the users can build a safety net so as to allow access to only authorized programs.



The following PL/SQL package `PKG_FIN_PROC` contains two subprograms — `P_FIN_QTR` and `P_FIN_ANN`. The `ACCESSIBLE BY` clause includes `PKG_FIN_INTERNALS` which means that all other program units, including anonymous PL/SQL blocks, are blocked from invoking `PKG_FIN_PROC` constructs.

```
/*Package with the accessible by clause*/
CREATE OR REPLACE PACKAGE pkg_fin_proc
ACCESSIBLE BY (PACKAGE pkg_fin_internals)
IS
    PROCEDURE p_fin_qtr;
    PROCEDURE p_fin_ann;
END;
/
```

 The `ACCESSIBLE BY` clause can be specified for schema-level programs only.

Let's see what happens when we invoke the packaged subprogram from an anonymous PL/SQL block.

```
/*Invoke the packaged subprogram from the PL/SQL block*/
BEGIN
    pkg_fin_proc.p_fin_qtr;
END;
/
pkg_fin_proc.p_fin_qtr;
*
ERROR at line 2:
ORA-06550: line 2, column 4:
PLS-00904: insufficient privilege to access object PKG_FIN_PROC
ORA-06550: line 2, column 4:
PL/SQL: Statement ignored
```

Well, the compiler throws an exception as invoking the whitelisted package from an anonymous block is not allowed.

The `ACCESSIBLE BY` clause can be included in the header information of PL/SQL procedures and functions, packages, and object types.

Granting roles to PL/SQL program units

Before Oracle Database 12c, a PL/SQL unit created with the definer's rights (default `AUTHID`) always executed with the definer's rights, whether or not the invoker has the required privileges. It may lead to an unfair situation where the invoking user may perform unwanted operations without needing the correct set of privileges. Similarly for an invoker's right unit, if the invoking user possesses a higher set of privileges than the definer, he might end up performing unauthorized operations.

Oracle Database 12c secures the definer's rights by allowing the defining user to grant complementary roles to individual PL/SQL subprograms and packages. From the security standpoint, the granting of roles to schema level subprograms, provides granular control as the privileges of the invoker are validated at the time of execution.

In the following example, we will create two users: U1 and U2. The user U1 creates a PL/SQL procedure `P_INC_PRICE` that adds a surcharge to the price of a product by a certain amount. U1 grants the execute privilege to user U2.

Test setup

Let's create two users and give them the required privileges.

```
/*Create a user with a password*/
CREATE USER u1 IDENTIFIED BY u1
/

User created.

/*Grant connect privileges to the user*/
GRANT CONNECT, RESOURCE TO u1
/

Grant succeeded.

/*Create a user with a password*/
CREATE USER u2 IDENTIFIED BY u2
/

User created.

/*Grant connect privileges to the user*/
GRANT CONNECT, RESOURCE TO u2
/

Grant succeeded.
```

The user U1 contains the `PRODUCTS` table. Let's create and populate the table.

```
/*Connect to U1*/
CONN u1/u1
/*Create the table PRODUCTS*/
CREATE TABLE products
```

```
(
    prod_id      INTEGER,
    prod_name    VARCHAR2(30),
    prod_cat     VARCHAR2(30),
    price        INTEGER
)
/

/*Insert the test data in the table*/
BEGIN
    DELETE FROM products;
    INSERT INTO products VALUES (101, 'Milk', 'Dairy', 20);
    INSERT INTO products VALUES (102, 'Cheese', 'Dairy', 50);
    INSERT INTO products VALUES (103, 'Butter', 'Dairy', 75);
    INSERT INTO products VALUES (104, 'Cream', 'Dairy', 80);
    INSERT INTO products VALUES (105, 'Curd', 'Dairy', 25);
    COMMIT;
END;
/
```

The procedure `p_inc_price` is designed to increase the price of a product by a given amount. Note that the procedure is created with the definer's rights.

```
/*Create the procedure with the definer's rights*/
CREATE OR REPLACE PROCEDURE p_inc_price
(p_prod_id NUMBER, p_amt NUMBER)
IS
BEGIN
    UPDATE products
    SET price = price + p_amt
    WHERE prod_id = p_prod_id;
END;
/
```

The user U1 grants execute privilege on `p_inc_price` to U2.

```
/*Grant execute on the procedure to the user U2*/
GRANT EXECUTE ON p_inc_price TO U2
/
```

The user U2 logs in and executes the procedure `P_INC_PRICE` to increase the price of Milk by 5 units.

```
/*Connect to U2*/
CONN u2/u2
/*Invoke the procedure P_INC_PRICE in a PL/SQL block*/
```

```
BEGIN
  U1.P_INC_PRICE (101,5);
  COMMIT;
END;
/
```

PL/SQL procedure successfully completed.

The last code listing exposes a gray area. The user U2, though not authorized to view PRODUCTS data, manipulates its data with the definer's rights.

We need a solution to the problem. The first step is to change the procedure from definer's rights to invoker's rights.

```
/*Connect to U1*/
CONN u1/u1
/*Modify the privilege authentication for the procedure to invoker's
rights*/
CREATE OR REPLACE PROCEDURE p_inc_price
(p_prod_id NUMBER, p_amt NUMBER)
AUTHID CURRENT_USER
IS
BEGIN
  UPDATE products
  SET price = price + p_amt
  WHERE prod_id = p_prod_id;
END;
/
```

Now, if we execute the procedure from U2, it throws an exception because it couldn't find the PRODUCTS table in its schema.

```
/*Connect to U2*/
CONN u2/u2
/*Invoke the procedure P_INC_PRICE in a PL/SQL block*/
BEGIN
  U1.P_INC_PRICE (101,5);
  COMMIT;
END;
/
BEGIN
*
ERROR at line 1:
ORA-00942: table or view does not exist
ORA-06512: at "U1.P_INC_PRICE", line 5
ORA-06512: at line 2
```

In a similar scenario in the past, the database administrators could have easily granted select or updated privileges to U2, which is not an optimal solution from a security standpoint. Oracle 12c allows users to create program units with invoker's rights but grant the required roles to the program units and not the users. So, an invoker right unit executes with invoker's privileges, plus the PL/SQL program role.

Let's check out the steps to create a role and assign it to the procedure. SYSDBA creates the role and assigns it to the user U1. Using the ADMIN or DELEGATE option with the grant enables the user to grant the role to other entities.

```
/*Connect to SYSDBA*/
CONN sys/oracle as sysdba
/*Create a role*/
CREATE ROLE prod_role
/
/*Grant role to user U1 with delegate option*/
GRANT prod_role TO U1 WITH DELEGATE OPTION
/
```

Now, user U1 assigns the required set of privileges to the role. The role is then assigned to the required subprogram. Note that only roles, and not individual privileges, can be assigned to the schema level subprograms.

```
/*Connect to U1*/
CONN u1/u1
/*Grant SELECT and UPDATE privileges on PRODUCTS to the role*/
GRANT SELECT, UPDATE ON PRODUCTS TO prod_role
/
/*Grant role to the procedure*/
GRANT prod_role TO PROCEDURE p_inc_price
/
```

User U2 tries to execute the procedure again. The procedure is successfully executed which means the value of "Milk" has been increased by 5 units.

```
/*Connect to U2*/
CONN u2/u2
/*Invoke the procedure P_INC_PRICE in a PL/SQL block*/
BEGIN
  U1.P_INC_PRICE (101,5);
  COMMIT;
END;
/
```

PL/SQL procedure successfully completed.

User U1 verifies the result with a `SELECT` query.

```
/*Connect to U1*/
CONN u1/u1
/*Query the table to verify the change*/
SELECT *
FROM products
/
```

PROD_ID	PROD_NAME	PROD_CAT	PRICE
101	Milk	Dairy	25
102	Cheese	Dairy	50
103	Butter	Dairy	75
104	Cream	Dairy	80
105	Curd	Dairy	25

Miscellaneous PL/SQL enhancements

Besides the preceding key features, there are a lot of new features in Oracle 12c. The list of features is as follows:

- An invoker rights function can be result-cached – Prior to Oracle Database 12c only the definers' programs were allowed to cache their results. Oracle 12c adds the invoking user's identity to the result cache to make it independent of the definer.
- The compilation parameter `PLSQL_DEBUG` has been deprecated.
- Two conditional compilation inquiry directives `$$PLSQL_UNIT_OWNER` and `$$PLSQL_UNIT_TYPE` have been implemented.

The Oracle Database 12c (12.1.0.2)

In-Memory option

The Oracle Database 12.1.0.2 introduces the In-Memory option that has the capability to speed up real-time analytics by an order of magnitude. The faster analytics complements and enables real-time decision making. Long-running reports and ad-hoc analytical queries are expected to benefit the most. The feature can be implemented without any application changes and works transparently with no manual hindrance, thus resulting in improved productivity.

The challenge

Enterprise applications have been reported to have mixed workloads – that is, OLTP workloads and analytics processing. In the past, there have been a couple of approaches to segregating the workloads. Mixed workload production databases can run on the same system, but running both of them simultaneously would degrade the OLTP performance. Running workloads on separate server systems impacts the real-time decision making because data on the analytics server has to be refreshed from time to time.

The problem statement and Oracle Database 12c In-Memory

Oracle Database is a trusted relational database management system that stores data in a row format. For transactional databases, data stored in a row format is a mandate because transactions work on a record basis and require all the attributes of a table in a single fetch. On the other hand, data analytics and reports, which run on few columns of data while also spanning many rows, work well with the columnar format. Until now, enterprises were forced to choose either of the two formats.

Oracle Database 12c In-Memory allows the database to be represented in a row format as well as a columnar format, thus providing the flavor of a "dual-format" architecture within the database. A piece of data can be represented in a row format as well as columnar format. The transactions continue to follow the row format of the data while the analytics workload work with the columnar format. The analytics get more real-time as the columnar format accelerates it by an order of magnitude. The best-of-both-worlds strategy is enabled by switching on the In-Memory feature in the Oracle Database.

The In-Memory feature marks a memory area (known as the In-Memory Column Store) within the **System Global Area (SGA)**. This memory space is used to hold the objects frequently referenced by the analytics queries and reports. It implies that enabling the In-Memory feature for a database doesn't require double memory requirements. However, databases may require some additional memory to accommodate their active objects in the In-Memory store.

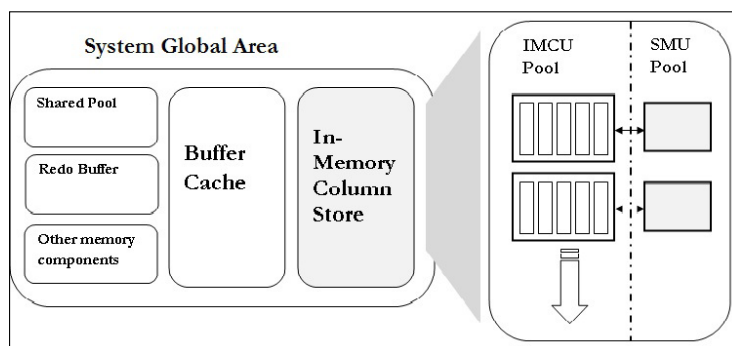
Oracle Database 12c In-Memory option features

The Oracle Database 12c In-Memory feature was released as an option in the patchset release (12.1.0.2) of Oracle Database 12c Release 1. The following list of features includes some must-know information about this option:

- The In-Memory column store is not a replacement for the buffer cache; rather, it supplements it.
- In-Memory Column Store is a new static pool within the **System Global Area (SGA)**. Being In-Memory, it is non-persistent and non-logging. It is not affected by Automatic Memory Management and the resident objects stay populated until they are manually flushed out.
- Administrators or users are authorized to identify those objects which when populated in the In-Memory column store, would yield the best performance.
- All objects except Index Organized Tables, Clustered tables, LONG columns, and Out-of-Line LOBs can be populated in the In-Memory column store.
- The Oracle Database optimizer is fully aware of the In-memory store; it decides which query would benefit from the buffer cache or in-memory columnar store.
- It is a licensed option, available starting from Oracle Database 12.1.0.2 Enterprise edition.

The Oracle Database 12c In-Memory Architecture

The System Global Area contains a new static pool, known as the In-Memory column store. The segments that are marked and populated in the In-Memory column store are oriented in the columnar format. Diving deep into the technical aspects of In-Memory column store, the static pool comprises of two pools: the **IMCU** (or **1MB**) pool and the **SMU** (or **64KB**) pool. The IMCU pool comprises In-memory compression units (IMCU) that hold the actual data in a columnar format. For each IMCU, there is a co-related SMU to store the IMCU's metadata and a transaction journal. The distribution of **1MB** and **64KB** pools are based completely on internal factors. The current allocation can be viewed under the `V$INMEMORY_AREA` dictionary view. The following figure shows the architecture of SGA and the In-Memory column store in Oracle Database 12c:



Controlling the In-Memory column store

The In-Memory Column Store can be configured through a new set of initialization parameters, introduced in Oracle 12c. These parameters control In-Memory dynamics such as sizing, the optimizer's behavior, and worker processes to be deployed for the population. Here is the list of initialization parameters:

- `INMEMORY_SIZE` (default 0): This configures the In-Memory store by setting this parameter for a minimum of *100MB*. The database must be bounced for the changes to take effect.
- `INMEMORY_QUERY` (default `ENABLE`): This parameter controls whether the queries should be optimized using the In-Memory store.
- `INMEMORY_MAX_POPULATE_SERVERS` (default 0): Configures the number of worker processes (max) to be used for In-Memory column store populate operations.
- `INMEMORY_CLAUSE_DEFAULT`: This sets the default In-Memory clause or sub-clause. By default, the value of the clause is `NULL`.
- `INMEMORY_TRICKLE_REPOPULATE_SERVERS_PERCENT`: This sets the percentage of worker processes that can perform trickle repopulation. The default value of the parameter is 1%.
- `INMEMORY_FORCE` (default `DEFAULT`): Setting this to `OFF` restricts the In-memory column store population.
- `OPTIMIZER_INMEMORY_AWARE` (default `TRUE`): This controls whether the optimizer should be aware or unaware of the In-Memory column store.

The INMEMORY clause

The objects required to be populated in the In-Memory column store can have the additional `INMEMORY` clause. The `INMEMORY` attribute can be specified for a table, columns, partition, materialized view, or a tablespace. In addition to the `INMEMORY` clause, there are other sub-clauses for some important aspects, such as population priority and compression.

The following In-Memory sub-clauses are applied by default along with the `INMEMORY` clause. To override the default behavior, you must specify the desired value.

- `MEMCOMPRESS`: The sub-clause determines the compression mode of the in-memory objects. The admissible compression modes are:
 - `NO MEMCOMPRESS`: No compression.
 - `MEMCOMPRESS FOR DML`: Compression for frequently transactional objects.

- `MEMCOMPRESS FOR QUERY LOW` (default): Balanced compression mode to optimally compress and ensure space savings. Enhances the query performance.
- `MEMCOMPRESS FOR QUERY HIGH`: Compression mode that focuses on query performance but checks the space savings too.
- `MEMCOMPRESS FOR CAPACITY LOW`: Compression mode for optimal space savings.
- `MEMCOMPRESS FOR CAPACITY HIGH`: Compression approach is the highest degree of space savings.
- **PRIORITY**: The `PRIORITY` sub-clause determines whether an object, which is marked as `INMEMORY`, can be populated automatically or manually. There are five possible values of `PRIORITY` clause:
 - `CRITICAL`: Critical priority objects are populated immediately after the database is opened or through the **In-Memory Co-ordinator (IMCO)** process's timely wake-up
 - `HIGH`: After the population of `CRITICAL` priority objects completes and the In-Memory column store has vacant space
 - `MEDIUM`: After the population of `CRITICAL` and `HIGH` priority objects completes and the In-memory column store has vacant space
 - `LOW`: After the population of `CRITICAL`, `HIGH`, and `MEDIUM` priority objects completes and the In-memory column store has vacant space
 - `NONE` (Default): The `NONE` priority segments are populated after the first full scan
- **DISTRIBUTE**: The `DISTRIBUTE` sub-clause is used in clustered environments (Oracle Database Real Application Cluster) to distribute the object data across the In-Memory Column Store on all the cluster nodes.
- **DUPLICATE**: The `DUPLICATE` sub-clause is exclusively for the members of the Oracle Engineered Systems family. It allows the duplication of the In-memory column store across selective or all nodes of the cluster for high availability.

Performance optimizations

Oracle Database In-Memory feature is designed for analytics performance. The optimizations that account for overall performance are as follows:

- Columnar Format and Vector Processing: The column format enables only the required column to be scanned, and not the complete record. The column format supports **Single Instruction Multiple Data (SIMD)** processing, which helps in processing multiple data values in each CPU instruction.
- Predicate evaluation and Join operations pushdown to the In-Memory column store: Predicates can be pushed down to the IM column store for evaluation. The In-Memory column store makes use of bloom filters to join multiple tables together.
- The In-Memory storage index provides min-max pruning that helps in preventing the IMCUs from scanning: Predicates can be checked against the IMCU header, which maintains information about minimum and maximum values. It helps in determining whether to scan or skip an IMCU.
- The evaluation of a query predicate can be minimized if the IMCU header satisfies the predicate: If the IMCU header fully or partially satisfies the predicate condition, the predicate evaluation can be prevented or reduced for the columnar units.

In-Memory Advisor

For large application databases, choosing the most suitable objects to be populated in the In-Memory Column Store can be a challenge. Oracle provides an In-Memory Advisor kit to recommend those objects whose in-memory format will yield the maximum benefits. The tool analyses the database workload through **Automatic Workload Repository (AWR)** and **Active Session History (ASH)** repositories, plan cardinalities, and parallel execution. Once the analysis is completed, it generates HTML advisory reports. The reports provide the list of objects that would benefit the most, when placed in the In-Memory column store.

The In-Memory Advisor is part of Oracle Tuning Pack and can be installed in **Oracle Database 11.2.0.3** and above.

Oracle Database In-Memory benefits

Oracle Database 12c In-Memory offers a dual-format architecture to support mixed workloads. An object can be represented in row format as well as columnar format. The columnar format is read-consistent and transactional-consistent with the data on disk. The In-Memory feature is embedded natively in the Oracle Database. Therefore, it is supported on all Oracle Database-supported platforms. Also, it is compatible with all database technologies such as Real Application Clusters, Multitenant, High Availability, and Exadata Engineered Systems.

Summary

This chapter familiarizes the readers with an overview of Oracle Database 12c. This chapter provides valuable insight to database developers in the application development space. Also, the chapter covers the top rated features of Oracle Database 12c, that is Multitenant and Database In-Memory. This chapter will help you understand the basic building blocks of a multitenant container database.

In the next chapter, we will cover the fundamentals of PL/SQL code design through cursors.

3

Designing PL/SQL Code

The structure of a PL/SQL block is one of the elementary components of PL/SQL as it showcases its modeling capabilities. It enables users to declare variables, include procedural constructs in the executable section, and embed exception management within the program.

All SQL statements within a PL/SQL block are executed as a cursor. Cursors are PL/SQL constructs that enable interaction with the data within a PL/SQL block. Cursor designing is an important skill in PL/SQL programming as it impacts the data access paradigm and also code performance. In this chapter, we are going to focus our discussion on cursors. Here is the chapter outline:

- Cursor fundamentals
 - a. How cursors work?
 - b. Implicit and explicit cursors
 - c. Cursor attributes
 - d. Cursor design guidelines
- Cursor variables
- Implicit `REF CURSOR` parameter binding
- Introduction to subtypes

Cursor structures

In PL/SQL, a cursor structure allows the processing of a `SELECT` statement and accesses the result returned by that query. Each and every SQL statement in a PL/SQL block is a cursor. A cursor is a handle to the chunk of the memory area where the SQL statements are processed and the result is stored. For a dedicated database, the chunk of memory is in the **User Global Area (UGA)** while, for shared server connections, the cursor context area is allocated in the **System Global Area (SGA)**.

Cursors can be of two types:

- **Implicit cursors:** Every SQL query in the executable or exception section of a PL/SQL block is an implicit cursor. `SELECT . . INTO`, `SELECT . . BULK COLLECT INTO`, `SELECT in CURSOR FOR loop`, `INSERT`, `UPDATE`, `DELETE`, and `MERGE` are implicit cursors.
- **Explicit cursors:** A cursor defined by the user or developer in the declaration section of a PL/SQL program is an explicit cursor.

Cursor execution cycle

A cursor is a handler to execute an SQL query and lives for the life of a session. Once the current session ends, the cursor no longer exists. After the cursor gets created implicitly or explicitly, it goes through the following stages of execution.

- **OPEN:** As soon as the cursor gets created, Oracle allocates a private area in the session's user global area (UGA). This private area is used for SQL statement processing. Prior to opening a cursor, it remains as a null pointer variable.



The initialization parameter `OPEN_CURSORS` governs the maximum number of cursors (from the library cache) that can be opened in a session.

- **PARSE:** Oracle checks the SQL statement for the syntactical correctness, semantics, and privileges.
- **BIND:** If the SQL statement needs additional input values for processing, the respective placeholders are replaced by actual values.

- **EXECUTE:** The SQL statement is executed following the conventional execution process. Oracle generates the hash value for the SQL statements and places it in the shared pool. Oracle also performs library cache lookup to search for any past executions of the same SQL. A successful lookup in the library cache avoids hard parsing of SQL statement. If the hash is not found, a new execution plan is generated and the SQL is processed. Once the SQL query is executed, the result set is placed in the UGA.
- **FETCH:** Fetch the record from the result set corresponding to the current position of the record pointer. The record pointer leaps forward by one after every successful fetch.
- **CLOSE:** The cursor handle is closed and the private context area is flushed out.

You can query the `V$OPEN_CURSOR` view to get the list of cursors used in the current session. Let us execute the following PL/SQL anonymous block and check the entries in `V$OPEN_CURSOR`:

```
connect scott/tiger
/*Declare a quick PL/SQL block */
DECLARE
    count_emp NUMBER;
    count_dep NUMBER;
BEGIN
    /*Create two implicit cursors */
    SELECT COUNT(*) INTO count_emp FROM emp;
    SELECT COUNT(*) INTO count_dep FROM dept;
END;
/
```

PL/SQL procedure successfully completed.

Let us query the `V$OPEN_CURSOR` view to check the open and PL/SQL cached cursors:

```
conn sys/oracle as sysdba
SELECT cursor_type,
       sql_text
FROM v$open_cursor
WHERE user_name='SCOTT'
AND cursor_type != 'DICTIONARY LOOKUP CURSOR CACHED'
ORDER BY cursor_type
/
```

```
CURSOR_TYPE          SQL_TEXT
-----
```



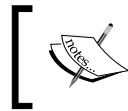
```
OPEN                                declare count_rec number; be
                                   in  select count(*) into count

PL/SQL CURSOR CACHED               SELECT COUNT(*) FROM DEPT
PL/SQL CURSOR CACHED               SELECT COUNT(*) FROM EMP
```

In the preceding output, the two `SELECT` queries (or implicit cursors of the PL/SQL block) are PL/SQL cursor-cached while the PL/SQL block is in the `OPEN` state.

Cursor attributes

Cursor attributes reveal the necessary information about the last active cursor. Cursor attributes are not persisted in the database but are aligned along with the query result set in the session memory. These attributes are `%ROWCOUNT`, `%ISOPEN`, `%FOUND`, and `%NOTFOUND`.



`%BULK_ROWCOUNT` and `%BULK_EXCEPTIONS` are additional cursor attributes used in bulk processing using the `FORALL` statement.

The cursor attributes are briefly explained as below:

- `%ROWCOUNT`: The attribute returns the number of rows fetched or affected by the SQL statement in the context area. It must be referenced within the cursor execution cycle. If referenced outside, it raises the `INVALID_CURSOR` exception.
- `%ISOPEN`: The attribute is set to `TRUE` if the cursor is currently open; otherwise it is `FALSE`. Programmers use this attribute outside the cursor execution cycle to check if the cursor is open or closed.
- `%FOUND`: The attribute returns `TRUE` if the row pointer points to a valid record. After the last record of the result set is reached, the attribute is set to `FALSE`.
- `%NOTFOUND`: The attribute returns the reverse of the `%FOUND` attribute.

Implicit cursors

Every SQL statement in the executable or exception section of a PL/SQL block is an implicit cursor. The database takes full charge of its entire execution cycle, meaning that the implicit cursor is auto-created, auto-opened, auto-fetched, and auto-closed. All of these steps are taken care by the Oracle Database. SQL statements can be `SELECT`, `INSERT`, `UPDATE`, `DELETE`, or `MERGE`, thus making an implicit cursor an SQL cursor.

The `SELECT` statement forming an implicit cursor is expected to return exactly one row. If it fails to return a single row, the implicit cursor raises `TOO_MANY_ROWS` or `NO_DATA_FOUND` exception. Exceptions can be trapped and handled with an informational message. If the cursor SQL is expected to return more than one row, you must create an explicit cursor.

Note that `SQL%` prefixes the cursor attributes for implicit cursors.

Cursor attributes	Description
<code>SQL%FOUND</code>	This attribute returns <code>TRUE</code> if <code>SELECT</code> fetches a single row or the DML statement affects a minimum of one row in the table. Otherwise, it is set as <code>FALSE</code> .
<code>SQL%NOTFOUND</code>	This attribute returns <code>TRUE</code> if <code>SELECT...INTO</code> fetches no row from the database. You might encounter <code>NO_DATA_FOUND</code> exception.
<code>SQL%ROWCOUNT</code>	This attribute returns 1 for the <code>SELECT</code> statement. For DML statements, it returns the number of rows affected by the DML. However, the attribute value is independent of the transaction state. If the transaction is rolled back to a savepoint, the attribute value is not restored to the one before rollback was issued.
<code>SQL%ISOPEN</code>	Always <code>FALSE</code> for implicit cursors.



In Oracle Database 12c, the maximum number returned by `SQL%ROWCOUNT` is 4,294,967,295.

The following PL/SQL block contains a `SELECT...INTO` statement in the executable section of the block:

```

/*Enable the SERVEROUTPUT to print the results */
SET SERVEROUTPUT ON
/*Demonstrate implicit cursor in PL/SQL execution block*/
DECLARE
    l_ename emp.ename%TYPE;
    l_sal emp.sal%TYPE;
BEGIN
    /*Select name and salary of employee 7369 */
    SELECT ename, sal
    INTO l_ename, l_sal
    FROM emp
    WHERE empno = 7369;
    DBMS_OUTPUT.PUT_LINE('Rows selected:' || SQL%ROWCOUNT);

```

```
END;  
/  
  
Rows selected:1  
  
PL/SQL procedure successfully completed.
```

The preceding PL/SQL block returns 1 because empno is the primary key in the emp table and there exists only one row against the value 7369.

Now let us try to update a multi-row data set in the employees table. The following PL/SQL block increases the salary of employees who are working in department 10:

```
/*Enable the SERVEROUTPUT to print the results */  
SET SERVEROUTPUT ON  
/*Demonstrate the cursor attribute during DML in a PL/SQL block*/  
BEGIN  
/*Increase the salary of employees from department 10*/  
    UPDATE emp  
    SET sal = sal + 1000  
    WHERE deptno = 10;  
    DBMS_OUTPUT.PUT_LINE('Rows updated:' || SQL%ROWCOUNT);  
END;  
/  
  
Rows updated:3  
  
PL/SQL procedure successfully completed.
```

Explicit cursors

Application developers can choose to create a cursor manually, perform open and fetch operations, and close the cursor. Such cursors are known as explicit cursors. They are more developer-friendly as they allow users to manage their execution stages and, most importantly, handle multi-row data sets.

An explicit cursor can be associated with SELECT queries only. The cursor prototype, defined in the DECLARE section of a PL/SQL block, should contain a valid name. The following PL/SQL block shows the cursor prototyping and handling stages in the executable section.

```
DECLARE  
    CURSOR [Cursor Name] [Parameters]  
    RETURN [Return type]  
    IS  
    [SELECT statement];
```

```

BEGIN
    OPEN [Cursor Name];
    FETCH...INTO [ scalar or composite variables ];
    CLOSE [Cursor Name];
END;

```

In the executable section of a PL/SQL block, a user has to open a cursor as `OPEN [cursor name]`. The data can be fetched using `FETCH [cursor name] INTO [variables or record variable]`. Once the fetch operation is over, a cursor can be closed using the `CLOSE [cursor name]` statement. Here is what happens at each of these stages:

- **OPEN stage:**
 1. **Open cursor:** It allocates a private work area in the user's session memory for cursor processing.
 2. **Parse SQL:** It validates the SQL query for syntax and privileges.
 3. **Bind SQL:** This provides an input value to the bind variables in the query.
 4. **Execute the query:** It executes the parsed SQL statement.
- **FETCH stage:** This stage iterates the data set for each fetch request. It fetches the data into block variables (or records) and increments the record pointer.
- **CLOSE stage:** This stage closes the cursor and releases the memory back to SGA.

Oracle supports parameterization of explicit cursors. If a `SELECT` statement has to be executed with the same predicates but different values, it is advisable to use parameterized cursors. Parameterization of a cursor is a powerful programming feature as it can improve coding standards by reducing the number of explicit cursor constructs in a program.

Structurally, a parameterized cursor is an explicit cursor with parameters. Parameters may or may not have default values. The developer supplies the parameter values at the time of opening the cursor in the program body. Optionally, you can also strongly prototype a parameterized cursor by specifying `RETURN` clause. The following cursor definition takes the department number as a parameter:

```

/*Cursor to fetch employee details from a department*/
CURSOR CUR_EMP (P_DEPTNO NUMBER)
IS
    SELECT *
    FROM emp
    WHERE deptno = P_DEPTNO;

```

You can also specify default value for the cursor parameters. For example:

```
/*Cursor to fetch employee details from a department*/
CURSOR CUR_EMP (P_DEPTNO NUMBER DEFAULT 10)
IS
    SELECT *
    FROM emp
    WHERE deptno = P_DEPTNO;
```

You can restrict the structure of cursor return type to protect its access.

```
/*Cursor to fetch employee details from a department*/
CURSOR CUR_EMP (P_DEPTNO NUMBER)
RETURN emp%ROWTYPE
IS
    SELECT *
    FROM emp
    WHERE deptno = P_DEPTNO;
```

If you wish to perform a transaction (update or delete) on a cursor result set, you can use the WHERE CURRENT OF clause in a DML statement. The WHERE CURRENT OF clause updates or deletes a the current row of the cursor result set. It is mandatory to declare the cursor with a SELECT FOR UPDATE query to secure a row-level exclusive lock on the cursor result set. The lock is released only after the transaction is committed or rolled back.

For example, the cursor cur_inc_comm in the following PL/SQL block locks the employee records in the cursor result set. The UPDATE statement modifies the employee's commission.

```
DECLARE
    CURSOR cur_inc_comm IS
        SELECT empno, comm
        FROM emp
        FOR UPDATE OF comm;
BEGIN
    FOR i IN cur_inc_comm
    LOOP
        UPDATE emp
        SET comm = comm*1.2
        WHERE CURRENT OF cur_inc_comm;
    END LOOP;
END;
```

Note that you can reproduce the WHERE CURRENT OF scenario by using the ROWID pseudocolumn.

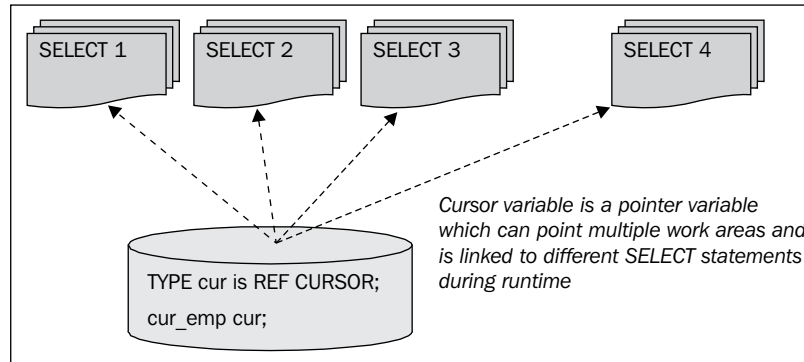
Cursor attributes play a key role in accessing the explicit cursor execution cycle. The attributes are auto-set at each stage and the following table shows the behavioral flow:

Event	%FOUND	%NOTFOUND	%ISOPEN	%ROWCOUNT
Before OPEN	Exception	Exception	FALSE	Exception
After OPEN	NULL	NULL	TRUE	0
Before the first FETCH	NULL	NULL	TRUE	0
After the first FETCH	TRUE	FALSE	TRUE	1
Before the next FETCH	TRUE	FALSE	TRUE	1
After the next FETCH	TRUE	FALSE	TRUE	n + 1
Before the last FETCH	TRUE	FALSE	TRUE	n + 1
After the last FETCH	FALSE	TRUE	TRUE	n + 1
Before CLOSE	FALSE	TRUE	TRUE	n + 1
After CLOSE	Exception	Exception	FALSE	Exception


Cursor variables

A cursor variable enables a cursor handler to be associated with multiple SQL queries. With respect to functionality, it is similar to an explicit cursor but with certain implementation changes. One of the fundamental differences is that, unlike a cursor, it is a variable of a cursor type. Therefore, it can potentially be referenced in a similar way to other program variables.

As a variable, it can be passed as a parameter to subprograms or used as a return type of a PL/SQL function. Cursor variables can be quite handy when sharing result sets between two subprograms or when a client pulls a data set from the database.



Cursor variables are created by defining a variable of the `REF CURSOR` type variable or an `SYS_REFCURSOR` type variable.

[ Cursor FOR loop does not support cursor variables]

The `REF CURSOR` syntax is as follows:

```
TYPE [CURSOR VARIABLE NAME] IS REF CURSOR [RETURN (return type)]
```

In the preceding syntax, the `RETURN` type of a cursor variable must be a record type. It is required in strong `ref cursors` to fix the return type of the result set.

In the following example, the PL/SQL block declares a `ref cursor` as a cursor type and a subsequent cursor variable. We will open the cursor variables for different `SELECT` statements in separate execution cycles.

```
/*Enable the SERVEROUTPUT parameter to print the results*/
SET SERVEROUTPUT ON
DECLARE
/*Declare a REF cursor type*/
  TYPE C_REF IS REF CURSOR;
/*Declare a Cursor variable of REF cursor type*/
  CUR C_REF;
  l_ename emp.ename%TYPE;
  l_sal emp.sal%TYPE;
  l_deptno dept.deptno%TYPE;
  l_dname dept.dname%TYPE;
```

```

BEGIN
/*Open the cursor variable for first SELECT statement*/
  OPEN cur FOR
    SELECT ename, sal
    FROM emp
    WHERE ename='JAMES';
  FETCH cur INTO l_ename, l_sal;
  CLOSE cur;
  DBMS_OUTPUT.PUT_LINE('Salary of '||l_ename||' is '||l_sal);

/*Reopen the cursor variable for second SELECT statement*/
  OPEN cur FOR
    SELECT deptno, dname
    FROM dept
    WHERE loc='DALLAS';
  FETCH cur INTO l_deptno, l_dname;
  CLOSE cur;

  DBMS_OUTPUT.PUT_LINE('Department name '||l_dname||' for '||l_
deptno);
END;
/

Salary of JAMES is 950
Department name RESEARCH for 20

PL/SQL procedure successfully completed.

```

Strong and weak ref cursor types

A `REF CURSOR` can be typed either strong or weak.

A `REF CURSOR` is strong if its return type is fixed at the time of declaration. The `RETURN` clause is used to specify the record type. A strong `ref cursor` can be opened for a `SELECT` statement, which returns the specified record type.

For example, a strong `ref cursor` having the return type record structure of the `employees` table:

```
TYPE c_strong_rf IS REF CURSOR RETURN emp%ROWTYPE;
```


A user-defined record can be declared and assigned as the return type of a strong ref cursor. The following PL/SQL declares a local record and cursor variable of the REF CURSOR type.

```
/*Demonstrate the strong ref cursor where type is a local record
structure*/
DECLARE
    TYPE myrec IS RECORD
        (myname VARCHAR2(10),
         myclass VARCHAR2(10));
    TYPE mycur IS REF CURSOR RETURN myrec;
    cur_var mycur;
```

A REF CURSOR without a return type makes it weak and SELECT statements with a different number of projected columns can be associated with it.

The cursor attributes of a cursor variable are same as those of an explicit cursor.

Working with cursor variables

By now, you may have realized that the execution cycle of a cursor variable is the same as that of an explicit cursor. Once opened for a SELECT query, the records can be fetched before the cursor is closed.

The following PL/SQL block declares a strong cursor variable that returns a record of EMP record type, but the cursor variable is opened for a different record structure (DEPT record type). The block fails to compile and raises a PLS exception.

```
/*Enable the SERVEROUTPUT parameter to print the results*/
SET SERVEROUT ON
/*Demonstrate the usage of cursor variable*/
DECLARE

/*Declare the local variables*/
    l_emp_details emp%ROWTYPE;
    l_row_num number;
    l_random_str varchar2(20);

/*Declare a ref cursor and its variable*/
    TYPE c_type IS REF CURSOR RETURN emp%ROWTYPE;
    cur_var c_type;
BEGIN

/*Open the cursor for SELECT query on DEPT table*/
    OPEN cur_var FOR
```

```

        SELECT *
        FROM dept;

/*Iterate the result set to display the fetch count*/
LOOP
    FETCH cur_var INTO l_emp_details;
    EXIT WHEN cur_var%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('Display results = ' || cur_var%rowcount);
END LOOP;

/*Close the cursor variable*/
CLOSE cur_var;
END;
/

        SELECT *
        *
ERROR at line 16:
ORA-06550: line 16, column 5:
PLS-00382: expression is of wrong type
ORA-06550: line 15, column 3:
PL/SQL: SQL statement ignored.

```

If the cursor variable is opened for a query whose record structure has fewer attributes than the ref cursor's return record, it raises the preceding exception. If the latter has more, it raises PLS-00394: wrong number of values in the INTO list of a FETCH statement.

SYS_REFCURSOR

SYS_REFCURSOR is an Oracle built-in cursor variable data type that declares a weak REF CURSOR variable without declaring the ref pointer type. The generic cursor variable is extensively used when passing a cursor variable as a parameter in stored subprograms with the return type of a PL/SQL function.

SYS_REFCURSOR acts as a cursor variable type in the following syntax:

```

DECLARE
    [Cursor variable name] SYS_REFCURSOR;

```

You can use SYS_REFCURSOR as parameter type in Oracle subprograms shown below:

```

PROCEDURE P_DEMO (P_DATA OUT SYS_REFCURSOR)
IS
    ...
    ...
END;

```

Cursor variables as arguments

A cursor variable can be passed as a formal parameter to a PL/SQL subprogram. Subprograms can share the pointer variable to access the result sets between them.

The following procedure accepts the department number as an input and displays the salary line-graph sorted by job codes:

```
/*Procedure using cursor variable as formal parameter*/
CREATE OR REPLACE PROCEDURE
p_sal_graph (p_dept NUMBER, p_emp_data OUT SYS_REFCURSOR)
IS

/*Declare a local ref cursor variable*/
    TYPE cur_emp IS REF CURSOR;
    cur_sal cur_emp;
BEGIN

/*Open the local ref cursor variable for the SELECT query*/
    OPEN cur_sal FOR
        SELECT empno, job, LPAD('*',sal/100,'.') graph
        FROM emp
        WHERE deptno=P_DEPT
        ORDER BY job;

/*Assign the cursor OUT parameter with the local cursor variable*/
    p_emp_data := cur_sal;
END;
/
```

Procedure created.

```
/*Declare a host cursor variable in SQL* PLUS*/
VARIABLE M_EMP_SAL REFCURSOR;

/*Execute the procedure P_SAL_GRAPH */
EXEC P_SAL_GRAPH(30, :M_EMP_SAL);
```

PL/SQL procedure successfully completed.

```
/*Print the host cursor variable*/
PRINT M_EMP_SAL
```

```
      EMPNO JOB              GRAPH
-----
-----
```

```

7900 CLERK      .....*
7698 MANAGER   .....*
7844 SALESMAN  .....*
7521 SALESMAN  .....*
7499 SALESMAN  .....*
7654 SALESMAN  .....*

```

6 rows selected.

Cursor variables – restrictions

The following list shows the restrictions on the usage of cursor variables:

- Cursor variables cannot be declared as the public construct in a package specification
- Cursor variables cannot be shared remotely across servers
- Cursor variables cannot be opened for a `SELECT FOR UPDATE` query
- Cursor variables are not physically stored in the Oracle Database
- Cursor variables cannot be assigned to `NULL`

Cursor design considerations

The factors that can impact the cursor design are as follows:

- **Implicit versus Explicit cursor:** If a `SELECT` query is confirmed to return only one record, it should be used as `SELECT . . . INTO`, thus making an implicit cursor. Another consideration could be cursor re-usability as implicit cursors are faster than explicit cursors.
- **Use Parameterized cursors:** Explicit cursor design depends on whether or not a cursor will be reused in a PL/SQL block. If the cursor query is expected to be re-run for similar predicates but different input values, it can be made parameterized. Parameterized cursors enhance the reusability of a cursor.
 - The query in the following cursor definition filters employee records by their hire date:

```

CURSOR cur IS
SELECT ename, deptno
FROM emp
WHERE hiredate < TO_DATE('01-01-1985','DD-MM-YYYY');

```

- If the query stands to be reused within the same program, it can be parameterized:

```
CURSOR cur (p_date DATE) IS
SELECT ename, deptno
FROM emp
WHERE hiredate < p_date;
```

- **Usage of cursor variables:** A cursor variable of `REF CURSOR` can be dynamically associated with multiple SQL queries.

Cursor design–guidelines

Here are some of the best practices that can be followed during application development to make the best use of cursors.

- Use a parameterized cursor if the explicit cursor has to be opened multiple times for different input values.
- You should follow the complete execution cycle of the cursor. An explicit cursor must be opened, fetched, and closed. If it is not closed, the cursor resources (data structures) are not cleared from the UGA, until the block execution is over.
- Except for `%ISOPEN`, all the cursor attributes must be referenced within the cursor execution cycle. It also holds true for implicit cursors.
- Use of `%ROWTYPE` must be encouraged to fetch a record from the cursor result set. It not only reduces the overhead of maintaining multiple local variables, but it also inherits the structure of the `SELECT` column list. For example, consider the following code snippet:

```
/*Cursor to select employees with its annual salary*/
CURSOR cur_dept IS
    SELECT ename, deptno, (sal*12) annual_sal
    FROM emp;

l_cur_dept cur_dept%ROWTYPE;
```

Note that the columns that are created virtually for calculative purposes must have an alias name for reference through the record variable.

- A cursor `FOR` loop associates a cursor with the `FOR` loop construct. It is a powerful feature in PL/SQL to simplify and enhance code writing techniques. It implicitly takes care of all the stages of cursor execution such as `OPEN`, `FETCH`, and `CLOSE`.

```
/*Demonstrate working with cursor FOR loop*/
DECLARE
CURSOR cur_dept IS
    SELECT ename, deptno
    FROM emp;
BEGIN
    FOR c IN cur_dept
    LOOP
        ...
    END LOOP;
END;
```

Implicit statement results in Oracle Database 12c

Oracle Database 12c allows a stored subprogram to return a result set implicitly using the `DBMS_SQL` package, and not just through the `REF CURSOR` variable. The new functionality is designed to ease the migration of non-Oracle application programs to Oracle.

Prior to this enhancement in Oracle Database 12c, the only way a PL/SQL stored subprogram could share a result set was through `OUT REF CURSOR` parameters. Later, parameter binding was required at the client end to retrieve the result sets.

The cursor is returned to the calling environment using new overloaded subprograms: `RETURN_RESULT` and `GET_NEXT_RESULT`. The `GET_NEXT_RESULT` can be used if the cursor query returns multiple result sets. The prototype for `RETURN_RESULT` is as follows:

```
PROCEDURE RETURN_RESULT (param_res IN OUT SYS_REFCURSOR,
                        to_client IN BOOLEAN DEFAULT TRUE);

PROCEDURE RETURN_RESULT (param_res IN OUT INTEGER,
                        to_client IN BOOLEAN DEFAULT TRUE);
```

The `param_res` parameter takes either the variable of `SYS_REFCURSOR` type or cursor id, which can be retrieved from `DBMS_SQL.OPEN_CURSOR`. The `to_client` parameter determines if the result set can be returned to the client program or calling subprogram.

Let us re-create the procedure P_SAL_GRAPH (created earlier) using the enhancement.

```
/*Create procedure to implicitly return the result set*/
CREATE OR REPLACE PROCEDURE p_sal_graph_12c (p_dept IN NUMBER)
AS

/*Declare a SYS_REFCURSOR variable*/
cur_sal SYS_REFCURSOR;
BEGIN

/*Open the cursor for department*/
OPEN cur_sal FOR
SELECT empno, job, LPAD('*',sal/100,'.') graph
FROM emp
WHERE deptno=P_DEPT
ORDER BY job;

/*Use DBMS_SQL.RETURN_RESULT to return the cursor*/
DBMS_SQL.RETURN_RESULT(cur_sal);
END;
/
```

Now we will invoke this procedure for department 30 in SQL* Plus. We get identical results with much reduced efforts:

```
/*Execute the procedure for department id 30*/
EXEC p_sal_graph_12c (30);
```

PL/SQL procedure is successfully completed.

ResultSet #1

EMPNO	JOB	GRAPH
7900	CLERK*
7698	MANAGER*
7844	SALESMAN*
7521	SALESMAN*
7499	SALESMAN*
7654	SALESMAN*

6 rows selected.

Subtypes

A **subtype** is a data type that gets evolved from an existing scalar data type. The purpose of creating subtypes, though not mandatory, is to customize the primary data types by controlling certain features such as nullability, range, or sign. An unconstrained subtype is often used in place of primary data types to maintain application standards.

The subtype inherits the behavior of its parent base type and extends it further by a distinguishing feature. For example, `NATURALN` is a subtype of `BINARY_INTEGER`, which prevents the entry of nulls and negative values. Similarly, `SIGNTYPE` permits only three fixed values: -1, 0, or 1.

The following table shows the base types and subtypes under each scalar data type:

Number	Character	Date/Time	Boolean
NUMBER	VARCHAR	DATE	BOOLEAN
DECIMAL/DEC	VARCHAR2	INTERVAL	
DOUBLE PRECISION	NVARCHAR2	TIMESTAMP	
FLOAT	CHAR		
INTEGER/INT	NCHAR		
NUMERIC	CHARACTER		
REAL	LONG		
SMALLINT	LONG RAW		
PLS_INTEGER	RAW		
BINARY_DOUBLE	ROWID		
BINARY_FLOAT	STRING		
BINARY_INTEGER	UROWID		
POSITIVE			
POSITIVEN			
NATURAL			
NATURALN			
SIGNTYPE			

Subtype classification

Subtypes can be predefined or user-defined. Pre-defined subtypes are system-built and maintained in the `STANDARD` package by Oracle Database. Here is a small extract from the `STANDARD` package listing the subtypes of the `NUMBER` family:

```
/*NUMBER family from STANDARD package*/
type NUMBER is NUMBER_BASE;
```



```
subtype FLOAT is NUMBER;
subtype INTEGER is NUMBER(38,0);
subtype INT is INTEGER;
subtype SMALLINT is NUMBER(38,0);
subtype DECIMAL is NUMBER(38,0);
subtype NUMERIC is DECIMAL;
subtype DEC is DECIMAL;
subtype BINARY_INTEGER is INTEGER range '-2147483647'..2147483647;
subtype NATURAL is BINARY_INTEGER range 0..2147483647;
subtype NATURALN is NATURAL not null;
subtype POSITIVE is BINARY_INTEGER range 1..2147483647;
subtype POSITIVEN is POSITIVE not null;
subtype SIGNTYPE is BINARY_INTEGER range '-1'..1;
```



FLOAT is an unconstrained subtype of NUMBER. Constrained subtypes such as NATURAL and NATURALN work mostly on ranges and nullability.

User-defined subtypes are created on top of predefined types with a specific manipulation. They are defined in the `DECLARE` section of a PL/SQL block or subprogram:

```
SUBTYPE [SUBTYPE NAME] IS [PREDEFINED TYPE] [CONSTRAINT | RANGE (range
specification)]
```

The following PL/SQL block declares a subtype of the `NUMBER` base type that has been constrained in the range of 1 to 10. If a variable of subtype datatype is assigned an out-of-range value, the `VALUE_ERROR` exception is raised.

```
DECLARE
/*Create a subtype with value range between 1 to 10*/
    SUBTYPE ID IS BINARY_INTEGER RANGE 1..10;
    L_NUM ID;
BEGIN
/*Assign a value beyond range*/
    L_NUM := 11;
END;
/
DECLARE
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error
ORA-06512: at line 5
```

Type compatibility with subtypes

Subtypes are interchangeable with their base types as long as the subtype definition is not violated. In the following program, the SUBTYPE ID is a BINARY_INTEGER with an assigned range between 1 and 10. The program raises the VALUE_ERROR exception if an out-of-range value is assigned to the subtype variable:

```
DECLARE
/*Create a subtype with value range between 1 to 10. Declare the
subtype variable*/
    SUBTYPE ID IS binary_integer range 1..10;
    L_NUM ID ;
    L_BN BINARY_INTEGER;
BEGIN
/*Assign a NUMBER variable to SUBTYPE variable*/
    L_NUM := 4;
    L_BN := 15;
    L_NUM := L_BN;
END;
/
DECLARE
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error
ORA-06512: at line 8
```

Summary

In this chapter, we discussed the importance of cursor structures in PL/SQL code. We covered the working of a cursor, the execution cycle, design considerations, and guidelines. The usage of cursors can be imperative while developing PL/SQL applications.

In the next chapter, we shall cover composite data types, and you will understand how collections can boost PL/SQL code performance and perform data caching.

Practice exercise

- What are the possible reasons that cause the `INVALID_CURSOR` exception to occur?
 1. Cursor result set has not been fetched.
 2. The cursor does not have parameters.
 3. The value of the `%ROWCOUNT` attribute has been referenced after closing the cursor.
 4. Cursor result set has been fetched into a non matching variable.
- Identify the guidelines to be considered when designing cursors in a PL/SQL block:
 1. Explicit cursors must be used irrespective of the number of records returned by the query.
 2. Cursor `FOR` loops must be used as it implicitly takes care of `OPEN`, `FETCH`, and `CLOSE` stages.
 3. Cursor data must be fetched as a record.
 4. Use `ROWNUM` to index the records in the cursor result sets.
- While processing DMLs as implicit cursors in a PL/SQL executable block, implicit cursor attributes can be used anywhere in the block.
 1. True.
 2. False.
- From the following options, identify the two correct statements about the `REF CURSOR` type?
 1. Ref cursors are reference pointers to cursor objects.
 2. `REF CURSOR` types can be declared in the package specification.
 3. `SYS_REFCURSOR` is a strong ref cursor type.
 4. A cursor variable cannot be used as arguments in stored subprograms.
- The `RETURN` type for a ref cursor can be declared using `%TYPE`, `%ROWTYPE`, or a user-defined record.
 1. True.
 2. False.

- Which two statements, among the following, are true about cursor variables?
 1. Cursor variables can process more than one `SELECT` statement.
 2. A cursor variable can be passed as program arguments across subprograms and even to the client end programs.
 3. A cursor variable can be declared as a public construct in the package specification.
 4. Cursor variables can be stored in a database as database columns.
- Similar to static explicit cursors, cursor variables can also be opened in the `FOR` loop
 1. True.
 2. False.
- Which of the following is true while creating subtypes from a table record structure?

```
SUBTYPE [Name] IS [TABLE] %ROWTYPE
```

 1. The subtype inherits complete column structure of the record structure.
 2. The subtype inherits the default values of the database columns in table.
 3. The subtype inherits the index information of the database columns.
 4. The subtype inherits none except the `NOT NULL` constraint information of the database columns.

4

Using Collections

A collection is a single-dimensional structure of homogeneous elements. Behaviorally speaking, it is quite similar to an array and a list structure available in other third generation languages. First introduced in Oracle 7 as PL/SQL tables, **Oracle 8i** rebranded collections as Index-by tables. Oracle 8i also introduced persistent collection types, namely nested tables and varrays. Oracle Database 9i renamed Index-by-tables to associative arrays.

Oracle Database offers a wide scope of usability of collections in PL/SQL programming. The language semantics not only allow you to create and maintain collections, but also provides multiple methods for array operations. This chapter helps you to understand the collection types in Oracle and, most importantly, which types suit a given problem. The chapter outline looks like this:

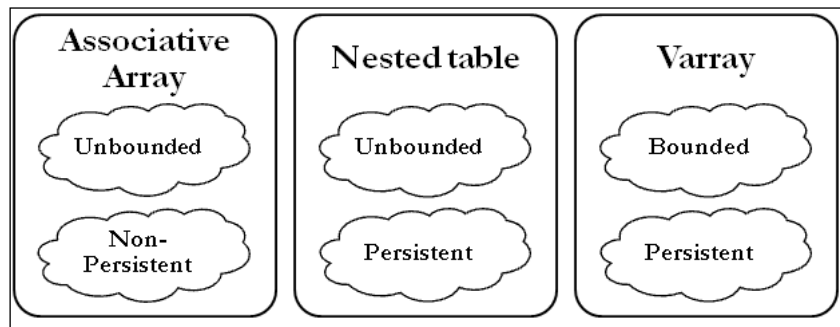
- An introduction to collections
 - Categorization
 - Selection of an appropriate collection type
- Associative arrays
- Nested tables
- Varrays
- PL/SQL collection methods
- Manipulating collection elements
- Collection design considerations

Introduction to collections

A collection is an array like homogeneous single-dimensional structure, which holds a set of elements of similar data type. Each cell in a collection, and hence, each element, is uniquely identified by its position index or the subscript. The element or the value contained in a cell can be of a valid SQL data type or a user-defined type. There are three types of collections: associative array; nested table; and varray.

A collection can be bounded or unbounded on the basis of it's collection size. Bounded collections have a fixed number of elements, as in arrays. Unbounded collections can have any number of elements. Varrays are bounded collections while the associative arrays and nested tables are unbounded.

On the basis of persistence in the database, a collection can be either a **persistent** or **non-persistent** collection. A persistent or SQL collection can either be created in the Oracle Database schema or declared within the PL/SQL block. A PL/SQL or non-persistent collection is declared in a PL/SQL block only. A nested table and a varray are persistent collections. An associative array is always a non-persistent collection.



Collection types

An **associative array** is a non-persistent unbounded collection which means that an associative array cannot be created in the Oracle Database schema, but locally declared within the declarative section of a PL/SQL block.

A **nested table** is a persistent collection of homogeneous elements that can be created in a database as a schema object as well as declared within a PL/SQL block. A nested table being an unbounded collection, has no limit on the number of elements.

A **varray** is a single-dimensional homogeneous collection that can be created in a database as well as in PL/SQL. Being a bounded collection, it can hold only a fixed number of elements.

Associative arrays

Associative arrays in Oracle are similar to conventional lists in other programming languages. An associative array is an unbounded array of cells and always defined in the declarative section of a PL/SQL program. While a cell is identified by an index of number or string type, it can hold a value of a scalar data type or user-defined composite type.

The syntax to declare an associative array in a PL/SQL block is as follows:

```
TYPE [COLL NAME] IS TABLE OF [ELEMENT DATA TYPE] NOT NULL
    INDEX BY [INDEX DATA TYPE]
```

In the syntax, the [INDEX DATA TYPE] signifies the data type of an array subscript. It can be BINARY_INTEGER, PLS_INTEGER, POSITIVE, NATURAL, SIGNTYPE, or VARCHAR2. The data types that are not supported as index types are RAW, NUMBER, LONG-RAW, ROWID, and CHAR.

The [ELEMENT DATA TYPE] can be one of the following:

- **PL/SQL scalar data type:** NUMBER (along with its subtypes), VARCHAR2 (and its subtypes), DATE, BLOB, CLOB, or BOOLEAN
- **Inferred data:** The data type inherited from a table column, cursor expression or predefined package variable
- **User-defined type:** A user-defined object type or collection type

Here are some sample declarations of an associative array:

```
/*Associative array of CLOB indexed by a number*/
TYPE clob_t IS TABLE OF CLOB INDEX BY PLS_INTEGER;

/*Array of employee ids indexed by the employee names*/
TYPE empno_t IS TABLE OF emp.empno%TYPE NOT NULL
    INDEX BY emp.ename%type;
```


The following PL/SQL program declares an associative array of some elements with a string type subscript. The `FIRST` and `NEXT` collection methods are discussed in detail in the *PL/SQL collection methods* section in this chapter:

```
/*Enable the SERVEROUTPUT on to display the output*/
SET SERVEROUTPUT ON

/*Start the PL/SQL block*/
DECLARE

/*Declare an associative array and a local variable of collection
type*/
    TYPE string_asc_arr_t IS TABLE OF NUMBER
    INDEX BY VARCHAR2(10);
    l_str string_asc_arr_t;
    l_idx VARCHAR2(10);
BEGIN

/*Assign the total days in each quarter*/
    l_str ('JAN-MAR') := 90;
    l_str ('APR-JUN') := 91;
    l_str ('JUL-SEP') := 92;
    l_str ('OCT-DEC') := 93;
    l_idx := l_str.FIRST;

    WHILE (l_idx IS NOT NULL)
    LOOP
        DBMS_OUTPUT.PUT_LINE('Value at ' || l_idx || ' is ' || l_str(l_idx));
        l_idx := l_str.NEXT(l_idx);
    END LOOP;
END;
/

Value at APR-JUN is 91
Value at JAN-MAR is 90
Value at JUL-SEP is 92
Value at OCT-DEC is 93
```

PL/SQL procedure successfully completed.

Similarly, the following PL/SQL block populates an associative array of string values with random values:

```
/*Enable the SERVEROUTPUT on to display the output*/
SET SERVEROUTPUT ON

/*Start the PL/SQL Block*/
```

```

DECLARE

/*Declare an array of string indexed by numeric subscripts*/
TYPE random_t IS TABLE OF VARCHAR2(12) INDEX BY PLS_INTEGER;
l_random random_t;

BEGIN
/*Insert the values using a FOR loop*/
FOR I IN 1..100
LOOP
    l_random(I) := DBMS_RANDOM.STRING('X', 10);
END LOOP;

/*Display the values randomly*/
DBMS_OUTPUT.PUT_LINE(l_random(5));
DBMS_OUTPUT.PUT_LINE(l_random(10));
DBMS_OUTPUT.PUT_LINE(l_random(50));
END;
/

5NHU5R5UMU
98P7NUDZZB
E6AG6TR07S

PL/SQL procedure successfully completed.

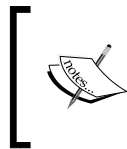
```

The features of associative arrays are as follows:

- An associative array can be sparse, meaning the index need not be consecutive.
- Being a non-persistent collection, it cannot participate in DML transactions. However, elements of an associative array can be used in the DML within a `FORALL` statement.
- It can be passed as an argument to other local subprograms.
- Sorting of an associative array depends on the `NLS_SORT` parameter.
- An associative array declared in a package specification behaves as a persistent array in a session.

Nested tables

A nested table is a persistent SQL collection that is used as a list to hold elements of the same data type. It can also be created in the database and defined in a PL/SQL program. It is an unbounded collection and the user doesn't have to maintain the cell index. Oracle automatically assigns the cell index as 1 to the first cell and moves onwards incrementally. A nested table collection type initially starts off as a dense collection but it becomes sparse due to delete operations.



A dense collection refers to a collection that is tightly populated which means no empty cells between the lower and upper indexes of the collection. A dense collection may become sparse by performing delete operations.

When a nested table is created as a schema object in the database, it can be referenced in a PL/SQL block as a variable parameter. A column of nested table types can be included in a table. An attribute of nested table types can exist in an object type. In a database schema, a nested table can be declared using the `CREATE TYPE . . .` statement:

```
CREATE [OR REPLACE] TYPE type_name IS TABLE OF [element_type] [NOT
NULL] ;
/
```

In the preceding syntax, `[element_type]` can be an SQL-supported scalar data type, a database object type, or a REF object type. The unsupported element types are `BOOLEAN`, `LONG`, `LONG-RAW`, `NATURAL`, `NATURALN`, `POSITIVE`, `POSITIVEN`, `REF CURSOR`, `SIGNTYPE`, `STRING`, `PLS_INTEGER`, `SIMPLE_INTEGER`, `BINARY_INTEGER` and all other non-SQL supported data types.

When a nested table is declared in a PL/SQL program, it behaves as a one-dimensional array without any index type or upper limit specification. It can be referenced only within the program in which it is declared. A table type definition in a PL/SQL block follows the below syntax:

```
DECLARE
TYPE type_name IS TABLE OF element_type [NOT NULL] ;
```

The `element_type` is a primitive data type (except cursor variables) or a user-defined type.

Modify and drop a nested table object type

To increase the precision of nested table elements, use the `ALTER TYPE` command with `CASCADE` or `INVALIDATE` options:

```
ALTER TYPE [type name] MODIFY ELEMENT TYPE [modified element type]
[CASCADE | INVALIDATE];
```

The `CASCADE` and `INVALIDATE` options determine whether the collection alteration has to cascade or invalidate the dependents.

The nested table type can be dropped using the `DROP` command, as shown in the following syntax (note that the `FORCE` keyword drops the type irrespective of its dependents):

```
DROP TYPE [collection name] [FORCE]
```

Design considerations of a nested table

As a database developer, you must understand the factors that influence nested table design and layout. For instance, retrieval of an out-of-line stored nested table structure along with the table columns could potentially impact the query design. This section talks about key aspects that should be considered during nested table design.

Nested table storage

Oracle maintains a separate storage table to store the nested table data. This storage table is integrally connected to the parent table through a system-generated 16-byte value known as `NESTED_TABLE_ID`. This value is used to correlate the parent row value and the connected storage table rows.

Nested table in an index - organized table

A nested table with a primary key behaves as an **index-organized table (IOT)**. If the primary key is constituted with `NESTED_TABLE_ID`, the parent and child rows are clustered together, which helps in fast retrieval of child rows. In addition, nested table compression can be enabled using the `COMPRESS` clause. A nested table in compressed mode prevents the repetition of a parent key for a set of child rows in the storage table, thus optimizing the space consumption. This format of organization in nested tables can be beneficial when the application retrieves a nested table as a complete unit.

Alternatively, indexing `NESTED_TABLE_ID` in the storage table can also accelerate the access and retrieval of child rows.

Nested table locators

Nested table locators are useful when a parent row correlates to a large set of child rows. Instead of performing joins and returning large data sets, Oracle retrieves a locator that can be used to access the child records whenever required. There are two ways to fetch the collection type as a location:

- Specify RETURN AS LOCATOR in the storage clause
- Use the NESTED_TABLE_GET_REFS hint in the SELECT statement

Nested table as the schema object

The following example demonstrates the creation of a nested table type in the database and using it as a column data type in a table. The object type NT_SCORES records the scores of a batsman in a cricket tournament:

```
/*Create the nested table in the database*/
CREATE TYPE nt_scores AS TABLE OF NUMBER;
/
```

Type created.

The metadata information for NT_SCORES is available in the USER_TYPES and USER_COLL_TYPES dictionary views:

```
/*Select query to get nested table metadata as collection type*/
SELECT type_name, typecode, type_oid
FROM USER_TYPES
WHERE type_name = 'NT_SCORES'
/
```

TYPE_NAME	TYPECODE	TYPE_OID
NT_SCORES	COLLECTION	1218C13A817D0D9FE0530F02000A6119

Note that the TYPECODE value shows the type of the object in the database and differentiates collection types from user-defined object types. Let's now query the USER_COLL_TYPES dictionary view to query the collection type and element type of the nested table:

```
SELECT type_name, coll_type, elem_type_name
FROM user_coll_types
WHERE type_name = 'NT_SCORES'
/
```

TYPE_NAME	COLL_TYPE	ELEM_TYPE_NAME
-----	-----	-----
NT_SCORES	TABLE	NUMBER

The `CREATE TABLE` statement creates a table with a column of the `NT_SCORES` table type. Although a nested table type column in a table resembles a table within a table, Oracle creates a separate storage table from the parent table. The `NESTED TABLE [Column] STORE AS [storage table]` clause specifies the storage table for the nested table type column. At the time of record creation, the data in the storage table is connected to the row in the parent table through a row identifier:

```
CREATE TABLE t_bat_scores
(name  VARCHAR2 (30),
 pos   NUMBER,
 score NT_SCORES)
NESTED TABLE score STORE AS nt_scores_st
/
```

Table created.

By default, the storage options for the storage table are the same as that of the main table. If you want the storage table on a different tablespace, you can specify the `TABLESPACE` clause alongside the `NESTED TABLE` clause:

```
NESTED TABLE score STORE AS nt_scores_st
(TABLESPACE nt_tbs_storage)
/
```

If you move the parent table to a different tablespace using the `ALTER TABLE . . . MOVE` statement, then the storage table doesn't move unless you explicitly move the storage table as follows:

```
ALTER TABLE nt_scores_st MOVE TABLESPACE nt_tbs_storage
/
```

Operations on a nested table type column

In terms of data contained in a column, the nested table type column stores the object type instance, instead of scalar data values. The structure of the object type instance must be noted while inserting the records in the table and querying the data from the table.

This section takes you through the different operations on the nested table type column and the object type instance.

Create a nested table instance

Let's insert the runs scored by the first two batsmen of the team in the last five innings. Note the object type instance can be inserted in a nested table type column using a collection type constructor. It is a default constructor with the same name as the collection and all member attributes:

```
INSERT INTO t_bat_scores (name, pos, score) VALUES
('Duckworth', 1, nt_scores(115, 37))
/
INSERT INTO t_bat_scores (name, pos, score) VALUES
('Duckworth', 2, nt_scores(71, 29, 13))
/
INSERT INTO t_bat_scores (name, pos, score) VALUES
('Lewis', 1, nt_scores(34, 65, 23))
/
INSERT INTO t_bat_scores (name, pos, score) VALUES
('Lewis', 2, nt_scores(0, 1))
/
commit
/
```

Querying the table data, you can see the nested values that is the instances of nested tables:

```
SELECT *
FROM t_bat_scores
/
```

NAME	POS	SCORE
Duckworth	1	NT_SCORES(115, 37)
Duckworth	2	NT_SCORES(71, 29, 13)
Lewis	1	NT_SCORES(34, 65, 23)
Lewis	2	NT_SCORES(0, 1)

In the next innings, Duckworth opens at No. 1 position and scores 125, while Lewis scores 45 at No. 2 position. You would want to update the existing object instances and not create a duplicate entry for the batsmen. Oracle allows the piecewise inserts and updates on the object instances. The piecewise score is inserted using the following INSERT statement:

```
INSERT INTO TABLE(SELECT score
FROM t_bat_scores
WHERE name = 'Duckworth'
AND pos=1)
```

```

VALUES (125)
/
INSERT INTO TABLE(SELECT score
                    FROM t_bat_scores
                    WHERE name = 'Lewis'
                    AND pos=2)

VALUES (45)
/
COMMIT
/

```

After the new scores are updated in the table, the object instances look like this:

```

SELECT *
FROM t_bat_scores
/

NAME                POS SCORE
-----
Duckworth            1 NT_SCORES(115, 37, 125)
Duckworth            2 NT_SCORES(71, 29, 13)
Lewis                1 NT_SCORES(34, 65, 23)
Lewis                2 NT_SCORES(0, 1, 45)

```

Due to a short run, Duckworth loses a run and his score is now 124. This change has to reflect back in the table. The following UPDATE statement helps you perform piecewise updates. The following SELECT query verifies the value update:

```

UPDATE TABLE (SELECT score
                FROM t_bat_scores
                WHERE name = 'Duckworth' and pos=1) P
SET P.COLUMN_VALUE = 124
WHERE P.COLUMN_VALUE = 125
/
SELECT *
FROM t_bat_scores
/

NAME      POS SCORE
-----
Duckworth 1  NT_SCORES(115, 37, 124)
Duckworth 2  NT_SCORES(71, 29, 13)
Lewis     1  NT_SCORES(34, 65, 23)
Lewis     2  NT_SCORES(0, 1, 45)

```


Querying a nested table column

When a table with a nested table column is queried, the nested table column appears as an instance of nested table object type. We already saw this in action in the last section.

The `TABLE` expression can be used to unnest or open the instance and display the data in relational format. The `TABLE` expression is used to access the attributes of nested table type. Oracle implicitly joins the parent row with the nested table row in the query output:

```
SELECT T.name, T.pos, T1.column_value
FROM t_bat_scores T, TABLE (T.score) T1
/
```

NAME	POS	COLUMN_VALUE
Duckworth	1	115
Duckworth	1	37
Duckworth	1	124
Duckworth	2	71
Duckworth	2	29
Duckworth	2	13
Lewis	1	34
Lewis	1	65
Lewis	1	23
Lewis	2	0
Lewis	2	1
Lewis	2	45

12 rows selected.

In the preceding `SELECT` query, `COLUMN_VALUE` is an Oracle pseudo-column that is used in the `SELECT` queries to signify the nested table column with no attribute name.

Nested table collection type in PL/SQL

In PL/SQL, a nested table can be declared and defined in the declaration section of a PL/SQL block as a local collection type. The PL/SQL variable of nested table type must be initialized before using it in the program body. Oracle raises the exception `ORA-06531: Reference to uninitialized collection if an uninitialized collection type variable is accessed in the program body.`

Collection initialization

When used in a PL/SQL program, the variables of nested table or varray collection types must be initialized. Associative arrays are local non-persistent arrays, so no initialization is required for them. This section has been included to give a brief description of collection initialization methods:

- Initialize a collection in the declarative section by using the default constructor:

```
/*Start the PL/SQL block*/
DECLARE

    TYPE coll_nt_t IS TABLE OF NUMBER;
/*Collection variable initialization using a default constructor*/
    L_LOCAL_VAR1 coll_nt_t := coll_nt_t (10,20);
/*collection variable initialization with an empty collection*/
    L_LOCAL_VAR2 coll_nt_t := coll_nt_t();
BEGIN
    ...
    ...
END;
```

- Initialize a collection in the executable section through assignment or the `SELECT...INTO` statement:

In the executable section, a `SELECT` statement can pull a collection instance (of the same collection type) into the local collection variable. This method of collection initialization is permissible only when the collection type exists as a schema object and is used to declare a PL/SQL variable.

In our earlier illustrations, `nt_scores` is a nested table in the database. Let us use the same type to declare a local collection variable:

```
/*Start the PL/SQL block*/
DECLARE
    l_loc_num nt_scores;
    l_loc_idx nt_scores;
BEGIN
/*Initializing the collection variable with SELECT...INTO*/
    SELECT num INTO l_loc_num
    FROM tab_use_nt_col
    WHERE id = 1;

/*Initializing the collection variable with an assignment*/
    l_loc_idx := l_loc_num;
END;
/
```

The following PL/SQL block declares a nested table in the declarative section. Observe the scope and visibility of the collection variable. Note that the COUNT method has been used to display the array elements:

```
/*Enable the SERVEROUTPUT to display the results*/
SET SERVEROUTPUT ON

/*Start the PL/SQL block*/
DECLARE

/*Declare a local nested table collection type*/
  TYPE nt_local IS TABLE OF NUMBER;
  l_array nt_local := nt_local (10,20,30);

BEGIN
/*Use FOR loop to parse the array and print the elements*/
  FOR I IN 1..l_array.COUNT
  LOOP
    DBMS_OUTPUT.PUT_LINE('Showing ' || i || ' value: ' || l_array(I));
  END LOOP;
END;
/

Showing 1 value: 10
Showing 2 value: 20
Showing 3 value: 30

PL/SQL procedure successfully completed.
```

Querying the nested table metadata

As a collection type, the nested table's metadata can be queried from the USER_TYPES or USER_COLL_TYPES views. For the nested table type column in a table, Oracle's USER_NESTED_TABLES and USER_NESTED_TABLE_COLS data dictionary views maintain the information of the parent and the nested tables. The USER_NESTED_TABLES static view maintains the information about the mapping of a nested table collection type with its parent table.

The following SELECT statement on USER_NESTED_TABLES queries the details of the storage table associated with the parent table:

```
SELECT parent_table_name,
       parent_table_column,
       table_name,
```

```

        table_type_name,
        storage_spec
FROM user_nested_tables
WHERE parent_table_name='T_BAT_SCORES'
/
PARENT_TABLE_NAME  PARENT_TAB  TABLE_NAME      TABLE_TYPE  STORAGE_SPEC
-----
T_BAT_SCORES       SCORE          NT_SCORES_ST  NT_SCORES    DEFAULT

```

Let us now query the nested storage table in the preceding dictionary view to list all its attributes:

```

SELECT COLUMN_NAME, DATA_TYPE, DATA_LENGTH, HIDDEN_COLUMN
FROM user_nested_table_cols
where table_name='NT_SCORES_ST'
/
COLUMN_NAME          DATA_TYP  DATA_LENGTH  HID
-----
NESTED_TABLE_ID      RAW        16           YES
COLUMN_VALUE         NUMBER     22           NO

```

The COLUMN_VALUE attribute is the default pseudo-column of the nested table as there are no named attributes in the collection structure. The other attribute, NESTED_TABLE_ID, is a hidden, unique, 16-byte, system-generated raw hash code, which stores the parent row identifier alongside the nested table instance to distinguish the parent row association. Its value is the same as the system supplied id (SYS_NCXXXX\$) value in the parent table.

Nested table comparison functions

Oracle provides SQL functions that can be applied to nested tables for multiset purposes. The functions are briefly described in the following table:

Collection function	Description
SET	Returns distinct elements of a nested table
CARDINALITY	Returns the count of elements in a nested table
[NOT] SUBMULTISET	Returns TRUE, if a nested table is a subset of the other nested table
POWMULTISET (SQL only)	Generates all possible non-empty sub-multisets from an input collection. Maximum cardinality is 32.
POWMULTISET_BY_CARDINALITY (SQL only)	Generates the non-empty sub-multisets from an input collection, filtered by the given cardinality.

Multiset operations on nested tables

The set operators (`UNION`, `INTERSECT`, and `MINUS`) in SQL are used to combine the results from more than one SQL query into a single result set. Oracle Database 10g introduced multiset operators to allow set operations on nested tables. A multiset operation combines more than one nested tables and returns a collection resulting from a multiset operator.

The multiset operations add a unique and exclusive capability to nested tables. Instead of following the trivial loop-through-collection approach to row-compare two collections for common or difference elements, you can now compare a collection as an object.



Multiset operators are available in PL/SQL, and SQL too.

The multiset operators are listed as below:

- `MULTISET UNION [ALL | DISTINCT]`: combines two nested tables of the same type and returns a nested table containing the elements from both the input collections. You can use `ALL` or `DISTINCT` to allow or prevent duplication of elements. `ALL` is the default.
- `MULTISET INTERSECT [ALL | DISTINCT]`: combines two nested tables of the same type and returns a nested table containing the elements, which are common in the two nested tables. If you use the `DISTINCT` option, Oracle removes the duplicate values from the final result set.
- `MULTISET EXCEPT [ALL | DISTINCT]`: works with two nested tables of the same type and returns a nested table containing the elements, which are present in the first but not in the second. If you use the `DISTINCT` option, Oracle removes the common elements.



For multiset operations, the input nested tables should be of the same type with comparable elements. If you are working with complex collection types that is a nested table with complex attributes, you must use the `MAP` order method to enable sorting of the collection elements. The result of a multiset operation is of the same collection type as the operands.

The following PL/SQL block demonstrates the usage of multiset operators.

```
/*Create a nested table collection object in SCOTT schema */  
CREATE OR REPLACE TYPE list_of_letters AS TABLE OF VARCHAR2 (1);  
/
```

```
/*Enable the serveroutput */
SET SERVEROUTPUT ON

/*Start the PL/SQL block*/
DECLARE

/*Declare two local collection variables*/
str1 list_of_letters := list_of_letters('O','R','A','C','L','E');
str2 list_of_letters := list_of_letters('D','A','T','A','B','A','S','E');

/*Local procedure to print the collection elements */
PROCEDURE print_nt (str IN list_of_letters)
IS
BEGIN

    FOR i in 1..str.COUNT
    LOOP
        DBMS_OUTPUT.PUT (str(i)||'.');
    END LOOP;

    DBMS_OUTPUT.NEW_LINE;
    DBMS_OUTPUT.PUT_LINE(RPAD(' ',50,'-'));
END print_nt;

BEGIN

/*Multiset intersect operation*/
DBMS_OUTPUT.PUT ('MULTISET INTERSECT => ');
PRINT_NT(str2 MULTISET INTERSECT str1 );

/*Multiset intersect distinct operation*/
DBMS_OUTPUT.PUT ('MULTISET INTERSECT DISTINCT => ');
PRINT_NT(str2 MULTISET INTERSECT DISTINCT str1 );

/*Multiset union operation*/
DBMS_OUTPUT.PUT ('MULTISET UNION => ');
PRINT_NT(str1 MULTISET UNION str2 );

/*Multiset union distinct operation*/
DBMS_OUTPUT.PUT ('MULTISET UNION DISTINCT => ');
PRINT_NT(str1 MULTISET UNION DISTINCT str2 );

/*Multiset except operation*/
DBMS_OUTPUT.PUT ('MULTISET EXCEPT => ');
```

```
PRINT_NT(str2 MULTISSET EXCEPT str1 );

/*Multiset except distinct operation*/
DBMS_OUTPUT.PUT ('MULTISSET EXCEPT DISTINCT => ');
PRINT_NT(str2 MULTISSET EXCEPT DISTINCT str1 );
END;
/

MULTISSET INTERSECT => A.E.
-----
MULTISSET INTERSECT DISTINCT => A.E.
-----
MULTISSET UNION => O.R.A.C.L.E.D.A.T.A.B.A.S.E.
-----
MULTISSET UNION DISTINCT => O.R.A.C.L.E.D.T.B.S.
-----
MULTISSET EXCEPT => D.T.A.B.A.S.
-----
MULTISSET EXCEPT DISTINCT => D.T.B.S.
-----

PL/SQL procedure successfully completed.
```

Varray

Oracle Database 8i introduced varrays as a modified format of a nested table. The varray or variable size array is quite similar to nested tables but bounded in nature. The varray declaration includes the count of elements that a varray can accommodate. The minimum varray index is 1, the current size is the total number of elements, and the maximum limit is the varray size. At any moment, the current size cannot exceed the maximum limit. Varrays are appropriately used when you know the maximum number of elements in a collection structure.

Like nested tables, varrays can be created in the database as schema objects as well as in a PL/SQL block. When created in the database as a schema object, varrays can be referenced in PL/SQL program units as variables, parameters and function return types. A table can have a column of a varray type. An object type can have an attribute of a varray type. The syntax for varrays, when defined as a database collection type, is as follows:

```
CREATE [OR REPLACE] TYPE type_name IS {VARRAY | VARYING ARRAY} (size_
limit) OF element_type
```

When a varray type column is included in a table, it is stored in line with the corresponding row in the parent table. Therefore, there is no need to have a separate storage table. If the varray column size exceeds 4 kilobytes, Oracle follows the out-of-line storage mechanism and stores the varray as an LOB.



The in-line storage mechanism of varrays helps Oracle to reduce the disk I/Os.

When declared in a PL/SQL block, a varray can be referenced within the current block only. In PL/SQL, varrays can be declared as follows:

```
DECLARE
  TYPE type_name IS {VARRAY | VARYING ARRAY} (size_limit) OF
    element_type [NOT NULL];
```

In the syntax, `size_limit` determines the maximum count of elements in the array. If the varray size has to be modified after creation, follow this `ALTER TYPE` syntax:

```
ALTER TYPE [varray name] MODIFY LIMIT [new size_limit]
[INVALIDATE | CASCADE];
```



The varray size can only be increased by using the `ALTER TYPE...MODIFY` statement. Even if the current maximum size is not utilized, Oracle doesn't allow the ripping-off of a varray size. If you try to reduce the varray size, Oracle raises the compilation error `PLS-00728: the limit of a VARRAY can only be increased and to a maximum 2147483647`.

The `INVALIDATE` and `CASCADE` options signify the invalidation or propagation effect on the dependent objects as a result of the type alteration.



Use the `DROP` command to drop a varray type from the database:

```
DROP TYPE [varray type name] [FORCE]
```

Varray as a schema object

The following example demonstrates the usage of a varray as a schema object. The varray captures the annual production values recorded by a geography for the years.

Let's create a varray of a composite object type:

```
CREATE OR REPLACE TYPE ot_num_2 AS OBJECT
(str VARCHAR2(25),
 num NUMBER
);
/
CREATE OR REPLACE TYPE v_geo_prod AS VARRAY(5) OF ot_num_2;
/
```

For metadata information, we'll query the USER_TYPES and USER_COLL_TYPES views:

```
SELECT type_name, typecode, type_oid
FROM USER_TYPES
WHERE type_name = 'V_GEO_PROD'
/
```

TYPE_NAME	TYPECODE	TYPE_OID
V_GEO_PROD	COLLECTION	122DC43ECA0412EEE0530F02000AC98A

```
SELECT type_name, coll_type, elem_type_name
FROM user_coll_types
WHERE type_name = 'V_GEO_PROD'
/
```

TYPE_NAME	COLL_TYPE	ELEM_TYPE_NAME
V_GEO_PROD	VARYING ARRAY	OT_NUM_2

The following CREATE TABLE script creates a table using the varray object type. Note that there is no storage table required for varrays:

```
CREATE TABLE t_annual_prod
(prod_code VARCHAR2(8),
 year NUMBER,
 production V_GEO_PROD)
/
```

At this stage, if we need to modify the varray structure that is, to increase the limit size, we can do so using the ALTER TYPE...MODIFY statement. Here is an example that increases the varray limit size to 10:

```
ALTER TYPE v_geo_prod MODIFY LIMIT 10 INVALIDATE;
```

The previous script invalidates all dependent objects. The other option is CASCADE, which cascades the changes to the dependent objects. The CASCADE [NOT] INCLUDING TABLE DATA option controls whether the underlying data image of varray type instances needs to be updated or just the table metadata.

If the size of varray column value exceeds 4000 bytes, it has to be stored as an LOB. For this purpose, you must include the LOB storage information in the varray metadata. We'll create a varray column with LOB storage for demo purpose and drop it later:

```
CREATE TYPE ref_geo_prod AS VARRAY(10) OF ot_num_2;
/
ALTER TABLE t_annual_prod
ADD (gl_prod ref_geo_prod)
VARRAY gl_prod STORE AS LOB prod_lob
/
ALTER TABLE t_annual_prod DROP COLUMN gl_prod
/
```

Operations on varray type columns

Similar to nested tables, the columns of varray store instances of varray object types. However, fundamental differences from nested tables impact certain operations in varrays. One of the major differences is that piecewise operations are not possible with varrays.

Inserting varray collection type instance

The following INSERT statements create two records in the T_ANNUAL_PROD table. The varray instance uses a collection constructor, while the object type instance uses the default object type constructor to provide the input values:

```
INSERT INTO t_annual_prod
VALUES ('PROD-A', 2010, v_geo_prod(ot_num_2('NAS',50),
                                     ot_num_2('EMEA',32),
                                     ot_num_2('APAC',47)))
/
INSERT INTO t_annual_prod
VALUES ('PROD-A', 2011, v_geo_prod(ot_num_2('NAS',54),
                                     ot_num_2('APAC',57),
                                     ot_num_2('EMEA',37)))
/
COMMIT
/
```

With varrays, piecewise inserts are not possible. If you have to add a new varray element, you must follow the varray column atomic update process.

Querying varray column

A varray column can be queried either as an object type instance or in a relational format. Let's check the ways to query data from T_ANNUAL_PROD table (ignore the output formatting issues):

```
set lines 150
col production format a50
SELECT * FROM t_annual_prod
/
```

PROD_COD	YEAR	PRODUCTION(STR, NUM)
PROD-A	2010	V_COUNTRY_PROD(OT_NUM_2('NAS', 50), OT_NUM_2('EMEA', 32), OT_NUM_2('APAC', 47))
PROD-A	2011	V_COUNTRY_PROD(OT_NUM_2('NAS', 54), OT_NUM_2('APAC', 57), OT_NUM_2('EMEA', 37))

The previous result contains the instances of varrays as well as the object type. The following query simply select the object type instances:

```
SELECT T.prod_code, T.year, value(t1) val
FROM t_annual_prod T, TABLE (T.production) T1
/
```

PROD_COD	YEAR	VAL(STR, NUM)
PROD-A	2010	OT_NUM_2('NAS', 50)
PROD-A	2010	OT_NUM_2('EMEA', 32)
PROD-A	2010	OT_NUM_2('APAC', 47)
PROD-A	2011	OT_NUM_2('NAS', 54)
PROD-A	2011	OT_NUM_2('APAC', 57)
PROD-A	2011	OT_NUM_2('EMEA', 37)

Well, this looks good. To unnest all the object values of an instance in a relational format, you can use the TABLE clause. Here is the query:

```
SELECT T.prod_code, T.year, T1.str, T1.num
FROM t_annual_prod T, TABLE (T.production) T1
/
```

PROD_COD	YEAR	STR	NUM
PROD-A	2010	NAS	50
PROD-A	2010	EMEA	32

PROD-A	2010	APAC	47
PROD-A	2011	NAS	54
PROD-A	2011	APAC	57
PROD-A	2011	EMEA	37

6 rows selected.

If the rows exist in the parent table without any child instance, you can specify the (+) sign as you would do with outer joins. For example, the previous query can be rewritten as:

```
SELECT T.prod_code, T.year, T1.str, T1.num
FROM t_annual_prod T, TABLE (T.production) (+) T1
/
```

Updating the varray instance

For varrays, piecewise inserts and updates are not possible. Only atomic changes are allowed, which means that an entire varray instance has to be modified to update a varray element.

The following PL/SQL block updates the production values of EMEA in the year 2010 through a constructor. Note the use of a SQL varray in the PL/SQL block:

```
/*Start the PL/SQL block*/
DECLARE
/*Create a local varray variable and initialize with the new varray
instance*/
    l_new_geoproduct v_geo_prod := v_geo_prod(ot_num_2('NAS',50), ot_
num_2('EMEA',42), ot_num_2('APAC',47));
BEGIN

/*Update the production instance with the local varray variable*/
    UPDATE t_annual_prod
    SET production = l_new_geoproduct
    WHERE prod_code = 'PROD-A'
    AND year=2010;
    COMMIT;
END;
/
```

Varray in PL/SQL

The following PL/SQL shows the local declaration of a varray object in a PL/SQL block:

```
/*Enable the SERVEROUTPUT to display the results*/
SET SERVEROUTPUT ON

/*Start the PL/SQL block*/
DECLARE
/*Declare a local varray type, define collection variable and
initialize it*/
    TYPE local_varray IS VARRAY(4) OF VARCHAR2(100);
    l_obj local_varray := local_varray('ORCL','SAP','IBM');

BEGIN
/*Iterate the array object to print the varray elements*/
    FOR I IN 1..l_obj.COUNT
    LOOP
        DBMS_OUTPUT.PUT_LINE('Varray elements: ' || l_obj(I));
    END LOOP;
END;
/

Varray elements:ORCL
Varray elements:SAP
Varray elements:IBM

PL/SQL procedure successfully completed.
```

Comparing the collection types

The following table compares the collection types based on the considerations.

Comparing factor	Nested table	Varray	Associative array
Maximum size	Unbound and grows dynamically.	Bounded. However, varray size, can only be increased thereafter.	Unbound and grows dynamically.
Sparsity	Starts dense but may become sparse due to deletions.	Always dense.	Can be sparse.

Comparing factor	Nested table	Varray	Associative array
Storage considerations	Out-of-line storage in a separate storage table.	In-line storage up to 4000 bytes. For out-of-line storage, the LOB clause must be specified.	Non-persistent collection uses program memory that is UGA.
Querying ability	A nested table type column can be queried as an instance. However, unnesting of an instance is possible using the TABLE or CAST clause.	A varray type column can be queried as an instance. However, unnesting of an instance is possible using the TABLE or CAST clause.	Non-persistent collection.
DML operations	Piecewise and atomic operations possible.	Only atomic operations are possible.	Only atomic operations allowed within a PL/SQL block.
Ordering	Unordered index	Retains the index order.	Retains the index order.
Performance	The join between the parent and storage table might be a deciding factor. Optimize using IOTs and locators.	Varrays stored in-line give better performance than nested tables because no joins are made during data retrieval.	Performance dependent on the size of associative arrays and block operations.

The common exceptions that you might encounter during the collection design phase are as follows:

- `COLLECTION_IS_NULL`: This exception occurs when the collection is NULL
- `NO_DATA_FOUND`: This exception occurs when the element corresponding to a subscript does not exist
- `SUBSCRIPT_BEYOND_COUNT`: This exception occurs when the index exceeds the number of elements in the collection
- `SUBSCRIPT_OUTSIDE_LIMIT`: This exception occurs when the index is not a legal value
- `VALUE_ERROR`: This exception occurs, if you try to access an element without an index

Selecting the appropriate collection type

Here are a few guidelines that can help you decide the appropriate usage of collection type in programs:

- Associative arrays:
 - Use when you have to temporarily hold the program data in a hash map like a structure or an array
 - Note that there is no method to compare associative arrays
 - Can be used for key-value data models with string keys
- Nested tables:
 - Can be used for modeling one-to-many relationships within the parent table
 - You can compare collections using equality and non-equality, or multiset operators
 - Can be used For performing piecewise transactions on the child rows
- Varray:
 - Use when you know the maximum count of elements in the collection structure
 - Except for nullity, you cannot perform comparison operations
 - The order of the elements has to be preserved

Oracle 12c enhancements to collections

Oracle Database 12c allows a join between a table and a collection type. If an SQL collection type is used as a return type of a function, the table and the function output can be joined using `CROSS APPLY` and `OUTER APPLY`. The function must use a value from the joining table as a parameter and return a collection variable of a nested table or varray type.

For the purpose of illustration, let us create the test tables using dictionary views from the Oracle Database. The table `T_TBS_OBJ` contains the tablespace information and `T_SEGMENTS` contains the segments created on these tablespaces:

```
/*Create table T_TBS_OBJ*/
CREATE TABLE t_tbs_obj
AS
SELECT tablespace_name, status, allocation_type
FROM user_tablespaces
```

```

/
/*Create table T_SEGMENTS*/
CREATE TABLE t_segments
AS
SELECT segment_name, segment_type, tablespace_name, bytes, blocks
FROM user_segments
/

```

We'll create the nested table collection to be used in the string:

```

CREATE TYPE nt_string AS TABLE OF VARCHAR2(128);
/

```

The function `F_GET_SEGMENTS` returns the segments created on a given tablespace as a nested table collection type:

```

/*Create a function to return segments created on a tablespace*/
CREATE OR REPLACE FUNCTION f_get_segments (p_name VARCHAR2)
RETURN nt_string
AS
    l_seg nt_string;
BEGIN

/*Select query to fetch the segments from T_SEGMENTS*/
    SELECT CAST(COLLECT(segment_name) as nt_string)
    INTO l_seg
    FROM t_segments
    WHERE tablespace_name = p_name;

    RETURN l_seg;
END;
/

```

The following query uses the `CROSS APPLY` method to join the table and the function. Note that the output lists the tablespace name and the segments created on it. Uniquely, there are two tablespaces that is `USERS` and `ORCL`. What about the remaining tablespaces? Let's figure that out in the next code listing:

```

SELECT *
FROM t_tbs_obj CROSS APPLY f_get_segments(tablespace_name)
/

```

TABLESPACE_NAME	STATUS	ALLOCATION	COLUMN_VALUE
USERS	ONLINE	SYSTEM	DEPT
USERS	ONLINE	SYSTEM	EMP
USERS	ONLINE	SYSTEM	SALGRADE
ORCL	ONLINE	SYSTEM	T_BAT_SCORES
ORCL	ONLINE	SYSTEM	T_ANNUAL_PROD
ORCL	ONLINE	SYSTEM	T_OBJ

Using Collections

ORCL	ONLINE	SYSTEM	T_TMP_DB
ORCL	ONLINE	SYSTEM	T_ROW_ARCH
ORCL	ONLINE	SYSTEM	T_DEF_COLS
ORCL	ONLINE	SYSTEM	T_FUN_PLSQL
ORCL	ONLINE	SYSTEM	NT_SCORES_ST
USERS	ONLINE	SYSTEM	PK_DEPT
USERS	ONLINE	SYSTEM	PK_EMP
USERS	ONLINE	SYSTEM	SYS_C0010393
USERS	ONLINE	SYSTEM	SYS_C0010392
USERS	ONLINE	SYSTEM	SYS_FK0000093178N00003\$

16 rows selected.

Note that there are other tablespaces such as SYSTEM and SYSAUX with no user segments created on them. By virtue of cross join, the tablespaces with no segments were eliminated from the output. In such case, we can use the OUTER APPLY join method:

```
SELECT *
FROM t_tbs_obj OUTER APPLY f_get_segments(tablespace_name)
/
```

TABLESPACE_NAME	STATUS	ALLOCATION	COLUMN_VALUE
SYSTEM	ONLINE	SYSTEM	
SYSAUX	ONLINE	SYSTEM	
UNDOTBS1	ONLINE	SYSTEM	
TEMP	ONLINE	UNIFORM	
USERS	ONLINE	SYSTEM	DEPT
USERS	ONLINE	SYSTEM	EMP
USERS	ONLINE	SYSTEM	SALGRADE
ORCL	ONLINE	SYSTEM	T_BAT_SCORES
ORCL	ONLINE	SYSTEM	T_ANNUAL_PROD
ORCL	ONLINE	SYSTEM	T_OBJ
ORCL	ONLINE	SYSTEM	T_TMP_DB
ORCL	ONLINE	SYSTEM	T_ROW_ARCH
ORCL	ONLINE	SYSTEM	T_DEF_COLS
ORCL	ONLINE	SYSTEM	T_FUN_PLSQL
ORCL	ONLINE	SYSTEM	NT_SCORES_ST
USERS	ONLINE	SYSTEM	PK_DEPT
USERS	ONLINE	SYSTEM	PK_EMP
USERS	ONLINE	SYSTEM	SYS_C0010393
USERS	ONLINE	SYSTEM	SYS_C0010392
USERS	ONLINE	SYSTEM	SYS_FK0000093178N00003\$
DWH_DATA	ONLINE	SYSTEM	
FIN_DATA	ONLINE	SYSTEM	

22 rows selected.

PL/SQL collection methods

Oracle provides a set of methods that can work with collections for development operations. The development operations can be deleting an element, trimming the collection type, or fetching first and last subscripts. The syntax for all the collection methods is as follows:

```
[COLLECTION].METHOD (PARAMETERS)
```

EXISTS

The `EXISTS` function checks the existence of an element in a collection. The general syntax of this function is `[COLLECTION].EXISTS(<index>)`. It accepts the subscript as the input argument and searches for it in the associated collection. If the element corresponding to the index exists, it returns `TRUE`, or else returns `FALSE`. It is the only method that doesn't raise any exception when used with a collection.

The following PL/SQL block declares a local nested table collection and its two variables. While one array is uninitialized, the other one is initialized with sample data. It checks the existence of the first element in both arrays:

```
/*Enable the SERVEROUTPUT on to display the output*/
SET SERVEROUTPUT ON

/*Start the PL/SQL block*/
DECLARE

/*Declare a local nested table collection*/
TYPE coll_method_demo_t IS TABLE OF NUMBER;

/*Declare collection type variables*/
L_ARRAY1 coll_method_demo_t;
L_ARRAY2 coll_method_demo_t := coll_method_demo_t (45,87,57);

BEGIN
/*Check if the first cell exists in the array 1*/
IF L_ARRAY1.EXISTS(1) THEN
    DBMS_OUTPUT.PUT_LINE('Element 1 found in Array 1');
ELSE
    DBMS_OUTPUT.PUT_LINE('Element 1 NOT found in Array 1');
END IF;

/*Check if the first cell exists in the array 2*/
IF L_ARRAY2.EXISTS(1) THEN
    DBMS_OUTPUT.PUT_LINE('Element 1 found in Array 2');
```

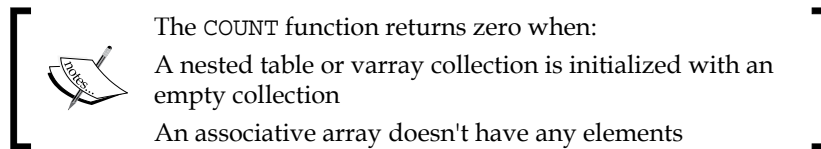
```
ELSE
    DBMS_OUTPUT.PUT_LINE('Element 1 NOT found in Array 2');
END IF;
END;
/

Element 1 NOT found in Array 1
Element 1 found in Array 2

PL/SQL procedure successfully completed.
```

COUNT

The COUNT function counts the number of elements in an initialized collection. For uninitialized collections, the method raises the COLLECTION_IS_NULL exception.



It can be used with all three types of collections.

The following PL/SQL block declares a local nested table collection and a variable of nested table type. Let us check the element count in both the collection variables:

```
/*Enable the SERVEROUTPUT on to display the output*/
SET SERVEROUTPUT ON

/*Start the PL/SQL block*/
DECLARE

/*Declare the local collection type*/
    TYPE nt_local IS TABLE OF NUMBER;

/*Declare and initialize the collection variables*/
    l_loc_var nt_local := nt_local (10,20,30);
BEGIN
    DBMS_OUTPUT.PUT_LINE('The array size is '||l_loc_var.count);
END;
/

The array size is 3

PL/SQL procedure successfully completed.
```

LIMIT

The `LIMIT` function returns the maximum number of elements that can be stored in a `VARRAY` collection type. The method raises the `COLLECTION_IS_NULL` exception for uninitialized collections.



For associative arrays or nested tables, the `LIMIT` method returns `NULL`.

The following PL/SQL block declares a local varray type and a variable of its type. The varray type variable has been initialized with test data. Observe the difference between the `COUNT` and `LIMIT` methods:

```

/*Enable the SERVEROUTPUT on to display the output*/
SET SERVEROUTPUT ON

/*Start the PL/SQL block*/
DECLARE

/*Declare local varray and its variable*/
TYPE va_local IS VARRAY(10) OF NUMBER;
l_array va_local := va_local (10,20,30);

BEGIN
/*Display the current count*/
  DBMS_OUTPUT.PUT_LINE('The varray size is '||L_ARRAY.COUNT);

/*Display the maximum limit*/
  DBMS_OUTPUT.PUT_LINE('The varray max size is '||L_ARRAY.LIMIT);
END;
/

The varray size is 3
The varray max size is 10

PL/SQL procedure successfully completed.
```

FIRST and LAST

The `FIRST` and `LAST` functions return the first and last elements of a collection. For an empty collection, both the methods return a `NULL` value. These methods can be used with all three types of collections. The `FIRST` and `LAST` methods raise the exception `COLLECTION_IS_NULL` to uninitialized collections.

The following PL/SQL block demonstrates the use of the `FIRST` and `LAST` methods with an initialized collection:

```
/*Enable the SERVEROUTPUT on to display the output*/
SET SERVEROUTPUT ON

/*Start the PL/SQL block*/
DECLARE

/*Display a local nested table collection*/
  TYPE coll_method_demo_t IS TABLE OF NUMBER;
  L_ARRAY coll_method_demo_t := coll_method_demo_t (10,20,30);

BEGIN

/*Display the first and last elements*/
  DBMS_OUTPUT.PUT_LINE('First element: ' || L_ARRAY (L_ARRAY.FIRST));
  DBMS_OUTPUT.PUT_LINE('Last element: ' || L_ARRAY (L_ARRAY.LAST));
END;
/

First element: 10
Last element: 30

PL/SQL procedure successfully completed.
```

PRIOR and NEXT

The `PRIOR` and `NEXT` functions take an input index and return its previous and next index. If the `PRIOR` and `NEXT` functions are used with first and last indexes respectively, the result is `NULL`.

Both the methods can be used with all three types of collections. The `PRIOR` and `NEXT` methods raise `COLLECTION_IS_NULL` exception when applied to uninitialized collections. The following PL/SQL shows the usage of the `PRIOR` and `NEXT` methods with a PL/SQL type collection:

```
/*Enable the SERVEROUTPUT on to display the output*/
SET SERVEROUTPUT ON

/*Start the PL/SQL block*/
DECLARE

/*Declare a local nested table collection*/
  TYPE nt_local IS TABLE OF NUMBER;
  L_ARRAY nt_local := nt_local (10,20,30,100,48,29,28);
```

```

    l_num_idx NUMBER;
BEGIN
/*Capture the element before 4th index*/
    l_num_idx := l_array.prior (4);
    DBMS_OUTPUT.PUT_LINE('Index before 4th element: '||l_num_idx);

/*Delete the fifth element*/
    l_array.delete (5);

/*Capture the element after 4th index*/
    l_num_idx := l_array.next (4);
    DBMS_OUTPUT.PUT_LINE('Index after 4th element: '||l_num_idx);
END;
/

```

Index before 4th element: 3

Index after 4th element: 6

PL/SQL procedure successfully completed.

In the preceding PL/SQL block, the local nested table collection was rendered sparse after the fifth element was deleted.

EXTEND

The `EXTEND` function is used to append elements to a persistent collection. It cannot be used with associative arrays. Being an overloaded function, `EXTEND` can be used to append a finite number of elements with a default value:

- `EXTEND`: It appends the collection with a `NULL` element.
- `EXTEND(x)`: It appends the collection with an `x` number of `NULL` elements.
- `EXTEND(x, y)`: It appends the collection with `x` elements with values as that of the `y` element. If the `y` element doesn't exist, the system raises a `SUBSCRIPT_BEYOND_COUNT` exception.

The following PL/SQL block demonstrates the extension of a collection type using all three signatures of the `EXTEND` method. The first extension appends the fourth `NULL` element to the array. The second extension appends the fifth and sixth `NULL` elements to the array. The third extension appends the seventh and eighth elements as 10 (the value of the first element) to the array:

```

/*Enable the SERVEROUTPUT on to display the output*/
SET SERVEROUTPUT ON

/*Start the PL/SQL block*/

```

```
DECLARE
/*Declare local nested table collection type*/
    TYPE nt_local IS TABLE OF NUMBER;

/*Declare collection type variable and initialize it*/
    l_ARRAY nt_local := nt_local (10,20,30);
    l_num_idx NUMBER;
BEGIN

/*Extend the collection using default signature */
    l_array.extend;
    l_num_idx := l_array (l_array.last);
    DBMS_OUTPUT.PUT_LINE(l_ARRAY.LAST||' element is = '||l_num_idx);

/*Extend the collection by adding two NULL elements*/
    l_array.extend (2);
    l_num_idx := l_array (l_array.last);
    DBMS_OUTPUT.PUT_LINE(l_ARRAY.LAST||' element is = '||l_num_idx);

/*Extend the collection by adding two elements with a value*/
    l_array.extend (2,1);
    l_num_idx := l_array (l_array.last);
    DBMS_OUTPUT.PUT_LINE(l_ARRAY.LAST||' element is = '||l_num_idx);
END;
/

4 element is =
6 element is =
8 element is = 10
```

PL/SQL procedure successfully completed.

The EXTEND method raises the COLLECTION_IS_NULL exception for uninitialized collections. If a varray is attempted for extension beyond its maximum allowed limit, Oracle raises an SUBSCRIPT_BEYOND_LIMIT exception.

TRIM

The TRIM function is used to cut the elements from a persistent collection. It cannot be used with associative array type collections. TRIM is an overloaded method, which can be used in the following two signatures:

- TRIM: It trims one element from the end of the collection.
- TRIM(n) : It trims n elements from the end of the collection. If n exceeds the total count of elements in the collection, the system raises a SUBSCRIPT_BEYOND_COUNT exception. No action is defined for NULL value of n.

The following PL/SQL block shows the operation of the TRIM method on an initialized PL/SQL table collection type:

```

/*Enable the SERVEROUTPUT on to display the output*/
SET SERVEROUTPUT ON

/*Start the PL/SQL block*/
DECLARE

/*Declare a local nested table collection type*/
  TYPE nt_local IS TABLE OF NUMBER;

/*Declare a collection variable and initialize it*/
  l_array nt_local := nt_local (10,20,30,40,50);
  l_num_idx NUMBER;
BEGIN

/*Trim the last element of the collection*/
  l_array.trim;
  l_num_idx := l_array (l_array.last);
  DBMS_OUTPUT.PUT_LINE(L_ARRAY.LAST||'element is  = '||l_num_idx);

/*Trim the last 2 elements of the collection*/
  DBMS_OUTPUT.PUT_LINE ('Trimming the array...');
  l_array.trim (2);
  l_num_idx := l_array (l_array.last);
  DBMS_OUTPUT.PUT_LINE(L_ARRAY.LAST||'element is  = '||l_num_idx);
END;
/

4 element is  = 40
Trimming the array...
2 element is  = 20

PL/SQL procedure successfully completed.

```

The TRIM method raises a `COLLECTION_IS_NULL` exception for uninitialized collections.

DELETE

The DELETE function is used to delete elements from a given collection. The DELETE operation leaves the collection sparse. Any reference to the deleted index would raise a `NO_DATA_FOUND` exception. The DELETE method raises a `COLLECTION_IS_NULL` exception for uninitialized collections. It can be used with all three types of collections.

The overloaded method can be used in the following signatures:

- DELETE: It flushes out all the elements of a collection
- DELETE(n): It deletes the *n*th index from the collection
- DELETE(n,m): It performs range deletion, where all the elements within the range of the subscripts *n* and *m* are deleted

The following PL/SQL block declares a collection along with its collection variable. This program displays the first element of the collection before and after the deletion of the first subscript:

```
/*Enable the SERVEROUTPUT on to display the output*/
SET SERVEROUTPUT ON

/*Start the PL/SQL block*/
DECLARE

/*Declare the local nested table collection*/
    TYPE nt_local IS TABLE OF NUMBER;

/*Declare and initialize a collection variable*/
    l_array nt_local := nt_local (10,20,30,40,50);
    l_num_idx NUMBER;
BEGIN

/*Capture the element at the first place*/
    l_num_idx := l_array (l_array.first);
    DBMS_OUTPUT.PUT_LINE('1st element is '||l_num_idx);

/*Delete the 1st element from the collection*/
    l_array.DELETE(1);

/*Capture the element at the first place*/
    l_num_idx := l_array (l_array.first);
    DBMS_OUTPUT.PUT_LINE('1st element is '||l_num_idx);

END;
/

1st element is 10
1st element is 20

PL/SQL procedure successfully completed.
```

Oracle doesn't allow the deletion of individual elements in a varray collection. Either all the elements of a varray are removed using the [VARRAY] .DELETE method or the elements can be trimmed from the end of the varray collection:

```

/*Enable the SERVEROUTPUT on to display the output*/
SET SERVEROUTPUT ON

/*Start the PL/SQL block*/
DECLARE

/*Declare the local varray collection*/
TYPE va_array IS VARRAY (10) OF NUMBER;

/*Declare a collection variable and initialize it*/
L_ARRAY va_array := va_array (10,20,30,40,50);

BEGIN
/*Delete the second element of varray*/
  L_ARRAY.DELETE(2);
END;
/
  L_ARRAY.DELETE(2);
  *
ERROR at line 8:
ORA-06550: line 8, column 3:
PLS-00306: wrong number or types of arguments in call to 'DELETE'
ORA-06550: line 8, column 3:
PL/SQL: Statement ignored.
```



It is recommended that the TRIM and DELETE methods must not be operated together or simultaneously on a collection. The DELETE method retains a placeholder for the deleted element while the TRIM method destroys the element from the collection. Therefore, the operation sequence DELETE(last) followed by TRIM(1) would result in the removal of a single element only.

Summary

This chapter covers one of the more interesting features of PL/SQL programming – collections. We discussed and learned the three types of collections in PL/SQL that is associative arrays, nested tables and varrays. Readers should understand the application of these collection types in different situations. The comparative study of all three would help readers to differentiate between each one and choose the most appropriate one in their development.

The next chapter will explore another distinctive capacity of PL/SQL as a language, that is in the use of external procedures.

Practice exercise

- Which two statements are true about associative arrays?
 1. Associative arrays can have negative subscripts.
 2. Associative arrays are always dense collections.
 3. Associative arrays don't need initialization in a PL/SQL block.
 4. The upper limit of associative arrays can be dynamically modified.
- Which of the following statements is true about nested tables?
 1. Nested tables are stored in a segment different from that of a parent table.
 2. Nested table columns can have string subscripts.
 3. Nested tables can grow dynamically up to any extent.
 4. A database column of nested table collection type can be separately queried by its storage name.
- Only `varrays` can have sequential numbers as subscripts.
 1. True
 2. False
- Which of the following associative array declarations is/are correct:

```
DECLARE
TYPE T1 IS TABLE OF NUMBER INDEX BY BOOLEAN;
TYPE T2 IS TABLE OF VARCHAR2(10) INDEX BY NUMBER;
TYPE T3 IS TABLE OF DATE INDEX BY SIGNTYPE;
TYPE T4 IS TABLE OF EMPLOYEES%ROWTYPE INDEX BY POSITIVE;
BEGIN
...
...
END;
```

 1. T1
 2. T2
 3. T3
 4. T4

- Which of the following statements is/are true about varrays?
 1. The limit of varray elements can be modified during runtime using the ALTER TABLE statement.
 2. A varray element can be deleted using the DELETE method.
 3. For an empty collection of varray type, the value of LAST is equal to COUNT.
 4. Varrays can exist as sparse collections.

- What will be the output of the following PL/SQL block:

```
DECLARE
  TYPE T IS TABLE OF NUMBER;
  L_NUM T := T(1,2);
BEGIN
  DBMS_OUTPUT.PUT_LINE(L_NUM(1));
  L_NUM := T(10,20);
  DBMS_OUTPUT.PUT_LINE(L_NUM(1));
END;
```

1. 1 and 1.
 2. 1 and 10.
 3. Oracle raises a COLLECTION_IS_NULL exception at line 5.
 4. 10 and 10.
- The EMPLOYEES table stores the details of 14 employees. Identify the solution of the error in the following PL/SQL program:

```
DECLARE
  TYPE EMP_VARRAY_T IS VARRAY (10) OF EMPLOYEES%ROWTYPE;
  L_EMP EMP_VARRAY_T := EMP_VARRAY_T();
BEGIN
  DBMS_OUTPUT.PUT_LINE(L_EMP.COUNT);
  SELECT *
  BULK COLLECT INTO L_EMP
  FROM EMPLOYEES;
  DBMS_OUTPUT.PUT_LINE(L_EMP.COUNT);
END;
/
0
DECLARE
*
ERROR at line 1:
ORA-22165: given index [11] must be in the range of [1] to [10]
ORA-06512: at line 6
```

1. The `varray` size must be increased to 14 or a higher limit.
 2. The `varray` variable must be initialized.
 3. Data for only 10 employees must be selected into a `varray` variable.
 4. The `varray` definition is wrong; a record type cannot be used as an element type of a `varray` collection.
- Which of the following statements are wrong about the collection methods?
 1. `EXISTS` raises a `NO_DATA_FOUND` exception if the element for the input subscript does not exist.
 2. `DELETE` can be used with `varrays`.
 3. `LIMIT` returns the current limit of nested table collections.
 4. `TRIM` removes an element of the collection from the end.

5

Using Advanced Interface Methods

Oracle enables the application developers to create PL/SQL routines that can invoke programs written in non-Oracle programming languages. An external program can be a **C**, **C++** based, or **Java**-based program. External procedures were introduced in Oracle 8i to add extensibility to the Oracle Database engine by allowing the non-Oracle programs to execute in the database kernel. This chapter covers the external procedures and their implementation in the following topics:

- Understanding external routines
- Architecture and benefits
- Executing external C programs from PL/SQL
- Executing external Java programs from PL/SQL

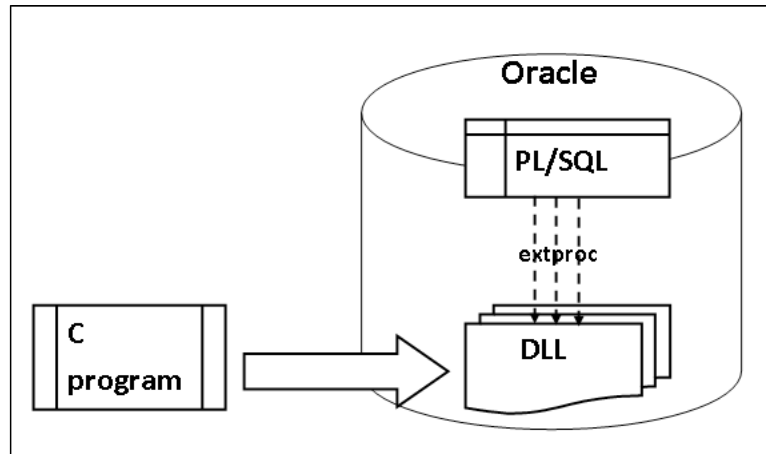
Overview of External Procedures

Oracle provides a rich platform for application programming through PL/SQL and the support for multilanguage programs. Programming languages that allow the technical translation of real world problems have different features and advantages, for example, PL/SQL is tightly integrated with SQL and computation-based logic can be run efficiently in C. Some of the crucial factors that determine the choice of a programming language can be code performance, portability, and security. Therefore, you might see application development involving multiple programming languages.

Oracle PL/SQL allows users to create a PL/SQL program that calls a C or Java procedure. The special purpose interface allows the users to utilize the strengths and features of multiple programming languages and support code reusability.

External Procedures

An external procedure is stored in a **dynamic linked library (DLL)** or a similar unit, which is prototyped in a call specification using Oracle PL/SQL. Whenever a PL/SQL procedure is called, the language compiler loads the target library at runtime and executes the external program as a PL/SQL procedure, entirely and natively within the Oracle Database. An interesting capability of an external procedure is that the call specification is independent of the code changes to the program.



The preceding diagram shows a high-level overview of the components involved in an external procedure call. We will discuss them in detail in this chapter.

External procedures enhance the development platform of the Oracle Databases. A C-based or Java-based program, when executed from the database server, avoids the network roundtrips made for client-to-database communication.

Components of external procedure execution flow

In this section, we will learn about the database components involved in the execution of an external procedure.

The extproc agent

The `extproc` agent plays the most vital role in interfacing external procedures in the Oracle Database. It is a process started by the Oracle Database or database listener, which facilitates the execution of an external procedure whenever required. It carries relevant information such as the library path, procedure name, and arguments. It returns the result of the execution back to the database processing engine.

In a dedicated server mode, each session will have a new instance of the `extproc` process. In a multiuser application, multiple sessions with multiple `extproc` processes may result in a disproportionate allocation of memory resources. In a shared server mode, a multithreaded `extproc` agent can provide the efficient utilization of server resources by maintaining a shared pool of the `extproc` processes.

The Library object

An external procedure can be executed as a shared library in Oracle. A library is an operating system file, which can be dynamically loaded upon request. On a Windows machine, you might be aware of the library files with a `.dll` extension, while on Unix and its flavors, the library is stored with a `.so` extension. A library can be shared by multiple external procedures. The external procedure can be written in **C**, **C++**, **Fortran**, **COBOL**, or **Visual Basic**. A shared library can be shared across multiple database sessions, thus optimizing the memory consumption.

For the representation of the shared library across a schema, a schema object is created as an alias library. This library can be created using the following syntax:

```
CREATE LIBRARY [schema].library_name
  {IS | AS} 'library_path'
  [AGENT 'agent_link'];
```

A library can be created by `SYS` or a user with the `CREATE LIBRARY` privilege. A user with the `EXECUTE` privilege on the library can refer it in the call specifications. The library path must be accessible by the `extproc` process, that is, either it must be explicitly specified in `EXTPROC_DLLS` or the library must reside in the `$ORACLE_HOME/bin/` or `$ORACLE_HOME/lib/` folders.



`EXTPROC_DLLS` is an environment parameter set in `extproc.ora`, (discussed later in this chapter).

Callout and Callback

There are two relevant terms — **callout** and **callback** — associated with the handling of an external program.

When the PL/SQL procedure invokes an external procedure, the call is referred to as a callout. If the external procedure invokes an SQL statement, which drives the database engine, the call is referred to as a callback.

Call Specification

A call specification is required to publish an external procedure. It enables the registration of an external procedure, database library, parameter mapping, memory management, and external procedure execution. It consists of a set of statements embedded in a PL/SQL subprogram. It can be specified in standalone PL/SQL subprograms, PL/SQL package specifications or body, and abstract data types.

The syntax for a call specification is as follows:

```
AS LANGUAGE [C | JAVA method signature]
[LIBRARY (library name)]
[NAME (external program name)]
[WITH CONTEXT]
[AGENT IN (formal parameters)]
[PARAMETERS (parameter list)]
```

The `AS LANGUAGE` clause specifies the publication of C and Java programs. The `AS EXTERNAL` clause is used to support the external programs with legacy applications. For Java programs, you need to specify the complete signature.

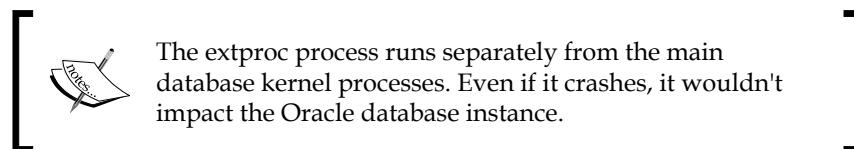
The `LIBRARY` is a valid schema object. It is a mandatory clause. You must have the `EXECUTE` privilege on the library.

The `NAME` is the external procedure name that has to be executed in the library. It is a mandatory input only if the calling PL/SQL subprogram is named differently from the external procedure.

The `PARAMETERS` clause specifies the parameter along with the data type, the respective position, and the pass by method (pass by value or reference). There might be some challenges in providing the parameter list because of the disparate data types available across the programming languages, support for nullability, and precision.

How an External Procedure executes

Whenever the PL/SQL runtime engine encounters an external procedure call, the Oracle Database starts the `extproc` process. The database passes on the information received from the call specification to the `extproc` process, which helps it to locate the external procedure within the library and execute it using the supplied parameters. The `extproc` process loads the dynamic linked library, executes the external procedure, and returns the result back to the database.



If any of the preceding steps fail to execute, an exception is raised. The following is a list of commonly encountered exceptions:

- **ORA-28576: Lost RPC connection to external procedure agent:** This exception is raised when the Oracle listener is not able to establish the connection with the `extproc` process.
- **ORA-28595: Extproc agent: Invalid DLL path:** This exception is raised when the `extproc` process is not able to locate the DLL in the specified location. You need to ensure that the library path is accessible by the process.
- **ORA-06521: PL/SQL Error mapping function:** This exception is raised when the external program is wrongly prototyped in the call specification. The reasons could be a parameter data type mismatch or the wrong program name.

Environment setup

The `extproc` process attempts to locate the library files in an accessible location. The location of the target libraries can either be specified with the listener or in an `extproc.ora` file. By default, the `extproc` process will be started by the Oracle Database. If you want to continue with the default behavior, it is recommended to specify the library paths in `extproc.ora`.

If you are working with distributed databases or a multithreaded `extproc` agent, you can override the default behavior by modifying `listener.ora` file.

TNSNAMES.ora

The `TNSNAMES.ora` file that gets configured at the time of the database software installation (or can be configured later), includes an entry for the `ORACLR_CONNECTION_DATA` service to support external services. This service verifies the network connection by using the `ADDRESS` parameter value and connects using the `CONNECT_DATA` parameter value. The following is the entry in `tnsnames.ora` file:

```
ORACLR_CONNECTION_DATA =
  (DESCRIPTION =
    (ADDRESS_LIST =
      (ADDRESS = (PROTOCOL = IPC) (KEY = EXTPROC1521))
    )
    (CONNECT_DATA =
      (SID = CLRExtproc)
      (PRESENTATION = RO)
    )
  )
```

The `ADDRESS` parameter value checks for the listener, which can receive **IPC (Internet Procedure Calls)** requests through the `KEY` value, `EXTPROC1521`. The `PROTOCOL` parameter has a fixed `IPC` value to establish the interaction between the server and external service requests. Once the `ADDRESS` setup matches the current active listener, it uses the `CONNECT_DATA` parameter value to spawn the `extproc` process. The `PRESENTATION` parameter is a performance optimizing parameter that directs the database server to concentrate and respond to the client through a protocol – **Remote-Ops (RO)**.



`CLRExtproc` is a listener mode that allows a PL/SQL programs to access external programs.



EXTPROC.ora

In the default configuration, you will need to set the `EXTPROC_DLLS` environment variable in `extproc.ora` for the `extproc` process. The syntax to be used is as follows:

```
SET EXTPROC_DLLS= <library path>
```

The possible values for the environment variable are `ONLY`, `ANY`, or a colon (:), separated by the actual DLL path.

- `NULL` (default): The `extproc` process can only load the libraries located in the `$ORACLE_HOME/bin` and `$ORACLE_HOME/lib` paths.

- **ONLY: [DLL:DLL...]:** This provides maximum protection by limiting the number of libraries on the system. The `extproc` process loads either of these libraries.
- **[DLL:DLL ...]:** This allows you to specify the DLL paths separated by a colon. The `extproc` process can load the libraries located in these paths or from the `$ORACLE_HOME/bin` and `$ORACLE_HOME/lib` folders.
- **ANY:** This allows any DLL on the server to be loaded by the `extproc` process.



Use a semicolon as a delimiter while specifying libraries on a Windows server.

In a distributed database system, you need to use the multithreaded `extproc` process to be spawned by the Oracle listener and not the database. The advantage of having the multithreaded `extproc` agent in an Oracle Database shared server architecture is optimized resource consumption. It uses a shared pool of `extproc` agent threads which are sequentially assigned in a database sessions in a queue. The multithreaded `extproc` agent ensures the efficient utilization of the system resources.



When running an Oracle database server in shared mode, the agent process and listener must reside on the same physical server.

Executing external C programs from PL/SQL

Let's us walk through an illustration on how to execute an external procedure, written in the C language, in the Oracle Database.

- **Step 1: Creating and compiling the C program.**

The following C program (`GetMax.c`) finds the maximum of the two number values:

```
#include <stdio.h>

/* Define the function returning the max between two numbers */
int GetMax(int num1, int num2)
{
    /* local variable declaration */
    int result;
```

```
    if (num1 > num2)
        result = num1;
    else
        result = num2;

    return result;
}
```

Compile the program by using a C compiler.

```
sh-4.3# gcc -c GetMax.c
```

- **Step 2: Generating the DLL and creating the library object in the Oracle Database.**

We will now generate the DLL for the C program:

```
sh-4.3# gcc -shared GetMax.c -o GetMax.dll
```

- The library location

The default paths searched by the `extproc` process for loading the required library are `$ORACLE_HOME/bin` and `$ORACLE_HOME/lib`. For testing purposes, we will continue with the defaults. For this reason, the setting in `extproc.ora` remains unchanged (to `NULL`), as follows:

```
SET EXTPROC_DLLS =
```

We will place the shared library generated by the compiler in the `$ORACLE_HOME/lib` folder. If you want to use a non-default path, set the `EXTPROC_DLLS` environment variable in the `extproc.ora` (`$ORACLE_HOME/hs/`) file.

The following script creates the library object in the `SCOTT` schema. Note that the `SYS` has granted the `SCOTT` user the `CREATE LIBRARY` privilege:

```
/*Connect as SYSDBA*/
CONN sys/oracle AS SYSDBA

/*Grant the CREATE LIBRARY privilege*/
GRANT CREATE LIBRARY TO SCOTT
/
```

Grant succeeded.

Now, the SCOTT user can create a library in the database for the shared library path.

```
/*Connect as SCOTT user*/
CONN scott/tiger

/*Create the alias library object using the .so DLL path*/
CREATE OR REPLACE LIBRARY GetMaxDll
AS '/u01/app/oracle/product/12.1.0.2/db_1/lib/GetMax.so'
/
```

Library created.

The library metadata can be queried from the USER_LIBRARIES dictionary view. This view captures the data for the libraries owned by the current user:

```
SELECT library_name,
       file_spec,
       dynamic,
       status
FROM user_libraries
/
```

LIBRARY_NAME	FILE_SPEC	D	STATUS
GETMAXDLL	/u01/app/oracle/product/12.1.0.2/db_1/lib/GetMax.so	Y	VALID

- **Step 3: Creating the PL/SQL wrapper function with a call specification.**

For the GetMax external program, let's create a call specification. Within the call specification, we will provide the library name, C program name, and parameters:

```
/* Create the function with call specification*/
CREATE OR REPLACE FUNCTION f_get_max
(p_num1 PLS_INTEGER, p_num2 PLS_INTEGER)
RETURN PLS_INTEGER

/* Call Specification starts */
AS LANGUAGE C

/*Specify the PL/SQL library object name*/
LIBRARY GetMaxDll
```

```
/*Specify the external function name*/  
NAME "GetMax"
```

```
/*Specify the parameters*/  
PARAMETERS (p_num1, p_num2);  
/
```

Function created.

- **Step 4: Testing the PL/SQL procedure.**

The following anonymous PL/SQL block invokes the F_GET_MAX function:

```
/*Enable the SERVEROUTPUT to display the block output*/  
SET SERVEROUTPUT ON  
DECLARE
```

```
/*Declare local variables*/  
  a PLS_INTEGER := 10;  
  b PLS_INTEGER := 30;  
  c PLS_INTEGER;  
BEGIN
```

```
/*Call the F_GET_MAX function */  
  c := F_GET_MAX (a, b);
```

```
/*Display the output*/  
  DBMS_OUTPUT.PUT_LINE ('The maximum number is - '||c);  
END;  
/
```

The maximum number is - 30

PL/SQL procedure successfully completed.

That's the correct result. We have just executed a C program from Oracle PL/SQL. As soon as the PL/SQL block execution starts, you can trace the extproc process by listing the processes. The extproc process is active until the end of the session.

```
[oracle@devhost ~]$ ps -ef | grep extproc  
oracle   5264      1  0 15:01 ?        00:00:00 /u01/app/oracle/  
product/12.1.0.2/db_1/bin/extproc (DESCRIPTION=(LOCAL=YES)  
(ADDRESS=(PROTOCOL=BEQ)))  
oracle   5266   5115  0 15:01 dev/1    00:00:00 grep extproc
```

The restrictions on C external procedures are listed as follows:

- The feature is limited to the platforms that support dynamically linked libraries
- The `extproc` process and PL/SQL procedure need to be on the same host
- The `library` clause cannot point to a remote location through a database link
- The maximum number of parameters to an external procedure is 128
- Parameters of cursor variable types are not supported with external procedures

Securing External Procedures with Oracle Database 12c

The Oracle Database creates the `extproc` process and runs under the operating system user, that starts the listener or runs an Oracle server process. Quite often, you will see the `extproc` process running as the `oracle` user. The `extproc` process is not physically associated with the Oracle Database.

Oracle Database 12c enables enhanced security for `extproc` by authenticating it against a user-supplied credential. This new feature allows the creation of a user credential and associates it with the PL/SQL library object. Whenever the application calls an external procedure, the `extproc` process authenticates the connection before loading the shared library.

The `DBMS_CREDENTIAL` package allows the configuration of the credential through member subprograms. The `CREATE LIBRARY` statement has been enhanced for credential specification. A new environment variable, `ENFORCE_CREDENTIAL`, can be specified in `extproc.ora` to control the authentication by the `extproc` process. The default value of the parameter is `FALSE`. Another new environment variable, `GLOBAL_EXTPROC_CREDENTIAL`, serves as the default credential and is only used when the credential is not specified for a library. If `ENFORCE_CREDENTIAL` is `FALSE` and no credential has been defined in the PL/SQL library, there will be no user authentication; this means the `extproc` process will authenticate by using the privileges of the user running the Oracle server.

The following PL/SQL block creates a credential by using `DBMS_CREDENTIAL.CREATE_CREDENTIAL`. This credential is built using the `ORADEV` user:

```
BEGIN
  DBMS_CREDENTIAL.CREATE_CREDENTIAL (
    credential_name => 'devhost_auth',
    user_name       => 'oradev',
```



```
        password          => 'oradev')
END;
/
```

The library definition will include an additional CREDENTIAL clause:

```
CREATE OR REPLACE LIBRARY myextlib
AS 'HelloWorld.so'
CREDENTIAL devhost_auth
/
```

When the extproc process reads the call specification and finds the shared library with a secured credential, it authenticates the library on behalf of the credential and then loads it.

Executing Java programs from PL/SQL

Like the C programs, Java programs can also natively execute in the Oracle Database. However, unlike C external programs, the Java classes and Java source files are stored as schema objects in the database.

External Java programs are not executed through OS-based shared libraries but use a Java shared library or libunit for execution. **Libunits** are similar to dynamically linked libraries but they are mapped one-to-one to the Java class and are not sharable across the other methods.

The libunit is loaded and executed by the **Java Virtual Machine (JVM)** which resides natively in the Oracle Database. The JVM uses the Java pool component of the shared global area to execute the Java-based external program.

Loading a Java class into a database

A Java program can be loaded into an Oracle Database by using the CREATE JAVA statement or the LOADJAVA utility.

The CREATE JAVA statement enables the Java Virtual Machine library manager to load a Java class into the Oracle Database and generate the RDBMS shared library or libunit in the target directory. The CREATE JAVA statement syntax is as follows:

```
CREATE JAVA CLASS USING BFILE (directory, java class)
```

Java classes can also be loaded by using the `LOADJAVA` utility. The command line utility uploads the Java class as a large object in a system-generated table and generates the shared library.

The `loadjava` utility can be used as per the following syntax:

```
loadjava {-user | -u} username/password[@database] [option ...]  
filename [filename ]...
```



You can use `loadjava -h | -help` to get a helper index.

Steps to execute a Java class from an Oracle PL/SQL unit

In this section, we will illustrate the loading of a Java class into an Oracle Database and invoke it by using a PL/SQL procedure.

- **Step 1: Creating a Java class.**

The `Compute` Java class includes the following two methods to add and multiply two numbers:

```
public class Compute {  
  
    public static int Sum (int x, int y) {  
        return x+y;  
    }  
  
    public static int Cross (int x, int y) {  
        return x*y;  
    }  
  
}
```

- **Step 2: Loading the Java class into Oracle.**

We will now upload the Java class into the Oracle Database by using the `loadjava` utility:

```
[oracle@devhost ~]$ loadjava -user scott/tiger Compute.java
```

A successful upload operation gets terminated without any confirmation message. You can confirm the load operation by querying the Java class in the USER_JAVA_CLASSES, USER_OBJECTS and USER_SOURCE dictionary views:

```
/*Query the USER_JAVA_CLASSES view*/
SELECT name, kind, accessibility, source
FROM user_java_classes
/
```

NAME	KIND	ACCESSIBILITY	SOURCE
-----	-----	-----	-----
Compute	CLASS	PUBLIC	Compute

```
/*Query the USER_OBJECTS view*/
COL object_name format a30
SELECT object_name, object_type, status
FROM USER_OBJECTS
WHERE TRUNC(created)=TRUNC(SYSDATE)
ORDER BY timestamp
/
```

OBJECT_NAME	OBJECT_TYPE	STATUS
-----	-----	-----
SYS_LOB0000093463C00002\$\$	LOB	VALID
SYS_IL0000093463C00002\$\$	INDEX	VALID
SYS_C0010407	INDEX	VALID
CREATE\$JAVA\$LOB\$TABLE	TABLE	VALID
JAVA\$OPTIONS	TABLE	VALID
Compute	JAVA SOURCE	INVALID
Compute	JAVA CLASS	INVALID

7 rows selected.

You can ignore the invalid status of the JAVA SOURCE and JAVA CLASS objects. They both will get validated upon invocation.

The JAVA\$OPTIONS table stores the compiler options:

```
SELECT *
FROM JAVA$OPTIONS
/
```

WHAT	OPT	VALUE
-----	-----	-----
Compute	encoding	UTF-8

The system-generated `CREATE$JAVA$LOB$TABLE` table stores the Java class as a SecureFile large object.



Starting with Oracle Database 12c, SecureFile is the default storage mechanism for large objects.

Let's drill down by a step to further check out the table structure from the `USER_TAB_COLS` dictionary view:

```
SELECT table_name,
       column_name,
       data_type
FROM user_tab_cols
WHERE table_name='CREATE$JAVA$LOB$TABLE'
/
```

TABLE_NAME	COLUMN_NAME	DATA_TYPE
CREATE\$JAVA\$LOB\$TABLE	LOADTIME	DATE
CREATE\$JAVA\$LOB\$TABLE	LOB	BLOB
CREATE\$JAVA\$LOB\$TABLE	NAME	VARCHAR2

Verify the LOB storage as a SecureFile for the LOB column in the table:

```
SELECT table_name, column_name, securefile
FROM user_lobs
WHERE table_name = 'CREATE$JAVA$LOB$TABLE'
/
```

TABLE_NAME	COLUMN_NAME	SEC
CREATE\$JAVA\$LOB\$TABLE	LOB	YES

- **Step 3: Publishing the Java class method by using a call specification.**

Let's create a PL/SQL wrapper that will include a call specification to register the Java program and specify the parameter mapping. Note that, unlike C program publishing, you don't need to specify the library unit and external procedure name. The `AS LANGUAGE` provides the interface between the Java class and Oracle PL/SQL.

The syntax for the call specification is as follows:

```
{IS | AS} LANGUAGE JAVA
NAME 'method_fullname (parameters)
[return java_type_fullname]';
```

We shall now create the call specification for the `Compute` Java class. The standalone function accepts two numbers and returns their sum using the `Sum` method:

```
/*Function to publish Java class method*/
CREATE OR REPLACE FUNCTION F_COMPUTE_SUM
(P_X NUMBER, P_Y NUMBER)
RETURN NUMBER
AS

/*Specify the external programs base language*/
LANGUAGE JAVA
NAME 'Compute.Sum(int,int) return int';
/
```

Function created.

- **Step 4: Verifying the Java class execution method.**

Let's invoke the `F_COMPUTE_SUM` function to verify the execution of the Java class method:

```
variable x number;
variable y number;
exec :x := 100;
```

PL/SQL procedure successfully completed.

```
exec :y := 95;
```

PL/SQL procedure successfully completed.

```
variable result number;
exec :result := f_compute_sum (:x, :y);
```

PL/SQL procedure successfully completed.

```
print result
```

```
      RESULT
-----
      195
```

Summary

External procedures demonstrate the extensibility feature of an Oracle Database that allows a program developed in non-Oracle scripting language to be executed natively in the database kernel. The readers will find it interesting to explore how they can enable complex algorithms to run on the server side.

In the next chapter, we will be discussing a data security feature called Virtual Private Database. From a data access standpoint, this feature is quite important and easy to implement with bigger benefits.

Practice exercise

- Which of the following statements are true about the `extproc` process?
 1. It loads the shared library of the external C program.
 2. It is started by the PL/SQL runtime engine.
 3. It is a session-specific process.
 4. The `extproc` process compiles the C program while loading.
- Oracle 7 introduced the external procedure feature for sending e-mails from PL/SQL.
 1. True
 2. False
- Determine the effect of dropping the library object that has been used in a PL/SQL call specification while it is still in use:
 1. The PL/SQL wrapper method gets invalidated.
 2. The shared library gets corrupted.
 3. The PL/SQL wrapper method still works fine as it has already been executed once.
 4. The PL/SQL wrapper method gives no output.
- Examine the following `TNSNAMES.ora` and `LISTENER.ora` entries and choose the correct option:

```
//TNSNAMES.ora
ORACLR_CONNECTION_DATA =
  (DESCRIPTION =
    (ADDRESS_LIST =
      (ADDRESS = (PROTOCOL = IPC) (KEY = EXTPROC1521))
    )
  )
```

```
(CONNECT_DATA =
  (SID = extproc)
  (PRESENTATION = RO)
)
)
//LISTENER.ora
SID_LIST_LISTENER =
(SID_LIST =
  (SID_DESC =
    (SID_NAME = CLRExtproc)
    (ORACLE_HOME = C:\ORCL\11.2.0\dbhome_1)
    (PROGRAM = EXTPROC1521)
    (ENVS= "EXTPROC_DLLS=ONLY:C:\ORCL\product\11.2.0\dbhome_1\BIN\
Ext.dll")
  )
)
)
LISTENER =
  (DESCRIPTION_LIST =
    (DESCRIPTION =
      (ADDRESS = (PROTOCOL = IPC) (KEY = EXTPROC1521))
      (ADDRESS = (PROTOCOL = TCP) (HOST = localhost) (PORT = 1521))
    )
  )
)
```

1. The KEY value under ADDRESS of ORACLR_CONNECTION_DATA must be extproc.
 2. The SID value under CONNECT_DATA of ORACLR_CONNECTION_DATA must be CLRExtproc.
 3. The KEY value under ADDRESS of LISTENER must be extproc.
 4. The PROGRAM value of SID_LIST_LISTENER must be extproc.
- Which of the following statements about the loadjava utility are true?
 1. It generates a shared library for the Java programs.
 2. It loads the Java program in the Oracle Database.
 3. The loadjava utility requires a Java compiler to run.
 4. It loads the Java class file into the Java pool in the database instance.
 - External programs in Java don't require the shared libraries to be executed from PL/SQL.
 1. True
 2. False

- An external C program looks like this:

```
#include<stdio.h>
#include<conio.h>
int GetDouble(int num)
{
    return num * 2;
    getch();
}
```

The PL/SQL wrapper method looks like this:

```
CREATE OR REPLACE FUNCTION F_GET_DOUBLE (P_NUM NUMBER)
RETURN NUMBER
AS EXTERNAL LIBRARY NUMLIB
NAME "GETDOUBLE"
LANGUAGE C
PARAMETERS (P_NUM INT);
```

When you call the PL/SQL wrapper, you get the following exception:

```
DECLARE
*
ERROR at line 1:
ORA-06521: PL/SQL: Error mapping function
ORA-06522: Unable to load symbol from DLL
ORA-06512: at "ORADEV.F_GET_DOUBLE", line 1
ORA-06512: at line 4
```

Identify the cause for this exception:

1. The NUMLIB library has been incorrectly placed.
2. The C program has syntactical errors.
3. The C program name in the call specification (the PL/SQL wrapper) should be `GetDouble` instead of `GETDOUBLE`.
4. The `extproc` process is not working properly.

6

Virtual Private Database

Information security is one of the key challenges that organizations have to address to protect data privacy and its regulation. Leakage of sensitive information or unauthorized access to data might lead to hazardous consequences. With the stringent protocols of data distribution and steep rise of the internet usage, data security can pose a tedious question to the organizations. Information that is prone to invasions can be personal details, financial data, credit card information, business leads, or an intellectual property. A relational database can expose potential areas of injection in the form of network access, over privileged user accounts, non-database file access, non sanitized inputs, and proliferation of production data to unregulated environments. Knowing the fatal consequences of data breaches, the data security solutions must be efficient to bypass the attacks from the database layer, network, and application.

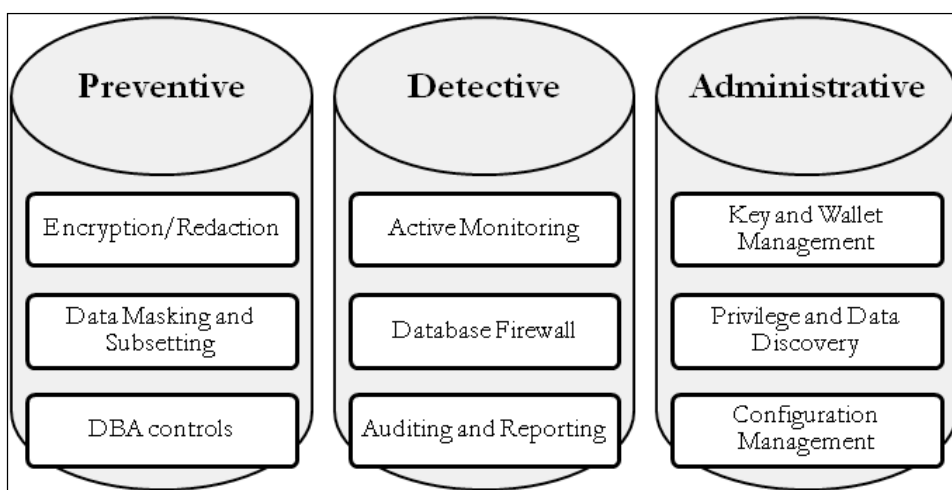
In this chapter, we will walk through an overview of Oracle's Database Security solution. We will focus on the implementation of a Virtual Private Database to develop secure applications. In addition, we will highlight the key security enhancements in Oracle Database 12c.

- Oracle Database Security overview
- Virtual Private Database
- Oracle Database 12c Security enhancements

Oracle Database Security overview

Oracle offers a comprehensive suite of database security solutions that aim to secure the databases and applications by protecting sensitive information, detecting threats and attacks, and managing security policies along the lines of organizational compliance. The **defense-in-depth** approach of Oracle can be categorized under preventive, detective, and administrative controls that ensure all the vulnerable areas of an application are covered. A powerful security strategy can be developed and laid down by using the right mix of security features. The factors impacting the formulation of a security strategy can be threat anticipation, data sensitivity, data distribution and user classification, compliance, and internal regulations.

Each of the above categories includes various products and technical features. The following diagram bifurcates the security pillars into security measures:



Oracle Database security feature under three categories

The Oracle Database Security solution includes the following database products:

- **Oracle Advanced Security:** Oracle Advanced Security is a licensed option available with the Oracle Database Enterprise Edition. This option includes various features that can be implemented at the database level to prevent data hacking and reduce the direct exposure of sensitive information to an unauthorized user. It helps protect the data from malicious attacks and comply with the security regulations like **Payment Card Industry Data Security Standard (PCI-DSS)** and Health Insurance Portability and Accountability Act (HIPAA). For example, you can encrypt the customer's bank account information in the database. Similarly, the sensitive information in the database can be redacted for support or call center executives.

The Oracle Advanced Security option includes:

- **Data Redaction:** The Data Redaction feature was introduced in Oracle Database 12c to apply on-the-fly masking policies to the sensitive information in the database. The feature has been backported to Oracle Database 11.2.0.4. Data redaction can be a handy feature as it is cost-efficient and dynamically obfuscates the data. This feature is covered in detail in the *Oracle Database 12c Security Enhancements* section of this chapter.
- **Transparent Data Encryption (TDE):** The encryption feature prevents the data, which is lying in storage, from outside attacks. The data is stored in an encrypted format, which gets auto-decrypted. It is installed at the time of the Oracle Database software installation. It is an easy-to-implement transparent solution and requires no changes to the existing SQL queries.



Securefile encryption is part of the Advanced Security option.

- The other features included under the Advanced Security option are Data Pump Export file encryption and **RMAN** Backup encryption to disk.
- **Virtual Private Database (VPD):** This is a security feature available in Oracle Database Enterprise Edition that controls the data visibility and accessibility depending on the database user authorization. First introduced in Oracle 8i, the feature works with VPD policies and session contexts. All queries issued by a user are appended by an additional predicate to restrict the rows in the result set. It is also known as Fine Grained Access Control or Row Level Security.
- **Real Application Security (RAS):** This feature was introduced in Oracle Database 12c to support the diversified security requirements in an enterprise application model. It presents an advanced version of the Virtual Private Database technology. Not only the data visibility, a Real Application Security policy also allows you to control the transactional permissions on the application objects. It is one of the smartest security solutions that can control the privileges of an application user on the business objects.
- **Oracle Key Vault (OKV):** Oracle Key Vault provides a robust key management platform that can be used as the centralized repository of TDE master keys, Oracle wallets, Java key stores, SSH keys, and other security credential files. The OKV store allows the end-to-end administration of the key lifecycle including creation, retention, and key rotation.

- **Oracle Database Vault:** Oracle Database Vault enables a security solution to safeguard the application data from privileged database users and ad hoc access. A multifactor database vault policy imposes deterrent controls on the user access to the database and reports violations, if any. It is a powerful proactive solution that helps in reducing the attack surface for highly privileged users like `SYSTEM`, `SYSDBA`, or a user with a large set of roles and privileges. Oracle provides default vault policies for the Oracle E-Business Suite (versions 11.5.10, 12.0, and 12.1), Oracle PeopleSoft application, Oracle Siebel CRM (version 7.7 and higher), and SAP applications.
- **Privilege Analysis:** The new feature in Oracle Database 12c analyzes the privileged users, roles, and database to report the used and unused privileges. Based on the analysis, the security administrator (or the DBA) can determine whether to retain or revoke the privileges of the developers or database users. This feature is quite important from the point of view of application security and helps reduce the risks from internal users.
- **Oracle Data Masking and Subsetting:** The Oracle Data Masking and Subsetting pack masks the original copy of the application data by physically obfuscating the sensitive information in either the complete application database or just its subset. The masked data can then be proliferated for analytical, testing, or regulatory purposes. The sensitive information and referential integrity model can be discovered based on the prebuilt data patterns. A portion of a large database can be extracted based on factors such as percentage, the number of rows, or column predicates. This pack contains a variety of masking formats and transformations that can be applied to the application data.
- **Oracle Label Security (OLS):** Oracle Label Security, first introduced in Oracle 8i, is a database option available in the enterprise edition of the Oracle Database. It enables the classification of data by labels. The security policies control the user access privileges and authorization on the application by the data labels. This option is very useful in implementing multilevel security in large legacy applications.
- **Oracle Audit Vault and Database Firewall (AVDF):** This product can be deployed in all Oracle Database editions and non-Oracle products such as Microsoft SQL Server, IBM DB2 for LUW, SAP Sybase ASE, MySQL databases and Oracle Big Data Appliance. You can define the rules to capture and block the unauthorized SQL access to the application data. Another feature of this product is the consolidation of the database audit with the system and network audit information, which then generates alerts and reports.



Strong authentication services such as Kerberos, PKI, and RADIUS along with network encryption are available in all the editions of the Oracle Database.

Fine-Grained Access Control

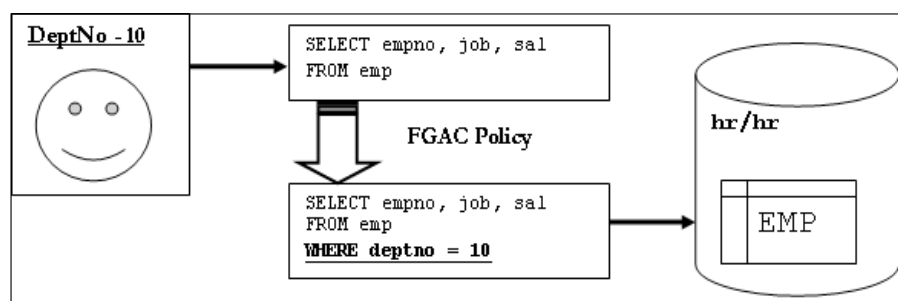
In an enterprise application, user isolation or role-based multitenancy ensures that data is accessed by the authorized users only. In a **Software as a Service (SaaS)** model application, tenant isolation is one of the prime demands. Application users are authorized to access only their world of data and not peek into other user's data.

Fine-grained access control (FGAC) enables the enforcement of security policies on the access of rows and columns based on the user's role and authorization. For example, an HR representative is authorized to view the details of employees that belong only to his vertical. Similarly, a relationship officer in a bank is authorized to access the account details of those customers that belong to his branch. The FGAC feature provides you with the mechanism to expose the authorized piece of data only to the approved user.

How FGAC works

Fine-grained access control enables the creation of security policies that can be associated with a table or view. Upon invocation, the security policy evaluates the policy function that returns a predicate clause, which is dynamically appended to the SQL queries issued by the user. Thus, the user views only the authorized rows from the table or a view. The security policy can limit either rows or columns or both.

In the following figure, a user from department 10 queries the EMP table. He issues a `SELECT` statement to view the employee details. Oracle invokes the FGAC policy attached to the EMP table and appends a predicate (that is, the `WHERE` clause) to the `SELECT` query. The modified `SELECT` query is then executed in the HR schema of the database.



How Fine-Grained Access Control works

The following are the salient features of fine-grained access control:

- Oracle provides the `DBMS_RLS` package for the management of security policies such as creation, drop, enable, and disable
- A policy can be set to be invoked only when given columns are accessed
- Multiple security policies can be associated with a table or view
- All security policies and the predicates are appended with the `AND` clause

Virtual Private Database

The working principle of the Virtual Private Database technology is that users should have isolated and distinguished data access. **Virtual Private Database (VPD)** is a feature that is built on fine-grained access control and uses application contexts to define and add the predicates to the SQL queries. Similar to FGAC, Virtual Private Database lays down the security framework through the security policies in the Oracle Database and associates them directly to tables, views, or synonyms. The security policies act as a safety net on the objects, and by no means they can be bypassed.

How does Virtual Private Database work?

Whenever a user issues an SQL query against a Virtual Private Database protected table, Oracle invokes the policy and evaluates the policy function. The policy function is a PL/SQL function that returns a predicate or a `WHERE` clause. The query is then rewritten using dynamic views to add the predicate clause. The modified query is executed against the database server and the result set is returned to the user. The predicate or the `WHERE` clause participates in the execution of the query, that is, the column indexes will impact the query execution plan.

Note that the Virtual Private Database protects only the `SELECT`, `INDEX`, and `DML` operations (`INSERT`, `UPDATE`, and `DELETE`). It doesn't work with `DDL`'s like `TRUNCATE` or `ALTER TABLE` statements.

Column-level Virtual Private Database

Row-level security can also be imposed on tables and views whenever a user selects sensitive columns. The sensitive columns can be specified at the time of the policy creation, that is, the `SEC_RELEVANT_COLS` parameter of the `DBMS_RLS.ADD_POLICY` procedure.

The column-level policy works only for the `SELECT` queries. When a user selects sensitive columns (implicitly or explicitly), the non-authorized rows are excluded from the result set. For example, if an HR representative queries public information such as the name and department of employees, the query will list all the records from the table. If he attempts to select the secured columns like `SAL`, the result set will be limited by the row-level security condition.

You can include all the rows with the sensitive column values marked as `NULL`. It can be achieved by specifying an attribute known as `SEC_RELEVANT_COLS_OPT` in the `DBMS_RLS.ADD_POLICY` procedure.

Virtual Private Database with Oracle Database 12c Multitenant

Virtual Private Database is supported within a pluggable database. You cannot create a security policy in the root container, that is, the container database.

Virtual Private Database components

The VPD technology makes use of the application package, policy function, application context, and the `DBMS_RLS` package to add policies. In the following section, we will discuss the building blocks of the Virtual Private Database implementation.

Application Context

A Virtual Private Database implementation with application contexts is useful when working with multiple security attributes. For example, in an organization, the data view authorization may be different for different job roles. In a manufacturing unit, A supervisor can be authorized to query the global parts while an executive can see only the local parts. In such cases, the job role can be a context attribute whose value can be used in the predicate clause returned by the policy function.

Application contexts are session variables that hold the user and session information securely. Context attributes are stored as key-value pairs under a given namespace, which is also known as a label. `USERENV` is a system-defined namespace in the Oracle Database that can provide session information such as `DB_NAME`, `SESSION_USER`, and others. User-defined application contexts can be created by a `SYSDBA` or users with the `CREATE ANY CONTEXT` privilege by using a trusted application package.

An application context can be either database session-based, client session-based, or global.

A database session-based application context is accessible only within a database user's session. Its attributes and values are stored in a session cache, **User Global Area (UGA)**. The database session-based application contexts can either be initialized locally from the session, externally using a database link, OCI interface or job queue process, or globally from a centralized directory using the Oracle Internet Directory. You can query the database session-based context attributes from the `SESSION_CONTEXT` dictionary view.

A client session-based application context is initialized using OCI functions (`OCIAppCtxSet`) and is stored in **User Global Area (UGA)** under the `CLIENTCONTEXT` namespace. The application can set the non-database attributes within the contexts and use them during policy creation. Ensure that the database session and client session-based application contexts do not conflict with each other.

A global application context can be accessed by all the database user sessions in a single instance or RAC environment. The context attributes and their values are stored in **System Global Area (SGA)**, which can be queried from the `GLOBAL_CONTEXT` dictionary view.

The following SQL queries the database name from the `USERENV` namespace using the `SYS_CONTEXT` function:

```
/*Query the database name from USERENV namespace*/
SELECT SYS_CONTEXT('USERENV','DB_NAME')
FROM dual
/

SYS_CONTEXT('USERENV','DB_NAME')
-----
orcl
```

You can create database session-based application contexts using the `CREATE CONTEXT` statement, as per the following syntax:

```
CREATE CONTEXT [context_name]
USING [trusted_package]
INITIALIZED [EXTERNALLY | GLOBALLY]
ACCESSED [LOCALLY | GLOBALLY]
```

Trusted package is a PL/SQL package or stored procedure that controls the values of the context attributes. A database session-based context can be initialized locally, externally or globally. The context type can be `ACCESSED GLOBALLY` for global application contexts or `ACCESSED LOCALLY` (default) for session-based application contexts. The following example shows the creation of a context using a trusted package:

```
/*Connect as SYSDBA*/
CONN sys/oracle AS SYSDBA

Connected.

/*Grant CREATE ANY CONTEXT, DROP ANY CONTEXT privileges to SCOTT*/
GRANT CREATE ANY CONTEXT, DROP ANY CONTEXT TO SCOTT
/

Grant succeeded.

/*Connect to SCOTT user*/
CONN scott/tiger

Connected.
```

You can create the trusted PL/SQL program before or after creating the context. The following PL/SQL procedure `P_APP_CONTEXT` uses the Oracle supplied package `DBMS_SESSION` to create a context namespace and member attributes.

```
/*Create the stored procedure to set the context attribute*/
CREATE OR REPLACE PROCEDURE p_app_context (p_val VARCHAR2)
IS
BEGIN

    /*Create a namespace DEMO_CONTEXT*/
    DBMS_SESSION.SET_CONTEXT(
        NAMESPACE => 'DEMO_CONTEXT',
        ATTRIBUTE => 'COUNTRY',
        VALUE      => P_VAL);
END;
/
```

The PL/SQL procedure can now be used to create the application context.

```
/*Create the context*/  
CREATE CONTEXT demo_context USING p_app_context  
/
```

Context created.

Now, use the trusted program to set the value of the COUNTRY context key. Use the same program to modify the context attribute value.

```
/*Call P_APP_CONTEXT to set context value*/  
EXEC p_app_context('LUXEMBOURG')  
/  
PL/SQL procedure successfully completed.  
/*Query the COUNTRY context*/  
SELECT SYS_CONTEXT('DEMO_CONTEXT', 'COUNTRY')  
FROM DUAL  
/
```

```
SYS_CONTEXT('DEMO_CONTEXT', 'COUNTRY')
```

```
-----  
LUXEMBOURG
```

Virtual Private Database policy function

The Virtual Private Database policy function is a PL/SQL function that is used to construct the predicate (or WHERE) clause. The policy function is specified during policy creation that gets executed at the time of policy enforcement. Note that the function execution performance directly impacts the query performance.

The following are the features of the policy function:

- The schema name and database object name are mandatory parameters (and in the same sequence) to the policy function. Upon execution, the parameter values are supplied by the DBMS_RLS package.
- The function must return a valid predicate in string format.
- The function must not perform SELECT or DML on the table to be protected through the Virtual Private Database.

Policy types

You can specify a policy type to optimize the performance of a Virtual Private Database security policy. The policy performance is limited by the resources consumed during the execution of the policy function and frequency of its execution.

A Virtual Private Database policy can be `STATIC`, `DYNAMIC`, `SHARED_STATIC`, `CONTEXT_SENSITIVE`, or `SHARED_CONTEXT_SENSITIVE`. The following table shows the different VPD policy types:

Policy type	Comments	When to use
<code>STATIC</code>	<ul style="list-style-type: none"> Same predicate for all the queries Predicate clause is cached in SGA 	Used when all the queries on a table have a mandatory predicate and performance is the priority.
<code>DYNAMIC (Default)</code>	<ul style="list-style-type: none"> Policy function is executed for every query No performance optimization 	Used when the predicate condition changes for each query issued on the Virtual Private Database protected object or table.
<code>SHARED_STATIC</code>	<ul style="list-style-type: none"> Policy shareable across schema objects Predicate clause is cached in SGA 	Used when multiple tables or views have the same columns.
<code>CONTEXT_SENSITIVE</code>	<ul style="list-style-type: none"> Policy is applicable only to a fixed value of the application context Include the namespace and attribute parameters in the policy definition 	Used when the predicate varies by the user or group.
<code>SHARED_CONTEXT_SENSITIVE</code>	<ul style="list-style-type: none"> Same as the context-sensitive policy Policy can be shared among multiple objects 	Used when the predicate varies by the user and can be shared by multiple database objects.

The DBMS_RLS package

The security policies are enforced using the Oracle-supplied `DBMS_RLS` package. It can be used to add, drop, or refresh a security policy, enable or disable a policy, and handle the policy groups. It is owned by the `sys` user and available in enterprise edition only.



For complete details, it is recommended that you refer to the Oracle Database 12c documentation at https://docs.oracle.com/database/121/ARPLS/d_rls.htm.

The following table lists the subprograms contained in the DBMS_RLS package:

Subprogram	Description
ADD_POLICY	Adds a fine-grained access control policy to a table, view, or synonym
DROP_POLICY	Drops a fine-grained access control policy from a table, view, or synonym
REFRESH_POLICY	Causes all the cached statements associated with the policy to be reparsed
ENABLE_POLICY	Enables or disables a fine grained access control policy
CREATE_POLICY_GROUP	Creates a policy group
ADD_GROUPED_POLICY	Adds a policy associated with a policy group
ADD_POLICY_CONTEXT	Adds the context for the active application
DELETE_POLICY_GROUP	Deletes a policy group
DROP_GROUPED_POLICY	Drops a policy associated with a policy group
DROP_POLICY_CONTEXT	Drops a driving context from the object so that it will have one less driving context
ENABLE_GROUPED_POLICY	Enables or disables a row-level group security policy
DISABLE_GROUPED_POLICY	Disables a row-level group security policy
REFRESH_GROUPED_POLICY	Re-parses the SQL statements associated with a refreshed policy

Demonstration

This section demonstrates the implementation of a Virtual Private Database to enforce row-level and column-level security.

Let's suppose that SCOTT is a master user that owns the employee details. All other sub users are authorized to view the details of the employees that belong to their job roles. For example, CLERK is authorized to view only the clerk's details while SALES is authorized to view only salesman data.

Let's prepare the test environment by creating the test users and granting them required privileges:

```
/*Create user CLERK, MGR, SALES. Grant CONNECT, RESOURCE roles*/
CONN sys/oracle as sysdba
CREATE USER clerk IDENTIFIED BY clerk
/
CREATE USER mgr IDENTIFIED BY mgr
/
CREATE USER sales IDENTIFIED BY sales
/
GRANT CONNECT, RESOURCE TO clerk
/
GRANT CONNECT, RESOURCE TO mgr
/
GRANT CONNECT, RESOURCE TO sales
/
GRANT EXECUTE ON dbms_ols TO public
/
GRANT CREATE ANY CONTEXT TO scott
/
GRANT CREATE PUBLIC SYNONYM TO scott
/
```

Let's create the MYJOB context using a trusted program:

```
CONN scott/tiger

/*Create the context MYJOB*/
CREATE CONTEXT myjob USING scott.job_context
/
```

The following JOB_CONTEXT procedure sets the MYJOB context attributes, based on the current session user. The context attribute values should match the expected value to be used in the query predicates:

```
/*Create the PL/SQL procedure */
CREATE OR REPLACE PROCEDURE job_context IS
BEGIN

/*Set the context based on the session user*/
IF SYS_CONTEXT ('USERENV', 'SESSION_USER') = 'CLERK' THEN
    DBMS_SESSION.SET_CONTEXT ('myjob','role','CLERK');

ELSIF SYS_CONTEXT ('USERENV', 'SESSION_USER') = 'SALES' THEN
    DBMS_SESSION.SET_CONTEXT ('myjob','role','SALESMAN');
```

```
ELSIF SYS_CONTEXT ('USERENV', 'SESSION_USER') = 'MGR' THEN
    DBMS_SESSION.SET_CONTEXT ('myjob', 'role', 'MANAGER');
END IF;

END;
/
```

As this procedure will be accessed by SALES, CLERK, and MGR, let's create a public synonym and grant the execute privilege to the users:

```
/*Create synonym for JOB_CONTEXT procedure*/
CREATE PUBLIC SYNONYM job_context FOR scott.job_context
/
GRANT EXECUTE ON scott.job_context TO clerk, mgr, sales
/
/*Grant select privilege on EMP table to the users */
GRANT SELECT ON emp TO clerk, mgr, sales
/
```

To set the context at the time of the database logon, let's create a logon trigger. This trigger calls the context setting procedure:

```
CONN sys/oracle AS SYSDBA

/*Create a logon trigger*/
CREATE OR REPLACE TRIGGER scott.set_security_context
AFTER LOGON ON DATABASE
BEGIN
    scott.job_context;
END;
/
```

Now, it's time to create the Virtual Private Database policy function. It lays down the logic of row-level security by constructing the predicate clause. In this case, the security policy virtually partitions the data view by job roles. The predicate uses the SYS_CONTEXT function to retrieve the job role corresponding to the session user.

```
CONN scott/tiger

/*Create policy function */
CREATE OR REPLACE FUNCTION fun_vpd
(schemaowner VARCHAR2, objectname VARCHAR2)
RETURN VARCHAR2 IS
BEGIN
```

```

/*Return the predicate clause*/
RETURN 'job = SYS_CONTEXT(''myjob'', ''role'')';
END;
/

/*Create public synonym for the policy function */
CREATE PUBLIC SYNONYM fun_vpd FOR scott.fun_vpd
/
GRANT EXECUTE ON scott.fun_vpd TO clerk, mgr, sales
/

```

The final step is the creation of policy on the object using the DBMS_RLS.ADD_POLICY procedure:

```

conn scott/tiger
BEGIN
  DBMS_RLS.add_policy(
    object_schema =>'SCOTT',
    object_name   =>'EMP',
    policy_name   =>'VPD_RLS',
    function_schema =>'SCOTT',
    policy_function =>'FUN_VPD',
    statement_types =>'SELECT');
END;
/

```

Let's connect to the MGR user and verify the Virtual Private Database in operation:

```

conn mgr/mgr
SELECT empno, ename, job, mgr, deptno
FROM scott.emp
/

```

EMPNO	ENAME	JOB	MGR	DEPTNO
7566	JONES	MANAGER	7839	20
7698	BLAKE	MANAGER	7839	30
7782	CLARK	MANAGER	7839	10

The CLERK user can query the details of only those employees that are clerks. Similarly, the other users can try to log in the database and verify the data view:

```

conn clerk/clerk
SELECT empno, ename, job, mgr, deptno
FROM scott.emp
/

```


EMPNO	ENAME	JOB	MGR	DEPTNO
7369	SMITH	CLERK	7902	20
7876	ADAMS	CLERK	7788	20
7900	JAMES	CLERK	7698	30
7934	MILLER	CLERK	7782	10

And then the SALES user:

```
conn sales/sales
SELECT empno, ename, job, mgr, deptno
FROM scott.emp
/
```

EMPNO	ENAME	JOB	MGR	DEPTNO
7499	ALLEN	SALESMAN	7698	30
7521	WARD	SALESMAN	7698	30
7654	MARTIN	SALESMAN	7698	30
7844	TURNER	SALESMAN	7698	30

You can enforce a column-level policy by specifying the sensitive columns at the time of the policy creation. The following PL/SQL block drops the last Virtual Private Database policy and creates a new one to enforce the row-level security:

```
conn scott/tiger
BEGIN

  /*Drop the row-level policy*/
  DBMS_RLS.DROP_POLICY ('SCOTT','EMP','VPD_RLS');

  /*Create a column level VPD policy*/
  DBMS_RLS.ADD_POLICY(
    object_schema => 'SCOTT',
    object_name    => 'EMP',
    policy_name     => 'VPD_COLUMN',
    policy_function => 'FUN_VPD',
    sec_relevant_cols => 'sal,comm');
END;
/
```

The preceding security policy specifies that whenever a user will try to query the SAL and COMM columns of the EMP table, the row-level security will be enforced. Let's verify this statement by connecting to the MGR user and querying the employee records:

```
CONN mgr/mgr
/*Select non-sensitive columns from EMP table*/
SELECT empno, deptno, job
FROM scott.emp
/
```

EMPNO	DEPTNO	JOB
7369	20	CLERK
7499	30	SALESMAN
7521	30	SALESMAN
7566	20	MANAGER
7654	30	SALESMAN
7698	30	MANAGER
7782	10	MANAGER
7788	20	ANALYST
7839	10	PRESIDENT
7844	30	SALESMAN
7876	20	CLERK
7900	30	CLERK
7902	20	ANALYST
7934	10	CLERK

14 rows selected.

Note that the preceding query displays all the employees because the SELECT query didn't project the SAL and COMM sensitive columns. If the query had contained the SAL column, the output would have been as follows:

```
/*Select sensitive columns from EMP table*/
SELECT empno, deptno, sal, job
FROM scott.emp
/
```

EMPNO	DEPTNO	SAL	JOB
7566	20	2975	MANAGER
7698	30	2850	MANAGER
7782	10	2450	MANAGER

In the preceding output, the employee records with the sensitive columns are restricted in the query output. However, you can still list all the employees and mask the SAL and COMM sensitive columns with NULLs by setting the SEC_RELEVANT_COLS_OPT attribute to DBMS_RLS.ALL_ROWS:

```
CONN scott/tiger
BEGIN

  /*Drop the previous policy*/
  DBMS_RLS.DROP_POLICY ('SCOTT', 'EMP', 'VPD_COLUMN');

  /*Create a new column-level policy*/
  DBMS_RLS.ADD_POLICY(
    object_schema => 'SCOTT',
    object_name   => 'EMP',
    policy_name   => 'VPD_COLUMN',
    policy_function => 'FUN_VPD',
    sec_relevant_cols => 'sal,comm',
    sec_relevant_cols_opt => DBMS_RLS.ALL_ROWS);
END;
/
```

With the preceding security policy, the output of the SELECT query will be as follows:

```
CONN mgr/mgr

/*Select sensitive columns from EMP table*/
SELECT empno, ename, job, sal, deptno
FROM scott.emp
ORDER BY job
/
```

EMPNO	ENAME	JOB	SAL	DEPTNO
7788	SCOTT	ANALYST		20
7902	FORD	ANALYST		20
7934	MILLER	CLERK		10
7900	JAMES	CLERK		30
7369	SMITH	CLERK		20
7876	ADAMS	CLERK		20
7698	BLAKE	MANAGER	2850	30
7566	JONES	MANAGER	2975	20
7782	CLARK	MANAGER	2450	10
7839	KING	PRESIDENT		10
7844	TURNER	SALESMAN		30

7654	MARTIN	SALESMAN	30
7521	WARD	SALESMAN	30
7499	ALLEN	SALESMAN	30

14 rows selected.

Virtual Private Database features and best practices

The following are the features and best practices that can be followed while working with a Virtual Private Database:

- You cannot query a Virtual Private Database protected table as `SELECT FOR UPDATE` because of the query rewrite implementation. However, the `FOR UPDATE` user query may work provided the VPD implied inline view for the query is not a complex one, that is, a non-analytic query, with no `DISTINCT`, no cursor expression, no `SET` operator.
- `SYS` user is exempted from the Virtual Private Database security policies. However, the `SYSDBA` actions can be monitored using Database Vault.
- Users with the `EXEMPT ACCESS POLICY` system privilege are exempt from the Virtual Private Database security policies. This privilege must be used judiciously as it bypasses all the fine-grained security controls.
- The policy function should be executed with definer's rights in major cases to avoid any uncertainty due to privileges.
- Avoid using outer joins and the `ANSI` join operations on the Virtual Private Database protected tables.
- If you are using edition-based redefinition in your application, you can associate an editioning view to apply the Virtual Private Database policies across all the editions.
- Virtual Private Database works with flashback query. You can query the business data to a past time or older `SCN` using the most recent security policies.
- During a direct path export operation, the Oracle Virtual Private Database policies (and Oracle Label Security policies) are not enforced.
- A schema level export operation fails if the schema contains a VPD protected object. In a scenario where a user has a valid justification to bypass security policies, he can be granted `EXEMPT ACCESS POLICY` privilege (without `WITH ADMIN OPTION`) to ignore VPD policies.

Virtual Private Database metadata

Oracle captures the static and dynamic details of the VPD policies in dictionary views. You can query the following dictionary views to find the metadata about the Virtual Private Database security policies:

Dictionary view	Comments
[ALL USER DBA]_POLICIES	The view captures the security policy on objects accessible to a user, owned by a user, or within a database.
[ALL USER DBA]_POLICY_ATTRIBUTES	The view captures the application context namespaces, their attributes, and their association with the Virtual Private Database policy.
[ALL USER DBA]_POLICY_CONTEXTS	The view captures information about the driving contexts for the objects.
[ALL USER DBA]_POLICY_GROUPS	The view captures information about the policy groups on the objects.
[ALL USER DBA]_SEC_RELEVANT_COLS	The view captures the specifications of the column-level Virtual Private Database policy on the objects.
V\$VPD_POLICY	The dynamic view captures information about the security policy associated with the cached cursors. In a multitenant environment, the view shows information about the current pluggable database. It's extremely useful in troubleshooting policy execution.

The following query gives the predicate applied to the queries executed against the Virtual Private Database protected table owned by the SCOTT user:

```
connect sys/oracle as sysdba
SELECT object_name,
       policy,
       s.sql_text,
       predicate
FROM v$vpd_policy vp, v$sql s
WHERE vp.sql_id=s.sql_id
AND object_owner='SCOTT'
/
```

OBJECT_NAME	POLICY	SQL_TEXT	PREDICATE
EMP	VPD_COLUMN	select * from scott.emp	job = SYS_
CONTEXT('myjob','role')			

For further information about the VPD_COLUMN policy, you can refer to the USER_POLICIES dictionary view:

```
connect scott/tiger
SELECT policy_name,
       policy_type,
       static_policy,
       function,
       pf_owner
FROM USER_POLICIES
/
```

POLICY_NAME	POLICY_TYPE	STA	FUNCTION	PF_OWNER
VPD_COLUMN	DYNAMIC	NO	FUN_VPD	SCOTT

Policy utilities—refresh and drop

The policy utility activities such as refreshing or dropping can be done through the DBMS_RLS package subprograms. Refreshing a policy inherits the latest changes made to the policy and its dependents. A policy refresh is required when the underlying referenced objects of the policy undergo any change that invalidates the dependent objects. During the policy refresh process, all the cached statements associated with the policy are parsed again. A policy that is already disabled cannot be refreshed.

A Virtual Private Database policy can be dropped using the DROP_POLICY subprogram of the DBMS_RLS package:

```
EXEC DBMS_RLS.DROP_POLICY('SCOTT','EMP','VPD_COLUMN')
/
```

Oracle Database 12c Security enhancements

Oracle Database 12c introduced a number of features and enhancements to further strengthen the in-depth security collateral. In this section, you will understand some of the key security enhancements and features in Oracle Database 12c. The following is a summary of the new security features:

- **Real Application Security(RAS):** Oracle Database 12c introduces a data authorization solution to provide end-to-end security in a multitier application architecture. You can now declare and enforce the application-level security policies in the database kernel. The RAS security model understands the application-level security policy constructs, such as application users, privileges, and roles within the database, and enforces the security policies in the context of the application. As well as data access, RAS can help the applications to secure the access control operations of an application user.
- **Oracle Data Redaction:** The Data Redaction feature masks the sensitive data on-the-fly before it is displayed to the user. This feature provides a rich library for masking the sensitive information while the data in the storage remains in an actual, that is, unmasked format.
- **Role and privilege analysis:** Privilege analysis helps to enhance operational security by identifying the least set of privileges to run an application. During application development, you may have granted system privileges `SELECT ANY TABLE` to a user or object privileges to the `PUBLIC` role. However, before moving it to production, you must verify the security aspect of the application by retaining the required privileges and revoking the unnecessary ones. You can now analyze the roles, users or contexts, sessions or database to detect used and unused privileges. A privilege is considered as unused if it is not utilized for compilation or run-time operations.

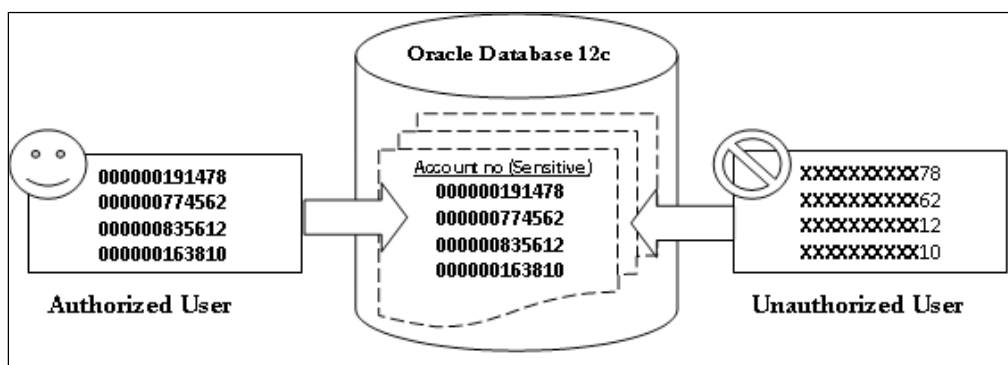
A user with the `CAPTURE_ADMIN` role can kick-off a privilege analysis operation. Note that a `SYS` user cannot be analyzed under privilege analysis.

- **Unified Auditing:** Oracle Database 12c introduces a unified audit trail which resolves the pain to maintain different stores of audit information. The new `UNIFIED_AUDIT_TRAIL` data dictionary view replaces the previous auditing views and consolidates the audit information from all the sources. The new `AUDIT_VIEWER` and `AUDIT_ADMIN` roles enable the separation of duty and provide easy management of auditing within the database. The new auditing framework supports conditional auditing to prevent the auditing of irrelevant operations.

- **New administrative privileges such as SYSBACKUP, SYSDG, and SYSKM:** These are the new privileges for dedicated tasks like backup, data guard operations, and key management in order to reduce dependency on SYSDBA and enable the separation of duties.
- **The READ object privilege and the READ ANY TABLE system privilege:** You can now grant the READ object privilege that allows a user to query tables, views, materialized views, and synonyms. The difference between the SELECT and READ object privileges is that SELECT allows a user to lock the table data using the LOCK statement or the FOR UPDATE clause, while READ is only for query purpose.
- **The ADMIN and DELEGATE options for code-based access control:** You can now protect your PL/SQL programs by granting roles to the program units and not to the users. You can grant the roles with the ADMIN or DELEGATE option to a program unit. With the DELEGATE option, the grantee is not allowed to grant the role to the other programs.

Oracle Database 12c Data Redaction

Oracle Database 12c Data Redaction prevents the exposure of sensitive data to the non-authorized users. The beauty of this feature lies in the fact that it enables dynamic masking, which means that the data is never changed on the storage or cache but gets redacted at runtime just before it is displayed to the user. The security policies are created in the database and are consistently active to hide sensitive information whenever requested by the applications. In addition, this feature has no impact on the integrity constraints and database operations such as the backup, restore, export, import, grid, and cluster operations.



The Oracle Database 12c Data Redaction feature has been backported to Oracle Database 11.2.0.4.

A typical use case of Data Redaction would be a read-only application that fetches the data from a production warehouse. Similarly, the redaction policies can be created for a call center application. There will be many such cases where data view is based on the user authorization.

Data Redaction exemptions and miscellaneous features

Data Redaction is a part of the Advanced Security option, which can be purchased with Oracle Database Enterprise Edition. The following are the salient features of Data Redaction:

- The `SYS` and `SYSTEM` users are always exempt from the redaction policies.
- The virtual columns cannot be redacted.
- The editioning views cannot be redacted.
- Users with the `EXEMPT REDACTION POLICY` system privilege are exempt from the redaction policies.
- You cannot perform the **CTAS (CREATE TABLE AS SELECT)** operation on a redacted table.
- A redaction policy can be controlled using the `DBMS_REDACT` package. A user must have the `EXECUTE` privilege on the package.
- Data Redaction policies can be created from Enterprise Manager Cloud Control 12c.

Data Redaction function types

Data Redaction can use the following methods of redaction:

- **Full redaction:** Full redaction refers to the complete replacement of a column value. A number type value will be replaced with zero or a dummy date can replace a date. For example, 15/01/2015 (and all the date values) can be redacted as 01/01/00.
- **Partial redaction:** Partial redaction refers to the masking of a portion of a column value. For example, the first few digits of telephone numbers or social security numbers are replaced with X. The masking format shown in the preceding figure depicts partial redaction.

- **Regular expressions:** You can use regular expressions to mask the data based on a pattern and not on the basis of a fixed length of the characters. For example, email addresses or credit card numbers can be masked based on the position of the special character.
- **Random redaction:** You can direct the redaction policy to generate a random value with a matching data type for the column carrying the sensitive data.
- **No redaction:** This is used for testing purposes and can be used before deploying the security policies during production.

Demonstration

Let's continue the case study from the Virtual Private Database demonstration. Note that EMP is a Virtual Private Database protected table. With Data Redaction, the objective will be to mask the values from the SAL and COMM columns of the EMP table for the CLERK, MGR, and SALES users.

The following PL/SQL block creates a redaction policy on the SAL column of the EMP table. As the redaction is for the access from three users, that is, SALES, MGR, and CLERK, the predicate expression using the SYS_CONTEXT function has been included.

```
CONNECT sys/oracle AS SYSDBA
/*Grant execute privilege on DBMS_REDACT to scott */
GRANT EXECUTE ON sys.dbms_redact TO SCOTT
/
CONNECT scott/tiger
BEGIN

/*Add a policy to redact SAL column */
DBMS_REDACT.add_policy(
  object_schema => 'SCOTT',
  object_name   => 'EMP',
  column_name   => 'SAL',
  policy_name   => 'redact_employee_info',
  function_type => DBMS_REDACT.full,
  expression    => q'|sys_context('userenv','current_user') IN
('SALES','MGR','CLERK')|'
);
END;
/
```

Let's verify the impact of the redaction policy. The SAL column is fully redacted by replacing the original column values by a default value, zero. Note that the Virtual Private Database policy is still active and enforced:

```
CONN mgr/mgr
SELECT empno, ename, job, sal, deptno
FROM scott.emp
/
```

EMPNO	ENAME	JOB	SAL	DEPTNO
7566	JONES	MANAGER	0	20
7698	BLAKE	MANAGER	0	30
7782	CLARK	MANAGER	0	10



For a full redaction policy type, the default values for the columns are picked up from the REDACTION_VALUES_FOR_TYPE_FULL dictionary view.

You can alter a redaction policy using the ALTER_POLICY procedure from the DBMS_REDACT package. The ACTION attribute is used to record the changes that you make while altering the policy.

Let's alter the redaction policy and mask the HIREDATE column value. This time, we will use the partial redaction technique to mask the date value.

```
CONNECT scott/tiger
BEGIN

/*Alter policy to include HIREDATE column */
DBMS_REDACT.alter_policy (
  object_schema    => 'SCOTT',
  object_name      => 'EMP',
  policy_name      => 'redact_employee_info',
  action           => DBMS_REDACT.add_column,
  column_name      => 'hiredate',
  function_type     => DBMS_REDACT.partial,
  function_parameters => 'm1d1y1900'
);
END;
/
```

Now, let's connect to the SALES user and check the masked values of the HIREDATE column:

```
CONN sales/sales
SELECT empno, ename, job, sal, hiredate
FROM scott.emp
/
```

EMPNO	ENAME	JOB	SAL	HIREDATE
7499	ALLEN	SALESMAN	0	01-JAN-00
7521	WARD	SALESMAN	0	01-JAN-00
7654	MARTIN	SALESMAN	0	01-JAN-00
7844	TURNER	SALESMAN	0	01-JAN-00

Let's mask another column to demonstrate the redaction using regular expression. We'll alter the EMP table to add the EMAIL column:

```
CONNECT scott/tiger
ALTER TABLE emp
ADD (email VARCHAR2 (30))
/
UPDATE emp
SET email = LOWER (ename) || '@xyz.com'
/
COMMIT
/
```

The following PL/SQL block masks the EMAIL column using a regular expression:

```
BEGIN

/*Alter policy to include EMAIL column */
DBMS_REDACT.alter_policy (
  object_schema      => 'SCOTT',
  object_name        => 'EMP',
  policy_name        => 'redact_employee_info',
  action             => DBMS_REDACT.add_column,
  column_name        => 'email',
  function_type       => dbms_redact.regexp,
  regexp_pattern     => dbms_redact.RE_PATTERN_EMAIL_ADDRESS,
  regexp_replace_string => dbms_redact.RE_REDACT_EMAIL_NAME
);
END;
/
```

The preceding policy includes the EMAIL column and masks it using a regular expression. Note the regular expression pattern and replace the string constants. The DBMS_REDACT package includes a list of the standard security constructs that can be readily used while creating the redaction policy. Let's connect to CLERK user and check the email column:

```
CONN clerk/clerk
SELECT empno, ename, job, sal, hiredate, email
FROM scott.emp
/
```

EMPNO	ENAME	JOB	SAL	HIREDATE	EMAIL
7369	SMITH	CLERK	0	01-JAN-00	xxxx@xyz.com
7876	ADAMS	CLERK	0	01-JAN-00	xxxx@xyz.com
7900	JAMES	CLERK	0	01-JAN-00	xxxx@xyz.com
7934	MILLER	CLERK	0	01-JAN-00	xxxx@xyz.com

The Data Redaction metadata

You can query the following dictionary views to find the metadata information about the redaction policies:

- REDACTION_POLICIES: This view stores the redaction policy details
- REDACTION_COLUMNS: This view stores the details of redacted columns in the database

The following two queries display the redaction policy details that were created in the preceding demonstration:

```
CONNECT sys/oracle AS SYSDBA
SELECT policy_name, expression
from REDACTION_POLICIES
/
```

POLICY_NAME	EXPRESSION
redact_employee_info	sys_context('userenv','current_user') IN ('SALES','MGR','CLERK')

```
SELECT object_name, column_name,function_type,function_parameters
FROM redaction_columns
/
```

OBJECT_NAME	COLUMN_NAME	FUNCTION_TYPE	FUNCTION_PARAMETERS
EMP	HIREDATE	PARTIAL REDACTION	m1d1y1900
EMP	SAL	FULL REDACTION	
EMP	EMAIL	REGEXP REDACTION	

Summary

In this chapter, we covered the fundamentals of database security solutions from Oracle. After an overview of the Oracle Database Security solution, we dived into the fine-grained access control and Virtual Private Database. The Virtual Private Database enforces row-level security through the policies and restricts the data access for unauthorized users. Depending on a user's identity and role, the application can set up multitenancy and ensure user isolation as well.

Oracle Database 12c made considerable enhancements to its security offering. The summary of these enhancements was included, while the Data Redaction feature was covered in detail along with demonstrations. In the next chapter, we will dive into another key area that has continuously gained more weight with time: handling of large objects in the Oracle Database. We will be focusing the majority of our discussion around SecureFiles and its optimizations.

Practice exercise

- Identify the correct statements about the working of Fine Grained Access Control.
 1. A table can have only one security policy.
 2. Different policies can be used to protect the `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements on a table, but not one.
 3. The policy function returns the predicate information as `WHERE <Column> = <Value>`.
 4. Once associated, the FGAC policy cannot be revoked from the table.
- A security policy can be associated to a group of objects by the DBA. State whether this is true or false.
 1. True
 2. False

- Choose the correct statement about DBMS_RLS.
 1. DBMS_RLS is used only for row-level security policies.
 2. The package is owned by SYS.
 3. It can create / drop / refresh policies and create/drop policy groups.
 4. Using DBMS_RLS to set the policy degrades the application performance.
- Identify the correct statements about the context of an application.
 1. A user who holds the CREATE CONTEXT privilege can create a context.
 2. It is owned by the SYS user.
 3. A user can check the context metadata in USER_CONTEXTS.
 4. The trusted package associated with the context must exist before the context is created.
- Arrange the sequence of the Virtual Private Database implementation using application contexts.
 - i. Creating policy function.
 - ii. Creating trusted package.
 - iii. Creating and setting application contexts.
 - iv. Associating a policy using DBMS_RLS.
 1. iii, ii, i, iv
 2. ii, iii, iv, i
 3. iii, iv, i, ii
 4. iv, i, ii, iii
- All the policies on different columns of the same table are collectively known as policy groups.
 1. True
 2. False
- Identify the correct statements about the policy types.
 1. Shared static policy is an extension to the static policies where multiple static policies can be shared among multiple users.
 2. Shared static policy is an extension to the static policies where a single static policy can be shared among multiple objects.
 3. STATIC is the default policy type.
 4. DYNAMIC is the default policy type.

- Pick the correct statement about the application contexts.
 1. Only a DBA can create a custom application context and add attributes under it.
 2. The DBA can modify all `USERENV` attributes.
 3. The package used for the context creation may or may not exist in the schema.
 4. Global contexts can be used by all the users on a server.
- A policy of the `CONTEXT_SENSITIVE` type executes the policy function once, every time the query is reparsed, if the local context has been changed.
 1. True
 2. False
- Identify the cause of the following exception:

```
SQL>SELECT * FROM employees;
select * from employees
          *
ERROR at line 1:
ORA-28110: policy function or package ORADEV.F_JOB_POLICY has
error
```

 1. The policy function `F_JOB_POLICY` does not exist.
 2. The policy function `F_JOB_POLICY` has not been specified in `DBMS_RLS.ADD_POLICY` to add the policy on the `employees` table.
 3. The predicate returned by the policy function is not appropriate for this query.
 4. The Virtual Private Database policy on the `employees` table is invalid and has errors.

7

Oracle SecureFiles

The fact that data is growing multifold at a tremendous speed has led to the evolution of a number of data management trends and Big Data is one of them. A large amount of information from web content, sensors, documents, images, and location services has pushed organizations to place serious thought to data management strategies and distil out nuggets of information. The data can be structured, semi-structured, or unstructured.

In relational database terminology, unstructured data such as binary files or documents are classified as large objects. Oracle has been allowing users to store large objects in the tables of a relational database for a long time. While the large objects are prototyped as BLOB or CLOB in a table, there is a provision to store XML documents, spatial data and text using dedicated data types. In this chapter, we will focus on large object handling in the Oracle Database. You will understand how large objects or LOBs are stored in the database, what the optimization features are, and how it has evolved over time. The Oracle Database 11g introduced a new format of the LOB storage known as SecureFiles. SecureFiles offer optimized storage and enable a better performance of the large objects in an Oracle database.

The chapter is outlined as follows:

- Introduction to Large Objects
- Classification of Large Objects
- Working with LOB data types
 - Creating LOB data types
 - Managing LOB data types
 - Migrating LONG to LOB

- Oracle SecureFiles
 - An overview
 - Working with Securefiles
 - Migrating BasicFiles to SecureFiles
- Oracle Database 12c SecureFiles enhancements

Introduction to Large Objects

As the name suggests, large objects or LOBs refer to large data. A column of a large object type in a table can store semi-structured or unstructured data. Semi-structured data can be a character-based document that can be processed in a near relational format. Unstructured data is a binary file that is difficult to interpret logically. For example, an XML file is a semi-structured document while an image or a graphics file is an unstructured format of the data.

Oracle Database supports the storage of large objects along the following aspects:

- **Storage:** Just like any other data, the Oracle Database allows the storage of large objects in columns within a table. The columns of LOB data types can efficiently store a semi or unstructured data object, compress it and even encrypt it in the database. The LOB datatype stored in the Oracle Database abides by the **ACID (Atomicity, Consistency, Isolation, and Durability)** properties. Oracle provides a wide range of administrative controls to manage and maintain large objects in the database.
- **Access:** Oracle SecureFiles (discussed later in this chapter) provides you with highly optimized access of large objects from the Oracle Database. SecureFiles, introduced in Oracle Database 11g and the default LOB storage option in Oracle Database 12c, accelerates LOB performance through vector optimization techniques. For semi-structured data types such as Oracle Text or Oracle Spatial, Oracle enables indexing techniques to improve query performance.
- **Security:** The LOB data type in the Oracle Database stays protected and secure. Fine-grained data access security policies apply to large objects as well.

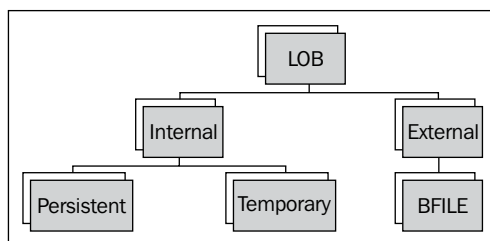
The use of the `LONG` and `LONG RAW` data types was succeeded by the `LOB` data types in Oracle 8i. Although `LONG` and `LONG RAW` are still supported, Oracle recommends the usage of the `LOB` data types in place of `LONG` and `LONG RAW` due to the following reasons:

- A table can have only one `LONG` or `LONG RAW` column
- A `LONG` or `LONG RAW` column can store data only up to 2 GB
- The `LONG` data can only be accessed sequentially

It is recommended for legacy applications using `LONG` or `LONG RAW`, to migrate to `LOBs` so as to leverage the benefits of storage optimization and enhanced performance.

Classification of Large Object datatypes

A `LOB` data type can be classified on the basis of its storage properties. The following figure branches out the different `LOB` types in Oracle:



Internal LOB

An internal `LOB` can be stored in and accessed from the database. A column of `BLOB`, `CLOB`, or `NCLOB` can be included in a table to store large objects, like all other columns. Note that the internal `LOB` data types can be used in PL/SQL as well.

Persistent and Temporary LOB

A `LOB` data type is persistent if it is physically stored in the database tables. As with all other column values, it can be a part of a database transaction.

A temporary `LOB` is used to instantiate a `LOB` in a PL/SQL program. It is used to perform the `LOB` operations, which are carried out in a temporary tablespace. If the temporary `LOB` value is inserted into a database table, the `LOB` instance becomes persistent.

External LOB

If the LOB data does not reside inside the Oracle Database, it is an External LOB. A column of an external LOB data type, `BFILE`, stores only the LOB locator and not the data. The data resides outside the database storage as an operating system file on the server host. The integrity and durability of external LOBs must be layered at the operating system level.



`BFILE` is a read-only data type and does not participate in database transactions.

LOB restrictions

A LOB is a special feature that allows a database management system to store unstructured data such as files and documents. However, there are certain restrictions on the usage of a LOB and its related operations, listed as follows:

- A LOB column cannot be a primary key of a table
- A LOB column cannot be used in the `ORDER BY` clause of `SELECT` query
- A LOB column cannot appear in the `GROUP BY` clause of a `SELECT` query
- A LOB column cannot be used to join two tables
- You cannot use the `DISTINCT` clause with a LOB column in a `SELECT` query
- A LOB column cannot be selected in a `UNION` query
- A B-Tree index or a bitmap index cannot be created on a LOB column

LOB data types in Oracle

Oracle provides four data types dedicated for declaring large objects, namely: `BLOB`, `CLOB`, `NCLOB`, and `BFILE`.

BLOB and CLOB

There are three types of internal LOB data types, namely:

- `BLOB`: The **Binary Large Object** data type is used to store large binary files that cannot be logically broken down to data bits such as PDFs, images, audios or videos, and so on.

- **CLOB:** The **Character Large Object** data type stores the single-byte character data in the database character set format. It supports fixed-width character formats.
- **NCLOB:** The CLOB data type that can store national character set data and support varying width format character sets.



Starting from Oracle 10g, Oracle can cast the CLOB data to the VARCHAR2 data implicitly.

BFILE

A **BFILE** is an SQL data type for the external LOB data type. It is the read-only data type that stores a locator for a binary file whose physical location is outside the Oracle Database. Deleting a **BFILE** column value (or setting it to **NULL**) will drop the reference pointer to the external file but will not delete the file.



The security and integrity of the externally located file must be managed and advised at the operating system level. Oracle recommends the usage of database directories in order to secure the user actions to the file location. A session-level initialization parameter `SESSION_MAX_OPEN_FILES` determines the maximum number of **BFILES** that can be opened in a session. The default parameter value is 10 but this can be modified using the `ALTER SESSION` command. If you want to close all the open files, you can do so by calling `DBMS_LOB.FILECLOSEALL`.

Some more related stuff

While working with the LOB data types, you must be familiar with a few more terms such as the LOB instance, LOB initialization, and the `DBMS_LOB` package. A thorough understanding is essential to perform the LOB operations.

The LOB locator

Structurally, a LOB instance is made up of a LOB locator and value. The LOB locator stores the pointer to the LOB value while the LOB value stores the actual physical data.

Be it an internal or external LOB, the column in the table holds just the locator. In the case of **BLOB**, **CLOB**, and **NCLOB**, the LOB locator is used to retrieve the value from the LOB instance. For **BFILE**, the locator is used to retrieve the externally located operating system file.

LOB instance initialization

The state of a LOB column can be NULL, empty, or populated. A LOB instance without a locator is a NULL LOB and cannot be passed as an argument to a program. To allow a LOB to be passed as a parameter to a program, it must be initialized. You can initialize a LOB instance of a persistent LOB column using the Oracle-supplied constructor functions, `EMPTY_BLOB ()` and `EMPTY_CLOB ()` for BLOB and CLOB respectively. The functions initialize the LOBs but do not populate them with any data. These functions can be used in the following scenarios:

- The LOB column default can be set to one of the constructor functions in a `CREATE TABLE` statement:

```
CREATE TABLE t_LOB_init
(id NUMBER,
 b_LOB BLOB DEFAULT EMPTY_BLOB (),
 c_LOB CLOB DEFAULT EMPTY_CLOB ()
)
/
```

Similarly, the column default can be set while adding a column using the `ALTER TABLE` command.

- An empty LOB instance can be inserted into the LOB column in an `INSERT` statement. For example, the following `INSERT` statement uses the `EMPTY_BLOB ()` function or the empty locator instead of a profile image:

```
/*Insert test data in the above table*/
INSERT INTO t_LOB_init (ID, B_LOB, C_LOB)
VALUES
(129, EMPTY_BLOB (), EMPTY_CLOB())
/
```

- The LOB variables in a PL/SQL block can be initialized either in the declarative section or program body:

```
/*Start the PL/SQL block*/
DECLARE

/*Declare local LOB variables*/
l_my_cLOB CLOB := EMPTY_CLOB ();
l_my_bLOB BLOB := EMPTY_BLOB ();

BEGIN
...
--executable section
...
```

```
END;
/
```

PL/SQL procedure successfully completed.


You can initialize a BFILE column using the BFILENAME function. The BFILENAME function accepts the database directory and file as inputs and returns a BFILE locator. A BFILE locator is a reference to the file located externally in the operating system. The syntax of the BFILENAME function is as follows:

```
FUNCTION BFILENAME(directory IN VARCHAR2, filename IN VARCHAR2)
RETURN BFILE;
```

The BFILENAME function is also used while populating a LOB instance from an external file in a PL/SQL block.

The DBMS_LOB package

Oracle provides a built-in package, DBMS_LOB, that allows users to perform transactional operations on the internal LOBs and read operations for BFILES. The package is owned by SYS. A user with the EXECUTE privilege on the DBMS_LOB package can invoke its subprograms.

 For the following subsections, refer to the Oracle Database Online Documentation 12c Release 1 (12.1)/Database Administration at https://docs.oracle.com/database/121/ARPLS/d_LOB.htm.

The DBMS_LOB constants

The DBMS_LOB package constants have been consolidated in the following table:

Constant	Description	Value
The DBMS_LOB constants		
FILE_READONLY	Open BFILE in read-only	0
LOB_READONLY	Read-only LOB	0
LOB_READWRITE	Read write LOB	1
LOBMAXSIZE	Maximum size of LOB	18446744073709551615
SESSION	Create temporary LOB for the duration of a session	10
CALL	Create temporary LOB for the duration of a transaction	12

Constant	Description	Value
The DBMS_LOB option types		
OPT_COMPRESS	SecureFile compression option	1
OPT_ENCRYPT	SecureFile encryption option	2
OPT_DEDUPLICATE	SecureFile deduplicate option	4
The DBMS_LOB option values		
COMPRESS_OFF	SecureFile compression OFF	0
COMPRESS_ON	SecureFile compression ON	1
ENCRYPT_OFF	SecureFile encryption OFF	0
ENCRYPT_ON	SecureFile encryption ON	2
DEDUPLICATE_OFF	SecureFile deduplication OFF	0
DEDUPLICATE_ON	SecureFile deduplication ON	4
The DBFS state value types		
DBFS_LINK_NEVER	Non-archived LOB	0
DBFS_LINK_NO	LOB archived but read by the database	2
DBFS_LINK_YES	LOB archived	1
The DBFS Cache Flags		
DBFS_LINK_CACHE	Archive and cache the LOB data	1
DBFS_LINK_NOCACHE	Archive the LOB data without caching it in the database	0

The DBMS_LOB data types

The DBMS_LOB package works with the following data types:

Data type	Description
BLOB	Source or destination binary LOB
RAW	Source or destination RAW buffer (used with BLOB)
CLOB	Source or destination character LOB (including NCLOB)
VARCHAR2	Source or destination character buffer (used with CLOB and NCLOB)
INTEGER	Specifies the size of a buffer of a LOB
BFILE	Large, binary object stored outside the database

The DBMS_LOB subprograms

The DBMS_LOB subprograms are listed in the following table:

Subprogram	Type	Description
APPEND	Procedure	Appends the contents of the LOB source to the LOB destination
CLOSE	Procedure	Closes a previously opened internal or external LOB
COMPARE	Function	Compares two entire LOBs or parts of two LOBs
CONVERTTOBLOB	Procedure	Reads the character data from a source CLOB or NCLOB instance, converts the character data to the specified character, writes the converted data to a destination BLOB instance in binary format, and returns the new offsets
CONVERTTOCLOB	Procedure	Takes a source BLOB instance, converts the binary data in the source instance to the character data using the specified character, writes the character data to a destination CLOB or NCLOB instance, and returns the new offsets
COPY	Procedure	Copies all, or a part, of the LOB source to the LOB destination
CREATETEMPORARY	Procedure	Creates a temporary BLOB or CLOB and its corresponding index in the user's default temporary tablespace
ERASE	Procedure	Erases the entire, or a part, of a LOB
FILECLOSE	Procedure	Closes the file
FILECLOSEALL	Procedure	Closes all previously opened files
FILEEXISTS	Function	Checks if the file exists on the server
FILEGETNAME	Procedure	Gets the directory object name and filename
FILEISOPEN	Function	Checks if the file was opened using the input BFILE locators
FILEOPEN	Procedure	Opens a BFILE
FRAGMENT_DELETE	Procedure	Deletes the data at the given offset for the given length from the LOB
FRAGMENT_INSERT	Procedure	Inserts the given data (limited to 32 KB) into the LOB at the given offset

Subprogram	Type	Description
FRAGMENT_MOVE	Procedure	Moves a chunk of bytes (BLOB) or characters (CLOB/NCLOB) from the given offset to the new offset specified
FRAGMENT_REPLACE	Procedure	Replaces the data at the given offset with the given data (not to exceed 32 KB)
FREETEMPORARY	Procedure	Frees the temporary BLOB or CLOB in the default temporary tablespace
GETCHUNKSIZE	Function	Returns the amount of space used in LOB CHUNK to store the LOB value
GETLENGTH	Function	Gets the length of the LOB value
GETOPTIONS	Function	Obtains settings corresponding to the option_types field for a particular LOB
GET_STORAGE_LIMIT	Function	Returns the storage limit for LOBs in your database configuration
INSTR	Function	Returns the matching position of the nth occurrence of the pattern in the LOB
ISOPEN	Function	Checks to see if the LOB was already opened using the input locator
ISTEMPORARY	Function	Checks if the locator is pointing to a temporary LOB
LOADBLOBFROMFILE	Procedure	Loads the BFILE data into an internal BLOB
LOADCLOBFROMFILE	Procedure	Loads the BFILE data into an internal CLOB
LOADFROMFILE	Procedure	Loads the BFILE data into an internal LOB
OPEN	Procedure	Opens a LOB (internal, external, or temporary) in the indicated mode
READ	Procedure	Reads the data from the LOB starting at the specified offset
SETOPTIONS	Procedure	Enables CSCE features on a per-LOB basis, overriding the default LOB column settings
SUBSTR	Function	Returns part of the LOB value starting at the specified offset
TRIM	Procedure	Trims the LOB value to the specified short length
WRITE	Procedure	Writes the data to a LOB from a given offset
WRITEAPPEND	Procedure	Writes a buffer at the end of a LOB

LOB usage notes

As a database administrator, you will find the following notes useful while working with LOBs in a database application:

- Object types can have the LOB type attributes.
- You can change the tablespace of a LOB segment by using the `ALTER TABLE . . . MODIFY` command.
- LOBs are stored inline along with the table row if the LOB size is less than or equal to 4000 bytes. You can specify `ENABLE | DISABLE STORAGE IN ROW` in the LOB storage clause for inline and out-of-line storage. However, the LOB locators are always stored in the LOB column of the table.
- No LOB storage specifications are applicable to the BFILE columns.

Oracle SecureFiles

Oracle SecureFiles were introduced in Oracle Database 11g Release 2 to enable optimized storage of large objects and enhanced performance. SecureFiles are re-engineered LOB storage architecture that is designed to leverage the joint capabilities of an Oracle Database and a file system. Oracle SecureFiles greatly benefitted from the file system-like features such as deduplication, compression, and encryption. In addition, the SecureFiles data works with trusted database features such as flashback, fine-grained auditing, Real Application Clusters, Automatic Storage Management, and Information Lifecycle Management. The optimization features in SecureFiles are deduplication, compression, and encryption.

With the introduction of Oracle Securefiles, the older LOB, that is, all LOBs until Oracle Database 11g Release 1, will be known as BasicFiles.

Deduplication and compression

Let us first understand what optimizes the SecureFiles storage on the disk. Oracle SecureFiles gives the best performance during the write operations due to the bulk allocation of contiguous blocks on the disk. The SecureFile segments must reside on an **Automatic Segment Space Management (ASSM)** managed tablespace.

In addition, advanced features such as deduplication and compression contribute largely to the LOB storage optimization by reducing the overall size of data getting written to the storage. The Oracle Database maintains an internal index of prefix hash and full hash for all the objects. During the write operations, the prefix hash is generated for the object, which is then matched against the prefix hashes available in the internal index. If the prefix hash is not matched, the object is written to the disk, otherwise the object hash comparison is performed. A mismatch during full hash comparison will again result in an object write. If the object hashes are matched, then a pointer (memory vector) to the master hash is written to the LOB column for the current write operation. Thus, for duplicate objects, the Oracle Database stores a single master copy. You can use `DEDUPLICATE` or `KEEP_DUPLICATES` to enable or disable the deduplicate feature.

With regular checks on the size and access, the Oracle Database can compress the SecureFiles, which maximizes their space utilization. You can specify the compression level as `COMPRESS LOW`, `COMPRESS MEDIUM` (default) or `COMPRESS HIGH` depending on the SecureFile access rate and available CPU cycles. To completely disable the compression, specify `NOCOMPRESS` in the SecureFile LOB storage specification. Note that the LOB compression is different from the table compression.

The deduplication and compression features transparently account for the enhanced performance of SecureFiles. The deduplication and compression features can be specified at the table as well as the partition level.

Encryption

A SecureFile LOB column can be safely and transparently encrypted or decrypted using the **Transparent Data Encryption (TDE)** feature of Oracle. You can use the `ENCRYPT` or `DECRYPT` clause to enable or disable the encryption feature.

If you enable all three advanced features for a SecureFile, the Oracle Database will first deduplicate, then compress, and finally encrypt it.

- Oracle SecureFile compression and deduplication are part of the Oracle Advanced Compression option
- Oracle SecureFile encryption is part of the Oracle Advanced Security option

File System Logging

You can also enable a file system-like logging for Oracle SecureFiles to reduce redo generation while loading the LOB data into a table. Typically, Oracle Database logs the data changes as well as metadata changes in the redo logs. On the other hand, a typical file system simply tracks the metadata changes. The Oracle SecureFile logging level can be set to `FILESYSTEM_LIKE_LOGGING` in order to enable the logging of only the LOB metadata. In the case of large size SecureFiles, the file system-like logging cuts short the redo generation by a huge margin and improves the data loading performance. The default logging mode for SecureFiles is `NOCACHE LOGGING`.

Write Gather Cache

Write Gather Cache (WGC) is a chunk of memory allocated within the buffer cache for staging large LOB write operations. The maximum size of the cache can be 4 **Mega Bytes (MB)**. When multiple concurrent server processes attempt to write the LOB data to the storage layer, the WGC facilitates the buffering of large write I/Os, which helps in the allocation of large adjacent space allocation. It is recommended to avoid frequent `COMMIT` operations as they will flush this cache. The SecureFile deduplication check is performed while the data is in Write Gather Cache. The usage of Write Gather Cache greatly improves the write performance of the SecureFiles.

Free space management

The free space management keeps track of the allocation and deallocation of the SecureFile LOBs in the LOB segments. A background free space monitor keeps a check on the current space usage of the SecureFile LOBs in the LOB segments. It is the background free space monitor that determines the allocation of extents.

While writing in a segment, the fixed `CHUNK` parameter value can lead to the fragmentation of the large object data. The Oracle SecureFiles use a larger and dynamic chunk size in order to support the large allocation of contiguous blocks on disk and avoid fragmentation. This feature greatly improves the write performance of SecureFiles.

Oracle SecureFiles require no indexes such as b-tree. In concurrent environments, index maintenance on the large objects can be a costly operation. SecureFiles use private metadata blocks that are contiguously located in the LOB segment.

Unlike older LOBs, SecureFiles do not suffer from high water mark contention issues because the freed space is deallocated and reclaimed simultaneously.

Oracle SecureFiles allow a LOB to be prefetched from the Oracle Database to improve read performance. The prefetch intelligence comes from the SecureFiles access patterns that increase throughput by a significant margin.

BasicFiles and SecureFiles

Starting from Oracle Database 11g, a LOB data can be stored either as a BasicFile or SecureFile. Oracle recommends that you use the SecureFile approach for storing large objects in the database because it optimizes the storage mechanism and boosts both the read and write performance. If your application uses an ASSM managed tablespace, you must create a LOB as a SecureFile.

Until Oracle 11g, the default storage type was BasicFile. However, with Oracle Database 12c, SecureFile is the default storage mechanism. The applications working with an older version of the large objects, that is, BasicFiles are advised to migrate to SecureFiles using the online redefinition approach.

The db_securefile parameter

Oracle Database 11g introduced a new initialization parameter `db_securefile` to control the behavior of LOB storage in the database. Until Oracle 11g, the default value of the parameter was `PERMITTED`. Starting from Oracle Database 12c, the default value of the `db_securefile` parameter is `PREFERRED`. The parameter can be set using the `ALTER SYSTEM` or `ALTER SESSION` statement. The parameter values are briefly described as follows:

- **PREFERRED:** This is the default parameter value in Oracle Database 12c or `COMPATIBLE` set to 12.0.0.0 and higher.
- **PERMITTED:** It allows the creation of LOBs as SecureFiles, if the LOB storage is specified as `SECUREFILE`.
- **ALWAYS:** All the LOB segments on the ASSM managed tablespaces are created as SecureFiles. Note that LOBs on a non-ASSM tablespace will still be BasicFiles.
- **FORCE:** All the LOB columns on an ASSM tablespace will be forcibly created as SecureFiles. In addition, no LOB can be created on a non-ASSM tablespace.
- **NEVER:** It restricts the creation of SecureFiles.
- **IGNORE:** It ignores the creation of SecureFiles. All the LOB columns are created as BasicFiles.

Working with LOBs

Now that we know the benefits of Oracle SecureFiles, let's illustrate how to work with the LOB data in the Oracle Database. In our case study, you will see the comparison between BasicFiles and SecureFiles.

Let's first verify the setting of the `db_securefile` initialization parameter:

```
conn sys/oracle as SYSDBA
show parameter db_securefile
```

NAME	TYPE	VALUE
db_securefil	string	PREFERRED

We will use the default setting of the parameter because it allows the creation of a SecureFile on the ASSM tablespace and the behavior can be overridden by explicitly specifying BasicFile.

For testing purposes, let's create two tablespaces with different segment space management. The `TBS_BASIC` tablespace is a manually managed tablespace while `TBS_SECURE` is an ASSM tablespace:

```
conn sys/oracle as SYSDBA

/*Create a manually managed tablespace*/
CREATE TABLESPACE tbs_basic
DATAFILE '/u01/app/oracle/oradata/orcl/tbs_basic.dbf'
SIZE 200M
SEGMENT SPACE MANAGEMENT MANUAL
/
```

Tablespace created.

```
/*Create a tablespace with ASSM*/
CREATE TABLESPACE tbs_secure
DATAFILE '/u01/app/oracle/oradata/orcl/tbs_secure.dbf'
SIZE 200M
SEGMENT SPACE MANAGEMENT AUTO
/
```

Tablespace created.

```
/*Verify the segment space management of tablespaces*/
SELECT tablespace_name, segment_space_management
FROM dba_tablespaces
```



```
WHERE tablespace_name like 'TBS%'
/
```

TABLESPACE_NAME	SEGMENT
-----	-----
TBS_BASIC	MANUAL
TBS_SECURE	AUTO

The following CREATE TABLE scripts create two tables: EMP_BASIC and EMP_SECURE. Both the tables include a BLOB column: MISC_DOCS. In the EMP_BASIC table, the MISC_DOCS column is stored as a BasicFile while in the EMP_SECURE table, the column is stored as a SecureFile. Note the BASICFILE and SECUREFILE keyword specification to differentiate the two storage approaches. The LOB segments will be created on the preceding tablespaces as well:

```
conn scott/tiger
```

```
/*Create a table EMP_BASIC*/
CREATE TABLE emp_basic
(empno, ename, deptno, job, sal, misc_docs)
LOB (misc_docs)
STORE AS BASICFILE
(tablespace TBS_BASIC)
AS
SELECT empno, ename, deptno, job, sal, empty_bLOB ()
FROM emp
/
```

```
/*Create a table EMP_SECURE*/
CREATE TABLE emp_secure
(empno, ename, deptno, job, sal, misc_docs)
LOB (misc_docs)
STORE AS SECUREFILE
(tablespace TBS_SECURE)
AS
SELECT empno, ename, deptno, job, sal, empty_bLOB ()
FROM emp
/
```

LOB metadata

The `USER_LOBS` dictionary view includes a `SECUREFILE` column to identify a column as SecureFile or BasicFile in a table. For each LOB type column in a table, Oracle creates a LOB segment separately from the table segment. Implicitly, Oracle also creates an internal index structure for the LOB column on the same tablespace specified for LOB. Note that the LOB index structure does not need to be rebuilt or maintained.

The following SQL statement queries the `USER_LOBS` dictionary view and lists the LOB segment, tablespace, in-row attribute, and SecureFile characteristic of the LOB columns:

```
conn scott/tiger

/*Query USER_LOBS for LOB metadata*/
WITH C AS
(
  SELECT table_name,
         column_name,
         segment_name,
         tablespace_name,
         in_row,
         securefile
  FROM user_LOBs
  WHERE table_name in ('EMP_BASIC','EMP_SECURE')
)
SELECT * FROM c
UNPIVOT
(column_value FOR column_name IN (table_name,column_name,segment_
name,tablespace_name,in_row,securefile))
/
```

COLUMN_NAME	COLUMN_VALUE
-----	-----
TABLE_NAME	EMP_BASIC
COLUMN_NAME	MISC_DOCS
SEGMENT_NAME	SYS_LOB0000093890C00007\$\$
TABLESPACE_NAME	TBS_BASIC
IN_ROW	YES
SECUREFILE	NO
TABLE_NAME	EMP_SECURE
COLUMN_NAME	MISC_DOCS
SEGMENT_NAME	SYS_LOB0000093893C00007\$\$
TABLESPACE_NAME	TBS_SECURE
IN_ROW	YES
SECUREFILE	YES

The following query shows the segment space allocation and initial extents and blocks allocated for BasicFile LOB and SecureFile LOB:

```
/*Query USER_SEGMENTS to query initial bytes for LOB*/
WITH C AS
(
  SELECT table_name,
         column_name,
         l.segment_name,
         s.segment_type,
         s.segment_subtype,
         to_char(s.bytes/1024) as bytes
  FROM user_LOBs l, user_segments s
  WHERE l.segment_name=s.segment_name
  AND l.table_name in ('EMP_BASIC','EMP_SECURE')
)
SELECT * FROM C
UNPIVOT
(column_value FOR column_name IN (table_name,column_name,segment_
name,segment_type, segment_subtype,bytes))
/
```

COLUMN_NAME	COLUMN_VALUE
TABLE_NAME	EMP_BASIC
COLUMN_NAME	MISC_DOCS
SEGMENT_NAME	SYS_LOB0000093890C00007\$\$
SEGMENT_TYPE	LOBSEGMENT
SEGMENT_SUBTYPE	MSSM
BYTES	64
TABLE_NAME	EMP_SECURE
COLUMN_NAME	MISC_DOCS
SEGMENT_NAME	SYS_LOB0000093893C00007\$\$
SEGMENT_TYPE	LOBSEGMENT
SEGMENT_SUBTYPE	SECUREFILE
BYTES	128

The preceding query output shows that a SecureFile needs a minimum of 16 blocks (that is, 128/8) in the first extent. At the same time, BasicFile requires a minimum of 3 blocks in the initial extent. However, the query result shows the allocation of 8 extents for the EMP_BASIC.MIS_DOCS columns.

Temporary LOB metadata can be queried from the V\$TEMPORARY_LOBS dictionary view.

Enabling the advanced features of a SecureFile

Let's check whether the advanced features, that is, compression, deduplication, and encryption for a SecureFile LOB are enabled or not:

```
/*Query USER_LOBS to view advanced features of SecureFile*/
SELECT column_name,
       encrypt,
       compression,
       deduplication
FROM user_LOBs
WHERE table_name='EMP_SECURE'
/

COLUMN_NAM ENCR COMPRE DEDUPLICATION
-----
MISC_DOCS  NO    NO      NO
```

The output from the preceding query shows that compression, deduplication, and encryption is not yet switched on for the MISC_DOCS column in the EMP_SECURE table. You can use the ALTER TABLE command to enable compression and deduplication for the SecureFile LOB column:

```
/*Modify the MISC_DOCS to enable compression and deduplication*/
ALTER TABLE emp_secure
MODIFY LOB(misc_docs)
(
  COMPRESS HIGH
  DEDUPLICATE
)
/

Table altered.
```

Let's verify the changes from the USER_LOBS table:

```
/*Query USER_LOBS to view advanced features of SecureFile*/
SELECT column_name,
       encrypt,
       compression,
       deduplication
FROM user_LOBs
WHERE table_name='EMP_SECURE'
/
```

COLUMN_NAM	ENCR	COMPRES	DEDUPLICATION
MISC_DOCS	NO	HIGH	LOB

You can also enable the LOB encryption for the MISC_DOCS table using the ALTER TABLE command. A SecureFile supports the following encryption algorithms:

- **3DES168:** This is the triple data encryption standard with a 168-bit key size
- **AES128:** This is the advanced data encryption standard with a 128-bit key size
- **AES192:** This is the default encryption algorithm. It is the advanced data encryption standard with a 192-bit key size
- **AES256:** This is the advanced data encryption standard with a 256-bit key size

The following program modifies a column to enable encryption:

```
/*Modify MISC_DOCS column to enable encryption*/
ALTER TABLE emp_secure
MODIFY
(
  misc_docs ENCRYPT USING 'AES192'
)
/
ALTER TABLE emp_secure
*
ERROR at line 1:
ORA-28365: wallet is not open
```

You might receive the preceding exception if the encryption wallet is not open. We will now authenticate the wallet by setting a password:

```
conn sys/oracle as SYSDBA
ALTER system
SET ENCRYPTION KEY
IDENTIFIED BY "orcl"
/

System altered.
```

Using the preceding password, you can authenticate and open the wallet:

```
ALTER system
SET ENCRYPTION WALLET OPEN
IDENTIFIED BY "orcl"
```

/

System altered.

Once the wallet is opened, it will create a file, `ewallet.p12`, in the default wallet directory:

```
[oracle@oradev12c wallet]$ ll
total 4
-rw-r--r--. 1 oracle oinstall 2848 May 16 06:54 ewallet.p12
```

You can also check the wallet status from the `V$ENCRYPTION_WALLET` dictionary view:

```
/*Query V$ENCRYPTION_WALLET to check the wallet status*/
SELECT wrl_type,
       wrl_parameter,
       status
FROM v$encryption_wallet
/
```

WRL_TYPE	WRL_PARAMETER	STATUS
FILE	/u01/app/oracle/admin/orcl/wallet	OPEN

Let's rerun the `ALTER TABLE` command to enable encryption for the `MISC_DOCS` column:

```
conn scott/tiger

/*Modify the MISC_DOCS to enable encryption*/
ALTER TABLE emp_secure
MODIFY
(
  misc_docs ENCRYPT USING 'AES192'
)
/

Table altered.
```

You can verify the advanced setting feature under the `USER_LOBS` dictionary view:

```
/*Query USER_LOBS to view advanced features of SecureFile*/
SELECT column_name,
       encrypt,
       compression,
       deduplication
```

```
FROM user_LOBs
WHERE table_name='EMP_SECURE'
/

COLUMN_NAM ENCR COMPRE DEDUPLICATION
-----
MISC_DOCS  YES  HIGH   LOB
```

Populating the LOB data

The following PL/SQL procedure writes an operating system file to a LOB column against a table record. It is customized for our test case to accept the directory name, file name, table name, and employee ID, for which the file has to be uploaded:

```
/*Start the PL/SQL block*/
CREATE OR REPLACE PROCEDURE upload_emp_docs
(p_dir VARCHAR2,
 p_file VARCHAR2,
 p_table VARCHAR2,
 p_empno NUMBER)
IS

/*Declaring LOB locator for the BLOB*/
  L_SOURCE_BLOB BFILE;

/*Declaring offset value for the LOB column*/
  L_AMT_BLOB  INTEGER := 4000;

/*Declaring temporary LOB columns for the LOB column*/
  L_BLOB BLOB := EMPTY_BLOB ();
  L_STMT CLOB := EMPTY_CLOB ();

BEGIN

  /*Create a BFILE locator for the external file*/
  L_SOURCE_BLOB := BFILENAME(p_dir, p_file);

  /*Create a temporary LOB*/
  DBMS_LOB.CREATETEMPORARY (l_bLOB, true);

  /*Opening the LOB locator in read only mode*/
  DBMS_LOB.OPEN(L_SOURCE_BLOB, DBMS_LOB.LOB_READONLY);

  /*Calculating the length of LOB locator*/
```

```

L_AMT_BLOB := DBMS_LOB.GETLENGTH(L_SOURCE_BLOB);

/*Load the temporary LOBs with the LOB locator*/
DBMS_LOB.LOADFROMFILE(L_BLOB, L_SOURCE_BLOB, L_AMT_BLOB);

/*Close the LOB locators*/
DBMS_LOB.CLOSE(L_SOURCE_BLOB);

/*Update the table with the temporary LOB variable*/
L_STMT := 'UPDATE '||p_table||' SET misc_docs = :p2 WHERE empno =
:p3';
EXECUTE IMMEDIATE l_stmt using l_bLOB, p_empno;

IF SQL%ROWCOUNT = 0 THEN
  DBMS_OUTPUT.PUT_LINE ('Wrong input - Employee does not exists');
ELSE
  DBMS_OUTPUT.PUT_LINE ('Document uploaded successfully for employee
'|p_empno);
END IF;

END;
/

```

Procedure created.

Let's test the procedure by uploading a PDF file for all the employees. Before we run the procedure, you must create a directory pointing to the file location and grant the read write privilege on the directory to the SCOTT user. Note that the database directory must point to a valid location on the server operating system; the reason being that the directory location is validated at the time of the directory execution only:

```

CONN sys/oracle AS SYSDBA
CREATE OR REPLACE DIRECTORY secure_dir AS '/u01/app/oracle/LOBs/'
/

```

Directory created.

```

GRANT READ, WRITE ON DIRECTORY secure_dir TO SCOTT
/

```

Grant succeeded.

The following PL/SQL block calls the `UPLOAD_EMP_DOCS` procedure to upload the PDF files in the `EMP_BASIC` and `EMP_SECURE` table for all the employees:

```
SET SERVEROUT OFF
BEGIN

    FOR I IN (SELECT rownum, empno FROM emp)
    LOOP

        /*Call the procedure to load in EMP_BASIC table*/
        upload_emp_docs('SECURE_DIR',
                        'ebook'||i.rownum||'.pdf',
                        'EMP_BASIC',
                        i.empno);

        /*Call the procedure to load in EMP_SECURE table*/
        upload_emp_docs('SECURE_DIR',
                        'ebook'||i.rownum||'.pdf',
                        'EMP_SECURE',
                        i.empno);

    END LOOP;

END;
/
```

PL/SQL procedure successfully completed.

You can check the size of the file uploaded against an employee using the `DBMS_LOB.GETLENGTH` function:

```
/*Verify the size of the BLOB in a row*/
SELECT empno, DBMS_LOB.GETLENGTH (misc_docs)
FROM emp_secure
WHERE empno = 7839
/
```

```
EMPNO DBMS_LOB.GETLENGTH(MISC_DOCS)
-----
7839  5769744
```

You can check the number of extents and blocks allocated for BasicFile and SecureFile using the following query:

```
/*Query USER_SEGMENTS to compare the total blocks consumed*/
```

```

SELECT table_name,
       l.segment_name,
       bytes/1024 KB,
       blocks,
       extents
FROM user_LOBs l, user_segments s
WHERE l.segment_name=s.segment_name
AND   l.table_name in ('EMP_BASIC','EMP_SECURE')
/

```

TABLE_NAME	SEGMENT_NAME	KB	BLOCKS	EXTENTS
EMP_BASIC	SYS_LOB0000093890C00007\$\$	81920	10240	81
EMP_SECURE	SYS_LOB0000093893C00007\$\$	7360	920	9

Temporary LOB operations

A temporary LOB enables the LOB operations such as creation and modification of the LOB column values. A temporary LOB consumes space in a temporary tablespace but it must be freed after usage. The DBMS_LOB package provides the APIs to handle temporary LOB actions.

Managing temporary LOBs

The DBMS_LOB package offers subprograms for temporary LOBs. The DBMS_LOB subprogram, ISTEMPORARY, determines whether a given LOB is temporary or not. Syntactically, the overloaded subprogram is as follows:

```

DBMS_LOB.STEMPORARY (LOB_loc IN BLOB)
RETURN INTEGER;
DBMS_LOB.STEMPORARY (LOB_loc IN CLOB CHARACTER SET ANY_CS)
RETURN INTEGER;

```

In the syntax, LOB_LOC is the LOB locator. A LOB locator can be a CLOB or BLOB type variable.

To create a temporary LOB, you use the DBMS_LOB.CREATETEMPORARY subprogram. It is an overloaded API to allow the LOB locator from a fixed or variable character set:

```

DBMS_LOB.CREATETEMPORARY
(
  LOB_loc IN OUT NOCOPY BLOB,
  cache IN BOOLEAN,
  duration IN PLS_INTEGER := DBMS_LOB.SESSION
);

```

```
DBMS_LOB.CREATETEMPORARY
(
  LOB_loc IN OUT NOCOPY CLOB CHARACTER SET ANY_CS,
  cache IN BOOLEAN,
  duration IN PLS_INTEGER := 10
);
```

In the preceding subprogram signatures:

- `LOB_loc`: This is the LOB locator.
- `cache`: This is the boolean parameter to specify whether the LOB should be cached in the buffer cache or not.
- `duration`: It specifies the life of the temporary LOB. It can be one of `SESSION`, `TRANSACTION`, or `CALL`. The default duration of a temporary LOB is `SESSION`.

`DBMS_LOB.FREETEMPORARY` frees the memory allocated for the temporary LOB. The syntax for the overloaded subprogram is as follows:

```
DBMS_LOB.FREETEMPORARY (LOB_loc IN OUT NOCOPY BLOB);
DBMS_LOB.FREETEMPORARY (LOB_loc IN OUT NOCOPY CLOB CHARACTER SET
ANY_CS);
```

Working with a temporary LOB

Let's write a PL/SQL program to illustrate the usage of the temporary LOB subprograms. We shall observe the creation, validation, and release of the temporary LOB in the program:

```
/*Enable the SERVEROUT to display the block output*/
SET SERVEROUTPUT ON

/*Start the PL/SQL block*/
DECLARE

  L_TEMP_LOB CLOB;
  AMT NUMBER;
  OFFSET NUMBER := 5;
  L_WRITE VARCHAR2(100) := 'Oracle 8i introduced LOB types';
  L_APPEND VARCHAR2(100) := 'Oracle 11g introduced SecureFiles';

BEGIN

  /*Create the temporary LOB*/
  DBMS_LOB.CREATETEMPORARY
  (
```

```

        LOB_loc => L_TEMP_LOB,
        cache => true,
        dur => dbms_LOB.session
    );

/*Verify the creation of temporary LOB*/
IF (DBMS_LOB.ISTEMPORARY(L_TEMP_LOB) = 1) THEN
    DBMS_OUTPUT.PUT_LINE('Given LOB is a temporary LOB');
ELSE
    DBMS_OUTPUT.PUT_LINE('Given LOB is a persistent LOB');
END IF;

/*Open the temporary LOB in read write mode*/
DBMS_LOB.OPEN
(
    LOB_loc => L_TEMP_LOB,
    open_mode => DBMS_LOB.LOB_READWRITE
);

/*Write the sample data in the temporary LOB*/
DBMS_LOB.WRITE
(
    LOB_loc => L_TEMP_LOB,
    amount  => LENGTH(L_WRITE),
    offset  => OFFSET,
    buffer  => L_WRITE );
DBMS_OUTPUT.PUT_LINE
(
    'Temporary LOB length after Write ' || DBMS_LOB.GETLENGTH(L_TEMP_
LOB)
);

/*Append the sample in the temporary LOB*/
DBMS_LOB.WRITEAPPEND
(
    LOB_loc => L_TEMP_LOB,
    amount  => LENGTH(L_APPEND),
    buffer  => L_APPEND
);

DBMS_OUTPUT.PUT_LINE
(
    'Temporary LOB length after Append ' || DBMS_LOB.GETLENGTH(L_TEMP_
LOB)

```

```
);

/*Display the complete content of the temporary LOB*/
DBMS_OUTPUT.PUT_LINE
(
  CHR(10)||'Temporary LOB Content: '
);

DBMS_OUTPUT.PUT_LINE
(
  DBMS_LOB.SUBSTR
  (
    L_TEMP_LOB,DBMS_LOB.GETLENGTH
    (L_TEMP_LOB), 1
  )
);

DBMS_LOB.CLOSE(LOB_loc => L_TEMP_LOB);
DBMS_LOB.FREETEMPORARY(LOB_loc => L_TEMP_LOB);

END;
/

Given LOB is a temporary LOB
Temporary LOB length after Write 34
Temporary LOB length after Append 67
Temporary LOB Content:
Oracle 8i introduced LOB typesOracle 11g introduced SecureFiles

PL/SQL procedure successfully completed.
```

Migrating LONG to LOBs

The database applications using LONG data types should migrate to LOB due to the following reasons:

- A LONG column can store a maximum of (2GB - 1) while an LOB can store a maximum data of (4GB - 1)
- A table can have one, and only one, LONG data type column while there is no restriction on the number of LOB columns in a table
- Data replication is not allowed with the LONG and LONG RAW columns



Migrating `LONG` to the `LOB` columns may generate a lot of redo. Therefore, it is advised to switch off the logging for the table containing the `LONG` column.

You can convert `LONG` to `CLOB` and `LONG RAW` to `BLOB` using either of the following listed approaches:

Use the `ALTER TABLE` command

For a table with the `LONG` type column, you can use the `ALTER TABLE` command to modify the column type to `LOB` with new storage specifications and migrate the data to a new space. Note that all the `LONG` data is migrated as `LOB` in the table. All the column-level characteristics of the `LONG` column are retained and carried forward to the `LOB` column. As a best practice, you should drop the domain indexes on the `LONG` columns, if any exist.

For example, the following command converts the `TEXT` column from `LONG` to `LOB`:

```
ALTER TABLE need_to_migrate MODIFY (text CLOB)
/
```

Using the `TO_LOB` function

You can use the `TO_LOB` utility function to convert the `LONG` data into `LOB` while performing the `CREATE TABLE AS SELECT` or `INSERT INTO...SELECT` actions. This method is beneficial when you want to perform the conversion of a subset of the `LONG` values by applying predicate clauses to the `SELECT` query. It cannot be used in a PL/SQL block.

The `TO_LOB` function converts `LONG` to `CLOB` or `NCLOB` and `LONG RAW` to `BLOB` data. Refer to the following example:

```
CREATE TABLE sales_new
AS
SELECT sales_id, sales_name, TO_LOB (sales_doc)
FROM sales_old
/
```

The `TO_LOB` function in a CTAS operation doesn't work for an Index Organized Table. However, the `INSERT AS SELECT` operation on an index-organized table still works.

Online Table Redefinition

Online table redefinition uses the `TO_LOB` utility function to perform the `LONG` to `LOB` conversion. You have to create a work-in-progress table of the same structure as the original table. Note that this time, you will use the `LOB` columns in place of the `LONG` columns. Later, you can call `DBMS_REDEFINITION.START_REDEF_TABLE` with the required parameters to kick off the conversion.

Migrating BasicFiles to SecureFiles

It is recommended to migrate the BasicFile LOBs to SecureFile LOBs using the Online Table Redefinition method. The advantage of using the Online Table Redefinition method is that none of the objects have to be offline during the process. In addition, the performance of the redefinition process can be enhanced by enabling parallelism. The following example demonstrates the online redefinition process.

A `TAB_BASICFILE` table contains the `DOC` column that is a BasicFile CLOB column:

```
SELECT *
FROM tab_basicfile
/
```

ID	DOC
1	Oracle 9i
2	Oracle 10g
3	Oracle 11g
4	Oracle 12c

The following query checks the `LOB` behavior from the `USER_LOBS` dictionary view:

```
SELECT column_name,
       securefile
FROM user_LOBs
WHERE table_name='TAB_BASICFILE'
/
```

COLUMN_NAME	SEC
DOC	NO

For redefinition, we will create a work-in-progress table with the same structure as the `TAB_BASICFILE` table. You can drop this table after the table redefinition activity is over. Note that this table should not have any indexes. During the redefinition process, the target table will carry the same data as the original table. Therefore, the host server must have sufficient free space available to perform the redefinition operation:

```

/*Creating target table TAB_SECUREFILE for redefinition*/
CREATE TABLE TAB_SECUREFILE
(id NUMBER,
 doc CLOB)
LOB(doc) STORE AS SECUREFILE
/

/*Verify the SecureFile feature*/
SELECT column_name,
       securefile
FROM user_LOBs
WHERE table_name='TAB_SECUREFILE'
/

COLUMN_NAME SEC
-----
DOC          YES

```

You can kick-start the table redefinition operation by calling the DBMS_REDEFINITION.START_REDEF_TABLE subprogram:

```

conn sys/oracle as SYSDBA
/*Start the PL/SQL block*/
DECLARE

    L_ERROR PLS_INTEGER := 0;
BEGIN

    /*Specify source and target tables for redefinition*/
    DBMS_REDEFINITION.START_REDEF_TABLE
    (uname      => 'SCOTT',          --Schema name of the tables
     orig_table => 'TAB_BASICFILE',  --Table to be redefined
     int_table  => 'TAB_SECUREFILE', --Interim table
     col_mapping => 'id id, doc doc' --Column mapping from
                                     source to interim table
    );

    /*Specify source and target tables for copying the dependents*/
    DBMS_REDEFINITION.COPY_TABLE_DEPENDENTS
    (uname      => 'SCOTT',          --Schema name of the tables
     orig_table  => 'TAB_BASICFILE', --Original table
     int_table   => 'TAB_SECUREFILE', --Interim table
     copy_indexes => 1,              --Copy indexes (1) or not (0)
     copy_triggers => true,          --Copy triggers(T) or not(F)
     copy_constraints => true,       --Copy constraints(T) or not(F)
     copy_privileges => true,        --Copy privileges(T) or not(F)
     ignore_errors => false,         --Ignore errors while copying a
                                     dependent object
    );

```



```
        num_errors      => L_ERROR, --Number of errors occurred while
                                copying dependent objects
        copy_statistics => FALSE,   --Copy statistics (1) or not (0)
        copy_mvlog      => FALSE --Copy materialized view log (1) or
                                not (0)
    );

    DBMS_OUTPUT.PUT_LINE('Errors := ' || TO_CHAR(L_ERROR));

/*Finish the redefinition process*/
    DBMS_REDEFINITION.FINISH_REDEF_TABLE
        (uname => 'SCOTT',          --Schema name of the tables
         orig_table => 'TAB_BASICFILE', --Table to be redefined
         int_table  => 'TAB_SECUREFILE' --Interim table
        );

END;
/
```

PL/SQL procedure successfully completed.

You can use `DBMS_REDEFINITION.ABORT_REDEF_TABLE` to abort the table redefinition process if you encounter the `ORA-23539: table "SCOTT"."TAB_BASICFILE" currently being redefined exception`.

Let's verify the `SECUREFILE` property for the `TAB_BASICFILE` table:

```
SELECT column_name,
       securefile
FROM USER_LOBS
WHERE TABLE_NAME='TAB_BASICFILE'
/
```

COLUMN_NAME	SEC
DOC	YES

`TAB_BASICFILE` exchanges its table properties with `TAB_SECUREFILE`:

```
SELECT column_name, securefile
FROM USER_LOBS
WHERE TABLE_NAME='TAB_SECUREFILE'
/
```

COLUMN_NAME	SEC
DOC	NO

Oracle Database 12c enhancements to SecureFiles

Oracle Database 12c adds the following features to the SecureFile LOBs:

- SecureFiles LOB will be the default storage: You might have seen this point earlier in the chapter so this is just reiterating the fact that SecureFiles will be the default storage option in Oracle Database 12c. The value of the `db_securefile` initialization parameter is `PREFERRED`, which means that all LOBs on the ASSM enabled tablespace will be created as SecureFiles (unless BasicFile is explicitly specified or the tablespace is non ASSM).
- Parallel DML support for LOBs: If all the LOB columns in a partitioned or non-partitioned table are SecureFiles, then DML operations can be executed in parallel.
- Data Pump uses SecureFile as the default LOB storage: You can now migrate a BasicFile to SecureFile using the Data Pump utility. During an export data pump operation, you can set the `TRANSFORM` handle as `LOB_STORAGE` to decide how the LOB segments will be created in the target database.
- The `LOB_STORAGE` handle can be `SECUREFILE`, `BASICFILE`, `DEFAULT`, or `NO_CHANGE` (default): The `NO_CHANGE` option does not manipulate the LOB storage. The `DEFAULT` option follows the default LOB storage on the target database. The `SECUREFILE` | `BASICFILE` options enable the creation of the LOB segments as SecureFiles or BasicFiles.

Summary

In this chapter, you learned how to handle large objects in Oracle. You now understand the dynamics of storing large objects in a database. A detailed discussion on the optimization features of Oracle SecureFiles would have helped you to differentiate the two storage approaches. The comparative illustration of a BasicFile and SecureFile will help in deducing the appropriate conclusions.

Moving forward, the next chapter will deal with another important aspect of database programming, that is, tuning the PL/SQL code. Tuning—an indispensable part of programming—is a skill for writing optimized code, which comes from knowledge and matures with experience.

Practice exercise

- Internal LOBs can be used as attributes of a user-defined data type:
 1. True.
 2. False.
- Internal LOBs cannot be passed as parameters to PL/SQL subprograms:
 1. True.
 2. False.
- Internal LOBs can be stored in a tablespace that is different from the tablespace that stores the table containing the LOB column:
 1. True.
 2. False.

- You issue the following command to create a table called LOB_STORE:

```
CREATE TABLE LOB_store
(LOB_id NUMBER(3),
 photo BLOB DEFAULT EMPTY_CLOB(),
 cv CLOB DEFAULT NULL,
 ext_file BFILE DEFAULT NULL)
/
```

Identify the issue in this script:

1. The table is created successfully.
 2. It generates an error because a BLOB column cannot be initialized with `EMPTY_CLOB()`.
 3. It generates an error because `DEFAULT` cannot be set to `NULL` for a `CLOB` column during table creation.
 4. It generates an error because `DEFAULT` cannot be set to `NULL` for a `BFILE` column during table creation.
- Identify the correct statements relating to the initialization of LOBs:
 1. An internal LOB cannot be initialized in the `CREATE TABLE` statement.
 2. A `BFILE` column can be initialized with the `EMPTY_BFILE()` constructor.
 3. The `EMPTY_CLOB()` and `EMPTY_BLOB()` functions can be used to initialize both the `NULL` and `NOT NULL` internal LOBs of the `CLOB` and `BLOB` types.
 4. Initialization is a mandatory step for LOB type columns.

- Which two statements are true about the `FILEOPEN` subprogram in the `DBMS_LOB` package?
 1. `FILEOPEN` can be used to open only internal LOBs.
 2. `FILEOPEN` can be used to open only external LOBs.
 3. `FILEOPEN` cannot be used to open temporary LOBs.
 4. `FILEOPEN` can be used to open internal and external LOBs.
- Temporary LOBs can be shared among the users who are currently connected to the server:
 1. True.
 2. False.
- Identify the correct statements about BFILES:
 1. A `BFILE` column in a table must be initialized with a dummy locator.
 2. BFILES cannot be used as attributes in an object type.
 3. The `BFILE` data type is a read-only data type.
 4. The external file still persists if the `BFILE` locator is deleted or modified.
- Pick the incorrect statements about Temporary LOBs:
 1. It resides in the user's temporary tablespace.
 2. It can be used during the `LONG` to `LOB` data type migration.
 3. It can be persistent for `SESSION`, `TRANSACTION`, or `CALL`.
 4. Temporary `LOB` of a `BFILE` type can be created.

- A `SAMPLE_DATA` table has the following structure:

Name	Null?	Type
SD_ID		NUMBER
SD_SOURCE		BFILE

You update a row in the table using the `UPDATE` statement, as follows:

```
UPDATE sample_data
SET sd_source = BFILENAME('SD_FILE', 'sample.pdf')
WHERE sd_id = 448;
```

However, you receive the error—ORA-22286: insufficient privileges—on the file or directory to perform `FILEOPEN`.

What could be the probable cause of this error?

1. The directory `SD_FILE` does not exist.
 2. The file `sample.pdf` does not exist.
 3. The user does not have the `READ` privilege on the directory.
 4. The file `sample.pdf` is a read-only file.
- Which of the following statements about SecureFiles are true?
 1. SecureFiles require an ASSM-enabled tablespace.
 2. A `BFILE` type column in a table can be declared as SecureFiles.
 3. A SecureFile is not affected by a `LOB` index contention.
 4. SecureFiles use a new cache component of the buffer cache to hold the `LOB` data.
 - Identify the incorrect statement about the compression feature in SecureFiles:
 1. Compression impacts performance during a `LOB` write operation.
 2. SecureFiles compression is part of the advanced compression feature in Oracle.
 3. Possible degrees of compression can be `MEDIUM` and `HIGH`.
 4. Oracle compresses all the `LOB` data at high priority if the feature has been enabled for a SecureFile.
 - The compression feature can be enabled only for encrypted SecureFiles:
 1. True.
 2. False.
 - A compressed table containing a `SecureFile` column will automatically enable compression for SecureFiles:
 1. True.
 2. False.
 - Identify the true statements about the deduplication feature of SecureFiles:
 1. `KEEP_DUPLICATES` is the default option.
 2. `DEDUPLICATE` retains one copy of the duplicate `LOB` data.
 3. The deduplication feature impacts write performance as Oracle compares the secure hash code with the available hash codes before writing to the disk.
 4. The deduplication of SecureFiles is checked on the basis of the filenames.

- Pick the correct statement for the encryption feature in `SecureFiles`:
 1. The `SecureFile` encryption keys are stored in the table.
 2. The `SecureFile` encryption keys are stored in the database.
 3. The `SecureFile` encryption keys are stored outside the database.
 4. Encryption algorithms cannot be modified for an encrypted `SecureFile` column.
- Which of the following statements are true for `BasicFile` to `SecureFile` migration in Oracle?
 1. The `BasicFile` to `SecureFile` migration can be done through a data pump operation.
 2. Table redefinition is preferred as it does the migration with all the resources connected online.
 3. The `DBMS_REDEFINITION` package can migrate only one `LOB` column at a time.
 4. Unnecessary space consumption makes the redefinition process less preferable over the partition method.

8

Tuning the PL/SQL Code

Database tuning refers to an exercise that aims to improve the performance of a database. The art of optimizing the database performance largely depends on how well one understands the database architecture, application design and server environment. The areas that can be potentially tuned are the application design, network, SQL queries, and PL/SQL code. This exercise starts with the detection and identification of a problem, followed by analysis and tuning recommendations. In this chapter, we will see tuning practices related to the Oracle PL/SQL code.

Programs written in Oracle PL/SQL can be tuned to optimize data-centric and CPU-intensive operations. In a PL/SQL program unit, statements like loops, dynamic SQL statements, routine calls, and the compilation mode can be tuned for better performance. This chapter will discuss the different code compilation techniques and their benefits. The chapter outline is as follows:

- The PL/SQL Compiler
 - Subprogram inline
- Native compilation of the PL/SQL Code
- Tuning the PL/SQL Code
 - Build secure applications using bind variables
 - Call parameters by reference
 - Avoid an implicit data type conversion
 - Understand the `NOT NULL` constraint
 - Select an appropriate numeric data type
 - Bulk processing in PL/SQL

The PL/SQL Compiler

The PL/SQL compiler converts a piece of PL/SQL code that is written in user-readable language semantics to system code. At a higher level, the code compilation is a three-step process that includes code parsing, parameter binding, and translation into **machine code (M-code)**. The compiler raises a syntax or compilation error if a PL/SQL construct or statement is incorrectly used. Once the PL/SQL program code is error-free and argument binding is done, the system code is generated and stored in the database.

Until Oracle 10g Release 1, the M-code was generated without any code optimization. Starting from Oracle Database 10g Release 2, M-code uses the PL/SQL optimizer to apply code formatting for better performance. The PL/SQL optimizer level is governed by a compilation parameter known as `PLSQL_OPTIMIZE_LEVEL`. The parameter value can be set at the system or session level as a number between 1 to 3. By default, the parameter value is 2.

The PL/SQL optimizer uses multiple techniques to optimize the PL/SQL code, for example, the removal of dead code from the program unit, or inlining the calls to subprograms. Let us first see how subprogram inlining works.

Subprogram inlining in PL/SQL

A PL/SQL program unit with a locally-declared and invoked subprogram can be optimized using the inlining method. While optimizing the program code, the PL/SQL optimizer replaces the subprogram invocation calls by the subprogram body itself. For the PL/SQL compiler to inline a local subprogram, the following criteria must be met:

- Set `PLSQL_OPTIMIZE_LEVEL` to 2 and use `PRAGMA INLINE`: The compiler inlines only those subprograms that are specified with `PRAGMA INLINE`.
- Set `PLSQL_OPTIMIZE_LEVEL` to 3: The compiler inlines all the subprogram calls. However, you can specify `PRAGMA INLINE` to prevent the inlining of a particular subprogram. To disable the inlining of a program unit, specify `PRAGMA INLINE ([subprogram unit], 'NO')`.

`PRAGMA INLINE` directs the compiler to inline the subprogram calls that are just succeeding it. For example, in the following PL/SQL block, the subprogram call is inlined in Line 5 but not in Line 7:


```
PROCEDURE P_SUM_NUM (P_A NUMBER, P_B NUMBER)      /* Line 1 */
...                                                /* Line 2 */
BEGIN                                              /* Line 3 */
PRAGMA INLINE ('P_SUM_NUM', 'YES')                /* Line 4 */
result_1 := P_SUM_NUM (10, 20);                    /* Line 5 */
```

```

.....                                /* Line 6 */
result_2 := P_SUM_NUM (100, 200);    /* Line 7 */
END;                                  /* Line 8 */
/                                    /* Line 9 */

```

At most of the scenarios, subprogram inlining improves the performance of a PL/SQL program. If the subprogram is a large unit, the cost of inlining would be more than the modular execution.


 **PRAGMA INLINE** ([subprogram], 'NO') will always override **PRAGMA INLINE** ([subprogram], 'YES') in the same declaration.

It is recommended that you inline those utility subprograms that are frequently invoked in a PL/SQL block. The intra-unit inlining is traceable through session-level warnings. The session-level warnings can be turned on by setting the `PLSQL_WARNINGS` parameter to `ENABLE:ALL`.

PRAGMA INLINE

PRAGMA is a compiler directive that hints the compiler. Any error in the usage of **PRAGMA** results in a compilation error. Note that a **PRAGMA**, which accepts an argument, cannot accept a formal argument but always needs an actual argument.

PRAGMA INLINE was introduced in Oracle Database 11g to support subprogram inlining for better PL/SQL performance. When the compilation parameter `PLSQL_OPTIMIZE_LEVEL` is 2, you have to specify the subprogram to be inlined using **PRAGMA INLINE**. When `PLSQL_OPTIMIZE_LEVEL` is 3, the PL/SQL optimizer tries to inline most of the subprogram calls, without requiring any **PRAGMA** specification. At this stage, if you perceive an inlined subprogram to be irrelevant or undesirable, you can disable the inlining of that particular subprogram through **PRAGMA INLINE**.

 **PRAGMA INLINE** impacts only when `PLSQL_OPTIMIZE_LEVEL` is either 2 or 3. When `PLSQL_OPTIMIZE_LEVEL` is 1, it has no effect.

PLSQL_OPTIMIZE_LEVEL

Oracle Database 10g introduced the `PLSQL_OPTIMIZE_LEVEL` initialization parameter to enable or disable PL/SQL optimization. If enabled, the optimizer deploys several optimization techniques in accordance with the level. The compilation settings of a PL/SQL program can be queried from the `USER_PLSQL_OBJECT_SETTINGS` dictionary view.

Until Oracle Database 10g, the parameter value could be either 0, 1, or 2. Starting from Oracle Database 11g, the new optimization level can be enabled by setting the parameter value to 3. Note that the default value of the parameter is 2. The parameter value can be modified using the `ALTER SYSTEM` or `ALTER SESSION` statements. Only the PL/SQL programs that are compiled after the parameter modification are impacted. You can also specify the `PLSQL_OPTIMIZE_LEVEL` value at the time of explicit compilation of a program unit. For example, a `P_OPT_LVL` procedure can be recompiled using the following statement:

```
ALTER PROCEDURE p_opt_lvl COMPILE PLSQL_OPTIMIZE_LEVEL=1
/
```

The following PL/SQL procedure demonstrates the subprogram inlining using `PRAGMA INLINE`. The procedure adds the series $(1^2) + (2^2) + (3^2) + \dots + (N^2)$ up to N terms:

```
/*Create a procedure*/
CREATE OR REPLACE PROCEDURE P_SUM_SERIES(p_count NUMBER)
IS

    l_series NUMBER := 0;
    l_time NUMBER;

/*Declare a local subprogram which returns the double of a number*/
    FUNCTION F_CROSS (p_num NUMBER, p_multiplier NUMBER) RETURN NUMBER
    IS
        l_result NUMBER;

    BEGIN
        l_result := p_num * p_multiplier;
        RETURN (l_result);
    END F_CROSS;

    BEGIN

        /*Capture the start time*/
        l_time := DBMS_UTILITY.GET_TIME();
```

```

/*Begin the loop for series calculation*/
FOR J IN 1..p_count
LOOP

    /*Set inlining for the local subprogram*/
    PRAGMA INLINE (F_CROSS, 'YES');
    l_series := l_series + F_CROSS(J,2);
END LOOP;

/*Time consumed to calculate the result*/
DBMS_OUTPUT.PUT_LINE('Execution time: ' || TO_CHAR(DBMS_UTILITY.GET_
TIME() - L_TIME));

END;
/

```

Case 1: When PLSQL_OPTIMIZE_LEVEL = 0

At level 0, the PL/SQL optimizer doesn't kick in, so no code optimization is enabled. The compiler maintains the code evaluation order, and performs object checks.

Enable PLSQL_WARNINGS to capture the inline operations:

```

ALTER SESSION SET plsql_warnings = 'enable:all'
/

```

Session altered.

Let's recompile the P_SUM_SERIES procedure with the PLSQL_OPTIMIZE_LEVEL=0 setting:

```

ALTER PROCEDURE P_SUM_SERIES COMPILE PLSQL_OPTIMIZE_LEVEL=0
/

SP2-0804: Procedure created with compilation warnings

show errors
Errors for PROCEDURE P_SUM_SERIES:

LINE/COL ERROR
-----
1/1      PLW-05018: unit P_SUM_SERIES omitted optional AUTHID clause;
         default value DEFINER used

```

Let's execute the procedure with a relatively large input value for a computation-intensive operation:

```
BEGIN
    p_sum_series (10000000);
END;
/
Execution time:776

PL/SQL procedure successfully completed.
```

Case 2: When PLSQL_OPTIMIZE_LEVEL = 1

At level 1, the PL/SQL optimizer applies conventional optimization techniques to a PL/SQL program unit. It eliminates the dead code, and skips the redundant and unnecessary code in the program while generating the p-code instructions.

What is **dead code**? If a PL/SQL statement remains unchanged within an iterative or conditional construct, and doesn't contribute to the program logic, it is known as dead code. While translating the program into the system code instructions, the PL/SQL optimizer identifies the pieces of dead code and prevents their conversion.

Let's recompile the P_SUM_SERIES procedure using the ALTER PROCEDURE statement for PLSQL_OPTIMIZE_LEVEL=1 and check the warnings:

```
ALTER PROCEDURE P_SUM_SERIES COMPILE PLSQL_OPTIMIZE_LEVEL=1
/

SP2-0804: Procedure created with compilation warnings

show errors
Errors for PROCEDURE P_SUM_SERIES:

LINE/COL ERROR
-----
1/1      PLW-05018: unit P_SUM_SERIES omitted optional AUTHID clause;
         default value DEFINER used
```

You can also query the compilation warnings from the USER_ERRORS dictionary view.

Although PRAGMA was specified, the F_CROSS invocation was not inlined. Therefore, the subprogram inlining doesn't seem to work when PLSQL_OPTIMIZE_LEVEL is set to 1:

```
BEGIN
    P_SUM_SERIES (10000000);
END;
```

```

/
Execution time:710
PL/SQL procedure successfully completed.

```

Case 3: When PLSQL_OPTIMIZE_LEVEL = 2

At level 2, the PL/SQL optimizer performs the intelligent code optimization through techniques like explicit subprogram inlining and code reorganization.

Recompile the P_SUM_SERIES procedure with PLSQL_OPTIMIZE_LEVEL=2 using the ALTER PROCEDURE statement:

```

ALTER PROCEDURE p_sum_series COMPILE PLSQL_OPTIMIZE_LEVEL=2
/

SP2-0805: Procedure altered with compilation warnings

```

Let's retrieve the compilation warnings. Oracle includes a new set of compilation warnings to demonstrate the inlining flow in the program. The PL/SQL optimizer is instructed by PRAGMA to inline the subprogram in line 28. At line 8, the function was removed and its body was merged into the main program. Line 28 shows the inlining request and action:

```

SQL> show errors
Errors for PROCEDURE P_SUM_SERIES:

LINE/COL  ERROR
-----
1/1        PLW-05018: unit P_SUM_SERIES omitted optional AUTHID clause;
          default value DEFINER used
8/3        PLW-06006: uncalled procedure "F_CROSS" is removed.
28/7       PLW-06004: inlining of call of procedure 'F_CROSS' requested
28/7       PLW-06005: inlining of call of procedure 'F_CROSS' was done

```

Let's check how a level 2-optimized and compiled program performs:

```

BEGIN
  P_SUM_SERIES (10000000);
END;
/
Execution time:456
PL/SQL procedure successfully completed.

```

Well done! It runs in 456ms, which is almost half of its baseline, that is, 2 times better performance.

Case 4: When PLSQL_OPTIMIZE_LEVEL = 3

Level 3 optimization focuses on the predictive and automatic inlining and code refactoring. Modify the compilation parameter in the session using the following statement:

```
ALTER SESSION SET plsql_optimize_level=3
/
```

Session altered.

To see the implicit inlining of the subprogram, let's recreate the P_SUM_SERIES procedure without the PRAGMA specification:

```
/*Create a procedure*/
CREATE OR REPLACE PROCEDURE P_SUM_SERIES(p_count NUMBER)
IS

    l_series NUMBER := 0;
    l_time NUMBER;

/*Declare a local subprogram to return the double of a number*/
    FUNCTION F_CROSS (p_num NUMBER) RETURN NUMBER IS
    BEGIN
        RETURN (p_num * 2);
    END F_CROSS;

BEGIN

    /*Capture the start time*/
    l_time := DBMS_UTILITY.GET_TIME();

    /*Begin the loop for series calculation*/
    FOR J IN 1..p_count
    LOOP

        /*Set inlining for the local subprogram*/
        l_series := l_series + F_CROSS (J);
    END LOOP;

    /*Time consumed to calculate the result*/
    DBMS_OUTPUT.PUT_LINE('Execution time: ' || TO_CHAR(DBMS_UTILITY.GET_
TIME() - L_TIME));
END;
/

SP2-0805: Procedure altered with compilation warnings
```

Let's check the compilation warnings:

```
show errors
Errors for PROCEDURE P_SUM_SERIES:

LINE/COL ERROR
-----
1/1          PLW-05018: unit P_SUM_SERIES omitted optional AUTHID clause;
              default value DEFINER used
8/3          PLW-06006: uncalled procedure "F_CROSS" is removed.
25/7         PLW-06005: inlining of call of procedure 'F_CROSS' was done
```

Note the warnings at line 8 and line 24. No `PRAGMA` specified, but the PL/SQL optimizer inlines the subprogram. Let's see how the procedure execution is impacted by level 3 optimization:

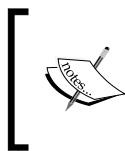
```
BEGIN
  P_SUM_SERIES (10000000);
END;
/

Execution time:420
PL/SQL procedure successfully completed.
```

The execution time is almost equal to the level 2 optimization, because the technique to optimize the code was similar in both the cases.

The consolidation of the execution numbers is shown in the following table:

PLSQL_OPTIMIZE_LEVEL	Inlining Requested	Inlining Done	Execution time	Performance factor
PLSQL_OPTIMIZE_LEVEL = 0	Yes	No	776	1
PLSQL_OPTIMIZE_LEVEL = 1	Yes	No	710	1.1x
PLSQL_OPTIMIZE_LEVEL = 2	Yes	Yes	456	1.7x
PLSQL_OPTIMIZE_LEVEL = 3	No	Yes	420	1.8x



It is recommended that you have `PLSQL_OPTIMIZE_LEVEL` as 2 for the database development and production environments. It ideally optimizes the PL/SQL code and enables the database developers to control the subprogram inlining.

Native and interpreted compilation techniques

Until Oracle 9i, all PL/SQL program units were compiled in interpreted mode. Starting with Oracle 9i, PL/SQL programs can be compiled either in interpreted mode or native mode. Let's quickly understand what are interpreted and native compilation modes.

The PL/SQL compiler generates machine code for the program units. Machine code is a set of instructions stored in the database dictionaries that runs against the **PL/SQL virtual machine (PVM)**. At the time of the program invocation, the M-code is scanned by one of the subroutines in the PVM. The scanning process involves the identification of the operation code and operands and routing the call to the appropriate subroutine. The scanning of the machine code instructions consumes systems resources, which may impact the runtime performance of the code. This is how interpreted compilation works. In the case of native compilation, a shareable **dynamic linked library (DLL)** is generated instead of a machine code. At runtime, the DLL is directly invoked and it invokes the subroutine along with the required set of arguments. With DLL, you don't have to go through the scanning procedure at runtime, which contributes to better code performance.

Until Oracle Database 9i and 10g, the challenge with native compilation was to generate a platform-specific shareable and nonportable library. It was made possible by the C compiler and linker commands located in `$ORACLE_HOME/plsql/spnc_` commands, which would compile and link the C translation of the machine code to a platform-specific library. In Oracle Database 9i, the libraries were stored on a file system, while in Oracle Database 10g, the libraries were stored in a catalog (database dictionary). A copy of the shared libraries was stored on a file system for backup and operating-system operations. The file system location used to be specified in `PLSQL_NATIVE_LIBRARY_DIR` and `PLSQL_NATIVE_LIBRARY_SUBDIR_COUNT`. The shared libraries were automatically extracted, restored, and deleted from the file system. If any of the file system libraries were lost or dropped accidentally, they were re-extracted and restored in the file system after the database was restarted.



The shared libraries generated during native compilation are stored in the `NCOMP_DLL$` dictionary.

Native compilation is supported with the **Real Application Cluster (RAC)** database option.

This approach worked well until the production DBAs showed reluctance in having the C compiler on production environments and procuring an additional C compiler license.

Oracle Database 11g Real Native Compilation

Oracle Database 11g introduced the real native compilation technique to address the below:

- Remove the C compiler and linker dependencies to generate platform specific libraries
- Improve code compilation performance
- No compromise with the runtime performance

Starting from Oracle Database 11g, the Oracle Database can generate platform-specific machine code from PL/SQL code and store it in the database catalog or an internal dictionary without the intervention of a third party C compiler. The native code is then loaded directly from the catalog without staging it on a filesystem. Oracle Database 11g real native compilation improves the compilation performance by a degree of magnitude.

The following list summarizes the salient features of real native compilation process:

- There is no dependency on a third party C Compiler.
- The native code is stored in the `SYSTEM` tablespace.
- There is no DBA task to set the filesystem initialization parameter.
- The compilation method is controlled by an initialization parameter: `PLSQL_CODE_TYPE [native/interpreted]`.
- Native compilation is supported for all types of PL/SQL program units.
- In `INTERPRETED` mode, the PL/SQL code is compiled to the equivalent machine code. The machine code instructions are scanned during the runtime and routed to the appropriate subroutine on the PL/SQL Virtual Machine.
- In `NATIVE` mode, the PL/SQL code is compiled to machine code, which is directly translated to the platform-specific. At runtime, the machine code is executed natively by the PL/SQL Virtual Machine.
- The real native compilation mode can be set at system level, session level, and object level. A natively compiled program unit can call an interpreted program and vice versa.

Selecting the appropriate compilation mode

During the database development phase, the PL/SQL program units tend to get recompiled frequently. At this stage, you would want to have the faster compilation of programs. Therefore, you can choose to compile the program units in interpreted mode.

Once the database is deployed for production, and has fewer chances of frequent recompilation, you can choose native compilation. PL/SQL native compilation enhances the runtime performance of the program units. Computation-intensive programs are expected to benefit most from the native compilation method. The performance gains will be less if the program is frequently switching the context between the SQL and PL/SQL engines.

Oracle allows having a mix of program units with different compilation modes. However, keep in mind that if a natively compiled unit invokes an interpreted unit, the program execution performance will be impacted.

Setting the compilation mode

The compilation method is set using the `PLSQL_CODE_TYPE` parameter. The admissible values for the parameter are `INTERPRETED` and `NATIVE`. The default value of the parameter is `INTERPRETED`. The parameter can be set using the `ALTER SYSTEM` or `ALTER SESSION` statement.

At the `SYSTEM` level:

```
ALTER SYSTEM SET PLSQL_CODE_TYPE = [NATIVE | INTERPRETED]
```

At the `SESSION` level:

```
ALTER SESSION SET PLSQL_CODE_TYPE = [NATIVE | INTERPRETED]
```

Alternatively, you can also compile a particular PL/SQL program unit in the native or interpreted method by recompiling an object using the `ALTER <object type>` statement. For example, the following script will recompile a procedure with a new value for `PLSQL_CODE_TYPE`, but reusing the existing compilation parameters:

```
ALTER PROCEDURE <procedure name> COMPILE PLSQL_CODE_TYPE=NATIVE REUSE
SETTINGS
/
```

The PL/SQL program units retain their compilation mode setting unless they are recompiled in a different compilation mode.

Querying the compilation settings

The compilation mode of an object can be queried from the data dictionary view: [USER | DBA | ALL]_PLSQL_OBJECT_SETTINGS. This dictionary view contains the compilation settings for standalone program units, package specification, package body, and packaged subprograms:

```
/*Describe the structure of USER_PLSQL_OBJECT_SETTINGS*/
SQL> DESC USER_PLSQL_OBJECT_SETTINGS
Name                               Null?      Type
-----
NAME                               NOT NULL   VARCHAR2(30)
TYPE                               VARCHAR2(12)
PLSQL_OPTIMIZE_LEVEL               NUMBER
PLSQL_CODE_TYPE                    VARCHAR2(4000)
PLSQL_DEBUG                        VARCHAR2(4000)
PLSQL_WARNINGS                     VARCHAR2(4000)
NLS_LENGTH_SEMANTICS               VARCHAR2(4000)
PLSQL_CCFLAGS                      VARCHAR2(4000)
PLSCOPE_SETTINGS                   VARCHAR2(4000)
ORIGIN_CON_ID                      NUMBER
```

Compiling a program unit for native or interpreted compilation

In this section, let's check how a PL/SQL program unit can be compiled in different compilation modes at the object level.

1. Show the compilation mode of the session:

```
/*Connect as sysdba*/
connect sys/oracle as sysdba

/*Display current setting of parameter PLSQL_CODE_TYPE*/
SELECT name,
       value
FROM v$parameter
WHERE name = 'plsql_code_type'
/

NAME                               VALUE
-----
plsql_code_type                     INTERPRETED
```

2. Create a standalone function, which currently gets compiled in the INTERPRETED mode:

```
/*Create a function*/
CREATE OR REPLACE FUNCTION f_get_caps (p_name VARCHAR2)
RETURN VARCHAR2
IS
BEGIN
    RETURN UPPER (p_name);
END;
/
```

Function created.

3. Verify the compilation mode of the F_GET_CAPS function:

```
/*Query the compilation settings for the function F_GET_CAPS*/
SELECT name,
       type,
       plsql_code_type,
       plsql_optimize_level OPTIMIZE
FROM USER_PLSQL_OBJECT_SETTINGS
WHERE name = 'F_GET_CAPS'
/
```

NAME	TYPE	PLSQL_CODE_TYPE	OPTIMIZE

F_GET_CAPS	FUNCTION	INTERPRETED	2

4. Recompile the function using native mode:

```
/*Explicitly compile the function*/
ALTER FUNCTION f_get_caps COMPILE PLSQL_CODE_TYPE=NATIVE
/
```

Function altered.

5. Confirm the change in the compilation mode of the F_GET_CAPS function:

```
/*Query the compilation settings for the function F_GET_CAPS*/
SELECT name,
       type,
       plsql_code_type,
       plsql_optimize_level OPTIMIZE
FROM USER_PLSQL_OBJECT_SETTINGS
WHERE name = 'F_GET_CAPS'
/
```

NAME	TYPE	PLSQL_CODE_TYPE	OPTIMIZE

F_GET_CAPS	FUNCTION	NATIVE	2

6. You can also verify the generation of a shared library for the `F_GET_CAPS` function by querying the `NCOMP_DLL$` dictionary view. The library is stored as a BLOB in the `NCOMP_DLL$` view:

```
connect sys/system as sysdba
SELECT object_name,
       dllname
FROM ncomp_dll$, dba_objects
WHERE obj#=object_id
AND owner='SCOTT'
/
```

OBJECT_NAME	DLLNAME
F_GET_CAPS	465F4745545F434150535F5F53434F54545F5F465F5F3934303837

The library is automatically deleted from the dictionary if the object is recompiled with a different compilation mode.

Recompiling a database for a PL/SQL native or interpreted compilation

You can compile all the PL/SQL program units in a database from interpreted to native compilation mode and vice versa. Oracle enables the bulk compilation of the PL/SQL program units through inbuilt scripts: `dbmsupnv.sql` and `dbmsupgin.sql`. The `dbmsupgnv.sql` script compiles all the program units using native compilation mode while `dbmsupgin.sql` compiles all the program units in interpreted mode. The following points should be considered during the planning stage:

- Choose the right compilation mode—the selection of the appropriate compilation mode is the key to ensure better code performance.
- SYSDBA or a user with DBA privileges can run the recompilation procedure. If the user is protected through the Oracle Database Vault option, the user must be granted the `DV_PATCH_ADMIN` role.
- Object type specifications cannot be compiled in native compilation mode.
- Skip the PL/SQL package specifications without an executable section from native compilation.
- You cannot exclude any PL/SQL program unit while compiling an entire database for interpreted compilation.

The following steps show how to compile a database for native or interpreted compilation mode:

1. Shutdown the database – you must shutdown the Oracle Database and the TNS listener. Ensure that all connections to the database from the application tier are terminated.
2. Set `PLSQL_CODE_TYPE` as `NATIVE` or `INTERPRETED` in the parameter file. If you are using a server parameter file (spfile), you can modify the parameter value before a database is shutdown, or after the database is started. The `dbmsupgnv.sql` script also sets `PLSQL_CODE_TYPE` before compiling the PL/SQL program units:

```
/*Alter the system to set the new compilation mode*/
SQL> ALTER SYSTEM SET PLSQL_CODE_TYPE=NATIVE SCOPE=SPFILE
/
```

System altered.

3. Verify and set `PLSQL_OPTIMIZE_LEVEL` as 2 (or higher).
4. Start the database in the upgrade mode:

```
SQL> connect sys/oracle as sysdba
```

Connected to an idle instance.

```
/*Startup in upgrade mode*/
SQL> startup upgrade
ORACLE instance started.
```

```
...
...
```

Database mounted.

Database opened.

If you are working in an Oracle Database 12c Multitenant architecture, you have to open the pluggable database in the `UPGRADE` and `RESTRICTED` mode:

```
ALTER PLUGGABLE DATABASE <pdb name> OPEN UPGRADE RESTRICTED
/
```

5. Connected as the `SYS` user, execute the `dbmsupgnv.sql` script:

```
/*Execute the recompilation script*/
SQL> @ORACLE_HOME/rdbms/admin/dbmsupgnv.sql
...
...
#####
####
```

```
#####
#####
dbmsupgnv.sql completed successfully. All PL/SQL procedures,
functions, type bodies, triggers, and type bodies objects in
the
database have been invalidated and their settings set to
native.
```

```
Shut down and restart the database in normal mode and
run utlrlp.sql to recompile invalid objects.
```

```
#####
#####
#####
#####
```

The `dbmsupgnv.sql` and `dbmsupgin.sql` scripts create a PL/SQL package `sys.dbmsncdb` with two subprograms: `SETUP_FOR_NATIVE_COMPILE` and `SETUP_FOR_INTERPRETED_COMPILE`. The `dbmsupgnv.sql` involves `SETUP_FOR_NATIVE_COMPILE` with a user supplied input. The input should be `TRUE` to compile all functions, procedures, package bodies, triggers, and type bodies. If the input is `FALSE`, package and object type specifications are also included in the process. If the PL/SQL package specifications contain an executable section, you should provide the input as `FALSE`.


The script modifies the compilation of all the PL/SQL programs to native but invalidates them at the same time.

6. Shut down and restart the database in the NORMAL mode:

```
/*Shutdown and startup the database*/
SQL> shutdown immediate
Database closed.
Database dismounted.
ORACLE instance shut down.
```

```
SQL> startup
ORACLE instance started.
...
...
Database mounted.
Database opened.
```


7. Execute the `utlrp.sql` script to recompile the invalidated objects. Although the objects that have been invalidated get automatically recompiled whenever they are subsequently invoked, it is recommended that you compile and revalidate the objects to reduce recompilation latencies.

 It is recommended that you run `utlrp.sql` in a restricted session. Any connection request from the application services or a database session may lead to deadlocks.

```
SQL> ALTER SYSTEM ENABLE RESTRICTED SESSION
/
```

System altered.

```
SQL> @ORACLE_HOME\rdbms\admin\utlrp.sql
```

The script recompiles all the program units that have a default compilation mode. You can rerun the script any number of times to recompile the invalidated objects. The recompilation operation can be optimized through parallelization wherein the degree of parallelism is selected based on `CPU_COUNT` and `PARALLEL_THREADS_PER_CPU`.


For troubleshooting and diagnosis, you can use the following queries:

- Query the invalid objects:

```
SELECT o.OWNER,
       o.OBJECT_NAME,
       o.OBJECT_TYPE
FROM DBA_OBJECTS o, DBA_PLSQL_OBJECT_SETTINGS s
WHERE o.OBJECT_NAME = s.NAME
AND o.STATUS = 'INVALID'
/
```

- Query the count of the PL/SQL programs in each compilation mode:

```
SELECT TYPE,
       PLSQL_CODE_TYPE,
       COUNT(*)
FROM DBA_PLSQL_OBJECT_SETTINGS
WHERE PLSQL_CODE_TYPE IS NOT NULL
GROUP BY TYPE, PLSQL_CODE_TYPE
ORDER BY TYPE, PLSQL_CODE_TYPE
/
```

 The PL/SQL objects with NULL `plsql_code_type` are Oracle's internal objects.

- Query the number of the PL/SQL objects that are compiled so far using `utlrp.sql`:

```
SELECT COUNT(*) FROM UTL_RECOMP_COMPILED
/
```

8. Disable the restricted session:

```
SQL> ALTER SYSTEM DISABLE RESTRICTED SESSION
/
```

System altered.

If you want to rollback the compilation mode to interpreted, you can follow the same steps, and replace `dbmsupgnv.sql` with the `dbmsupgin.sql` script.

Tuning PL/SQL code

We have just discussed two key features in the performance management space. The PL/SQL code optimization and compilation can significantly improve runtime performance. Apart from the code optimization techniques, there are several efficient coding practices that may impact overall PL/SQL performance. PL/SQL performance management is often regarded as less tedious, when compared to the database tuning exercise because the area of operation is a program unit and doesn't require the instance to be restarted. In this section, we will discuss some of the selective but key features to improve PL/SQL coding practices for better performance.

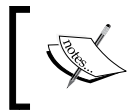
Build secure applications using bind variables

It is highly encouraged to use bind variables in SQL statements while building robust database applications. Bind variables reduce the parsing overhead by enabling an SQL statement to be executed using its already parsed image stored in the shared SQL area of the database memory.

Every SQL statement executed by the SQL engine is parsed, optimized, and executed. Only after the SQL statement is executed is the result fetched and displayed to the user. The SQL execution steps are briefly described, as follows:

- Generate a hash value for the SQL statement.
- Look for a matching hash value of cached cursors in the shared pool.

- If the match is not found, Oracle continues with the statement parsing and optimization stages. This is hard parsing. It is a CPU-intensive operation and involves contention for latches in the shared SQL area.
- If the match is found, reuse the cursor and reduce parsing to a privilege check. This is a soft parse. Thereafter, without needing further optimization, the explain plan is retrieved from the library cache, and the SQL query is executed.



Oracle Database 12c introduced adaptive query optimization, which allows the optimizer to re-optimize the existing explain plans upon subsequent executions of an SQL query.

- The result set is fetched.

In an enterprise application, many `SELECT` statements or transactional statements (`INSERT`, `UPDATE`, or `DELETE`) are similar in structure but they run with different predicate values. The hashes for two of the same SQL statements with different values will never be the same, thus resulting in a hard parse. A hard parse, being a non-scalable expensive operation, degrades the application performance. The only way to get rid of the problem is to encourage soft parsing by maximizing the use of cached cursors and associated explain plans.

The use of bind variables promotes soft parsing. A bind variable is a substitution variable that acts as a placeholder for the literal values. Bind variables enable an SQL statement to appear the same even after multiple runs with different input values. This reduces hard parsing and the existing plans in the library cache can be used to optimize the current SQL statements.



Avoid using hard-coded literals in SQL queries and PL/SQL programs.

For example, the following `SELECT` query will have the same hash value for the different values of employee ids:

```
/*Select EMP table with a bind variable*/  
SELECT  ename,  
        deptno,  
        sal  
FROM    emp  
WHERE   empno = :empid  
/
```

Oracle PL/SQL supports the use of bind variables. All references to a block variable or program argument are treated as a bind variable. In the case of a dynamic SQL, either using `DBMS_SQL` or `EXECUTE IMMEDIATE`, you must use bind variables. Bind variables help dynamic SQL in two ways. First it improves the code performance. Secondly, it reduces the risk of SQL injection by covering the vulnerable areas. Let's conduct a small illustration to see the benefits of bind variables in PL/SQL.

The following PL/SQL block finds the count of distinct object type accessible by the SCOTT user. It executes the `SELECT` query using `EXECUTE IMMEDIATE`. In order to get the accurate results, we will flush the shared pool:

```
connect sys/oracle as sysdba
ALTER SYSTEM FLUSH SHARED_POOL
/
connect scott/tiger
SET SERVEROUT ON

/*Start the PL/SQL block*/
DECLARE

    /*Local block variables */
    l_count NUMBER;
    l_stmt VARCHAR2(4000);
    clock_in NUMBER;
    clock_out NUMBER;

    TYPE v_obj_type is TABLE OF varchar2(100);
    l_obj_type v_obj_type := v_obj_type ('TYPE', 'PACKAGE', 'PROCEDURE',
    'TABLE', 'SEQUENCE', 'OPERATOR', 'SYNONYM');
BEGIN

    /*Capture the start time */
    clock_in := DBMS_UTILITY.GET_TIME ();

    /*FOR loop to iterate the collections */
    FOR I IN l_obj_type.first..l_obj_type.last
    LOOP
        l_stmt := 'SELECT count(*)
                    FROM all_objects
                    WHERE object_type = ''' || l_obj_type(i) || ''';
        EXECUTE IMMEDIATE l_stmt INTO l_count;
    END LOOP;

    /*Capture the end time */
```

```
clock_out := DBMS_UTILITY.GET_TIME ();

DBMS_OUTPUT.PUT_LINE ('Execution time without bind variables:'||TO_
CHAR(clock_out-clock_in));

END;
/

Execution time without bind variables:948

PL/SQL procedure successfully completed.
```

Now, we'll rewrite the above PL/SQL block using bind variables:

```
connect sys/oracle as sysdba
ALTER SYSTEM FLUSH SHARED_POOL
/
connect scott/tiger
SET SERVEROUT ON

/*Start the PL/SQL block */
DECLARE

/*Local block variables */
l_count NUMBER;
l_stmt VARCHAR2(4000);
clock_in NUMBER;
clock_out NUMBER;

TYPE v_obj_type is TABLE OF varchar2(100);
l_obj_type v_obj_type := v_obj_type ('TYPE', 'PACKAGE', 'PROCEDURE',
'TABLE', 'SEQUENCE', 'OPERATOR', 'SYNONYM');
BEGIN

/*Capture the start time */
clock_in := DBMS_UTILITY.GET_TIME ();

/*FOR loop to iterate the collection */
FOR I IN l_obj_type.first..l_obj_type.last
LOOP

/*Build the SELECT statement by using bind variable*/
l_stmt := 'SELECT count(*)
          FROM all_objects
          WHERE object_type = :p1';
/*Use dynamic SQL to execute the SELECT statement*/
EXECUTE IMMEDIATE l_stmt INTO l_count USING l_obj_type(i);
```

```

END LOOP;

clock_out := DBMS_UTILITY.GET_TIME ();

DBMS_OUTPUT.PUT_LINE ('Execution time with bind variables:' || TO_
CHAR(clock_out-clock_in));

END;
/

Execution time with bind variables:121

PL/SQL procedure successfully completed.

```

The block with the bind variables gets executed at least 8 times faster than the one which uses literals in the SQL query inside the PL/SQL block. The reason for the performance gain is the soft parsing of the `SELECT` statement.

In the case of legacy applications or access protected applications, you might face difficulties in modifying the code to include the bind variables. Oracle makes this daunting task easier by controlling the cursor sharing behavior through a switch. You can set `CURSOR_SHARING` parameter to `EXACT` or `FORCE` to share the cursors across the sessions in a database instance. If the parameter is set to `EXACT`, only the SQL statements that have exactly the same structure and parameters will be shared. If the parameter is set to `FORCE`, then Oracle attempts to substitute all the literals in a query with system-generated bind variables. By default, the parameter value is set to `EXACT`. Cursor sharing greatly improves the performance. At the same time, forced cursor sharing involves extra effort in searching for the same cursor in the shared pool.



The `SIMILAR` value of `CURSOR_SHARING` has been deprecated in Oracle Database 12c.

Call parameters by reference

A PL/SQL program invocation can pass a parameter either by its value or by reference. Arguments in the `IN` mode are passed by reference (the default) while the arguments in the `OUT` and `IN OUT` modes are passed by value. In the case of pass by value method, the parameter value before the invocation has to be stored in a temporary variable. This temporary variable is a memory variable and is used to assign the value back to the parameter, if the subprogram call ends in an unhandled exception. If the PL/SQL program accepts composite data types and complex variables in the `OUT` and `IN OUT` modes, copying and maintaining the temporary variable can cause a performance overhead, thereby slowing down the program execution.

Oracle recommends that you specify the `NOCOPY` hint for the pass by value parameters. A parameter with the pass mode as `OUT NOCOPY` or `IN OUT NOCOPY` is passed by reference, thus avoiding the overhead of maintaining a temporary variable. For simple data values, which are generally small, the gain will be negligible.

Avoiding an implicit data type conversion

Oracle converts the data types to a compatible type during the operations like assignment, predicates with value comparison, or passing parameters to subprograms. This is an implicit activity that follows a data type precedence to convert the data types. However, it may cause an overhead when executing a critical program. The following are the best practices that can reduce the implicit data type conversion in the programs:

- Understand that SQL data types and PL/SQL data types have different internal representations. You should explicitly convert the SQL type variables to PL/SQL types and then use those in an expression. For example, `PLS_INTEGER` is a PL/SQL data type that uses machine arithmetic to speed up compute intensive operations. If `NUMBER` type variables are used in an expression, Oracle uses library arithmetic, and therefore, there are no performance gains.
- A variable getting assigned to a table column should be of the same data type to avoid an implicit data type conversion. As a best practice, declare the variables with the `%TYPE` attribute. Similarly, a record can be declared with the `%ROWTYPE` attribute. Such variable declarations always remain synchronized with the database columns, improve the program's scalability, and reduce human errors. Consider the following declarations:

```
empname emp.ename%type
type emp_rec is record of emp%rowtype;
CURSOR c IS
SELECT prod_id, prod_code, prod_name
FROM products;
prod_rec c%rowtype;
```

- Use the SQL conversion function to convert the to-be assigned variable to the target variable's data type. Some of the conversion functions are `TO_CHAR`, `TO_NUMBER`, `TO_DATE`, `TO_TIMESTAMP`, `CAST`, and `ASCIISTR`.

- Pass a variable of correct data type while invoking the PL/SQL program units with parameters. You can also include overloaded subprograms in a PL/SQL package that accepts the parameters of the different data types. You can invoke the packaged subprogram that best matches the available set of variables. For example, the following package has two overloaded functions. You can invoke either of the two depending on the type of variables available:

```
CREATE OR REPLACE PACKAGE pkg_sum AS
    FUNCTION p_sum_series (p_term PLS_INTEGER, p_factor PLS_INTEGER)
        RETURN PLS_INTEGER;

    FUNCTION p_sum_series (p_term NUMBER, p_factor NUMBER)
        RETURN NUMBER;
END;
/
```

Understanding the NOT NULL constraint

A block variable can be declared `NOT NULL` at the time of the declaration. Oracle performs the nullability test every time a value is assigned to the `NOT NULL` variable. In large PL/SQL program units, the nullability check can be an overhead and impact the PL/SQL block's runtime performance. Instead, it is advised to have an explicit nullability check for the variable in the executable section of the block.

The following PL/SQL program compares the performance of a `NOT NULL` variable and nullable local variable:

```
/*Enable the SERVEROUTPUT to display block results*/
SET SERVEROUTPUT ON

/*Start the PL/SQL block*/
DECLARE
    l_nn_num  NUMBER NOT NULL := 0;
    l_num     NUMBER := 0;
    clock_in  NUMBER;
    clock_out NUMBER;
BEGIN

    /*Capture the start time*/
    clock_in := DBMS_UTILITY.GET_TIME();

    /*Start the loop*/
    FOR I IN 1..1000000
    LOOP
```



```
        l_nn_num := l_nn_num + i;
    END LOOP;

    clock_out := DBMS_UTILITY.GET_TIME();

    /*Compute the time difference and display*/
    DBMS_OUTPUT.PUT_LINE('Time for NOT NULL:' || TO_CHAR(clock_out-clock_in));

    /*Capture the start time*/
    clock_in := DBMS_UTILITY.GET_TIME();

    /*Start the loop*/
    FOR I IN 1..1000000
    LOOP
        l_num := l_num + i;
    END LOOP;

    clock_out := DBMS_UTILITY.GET_TIME();

    /*Compute the time difference and display*/
    DBMS_OUTPUT.PUT_LINE('Time for NULL:' || TO_CHAR(clock_out-clock_in));

END;
/

Time for NOT NULL:111
Time for NULL:76

PL/SQL procedure successfully completed.
```

In the preceding PL/SQL block, the nullable variable outperforms the NOT NULL variable by one and a half times. The reason for this performance gain is the reduction in the number of nullability checks.

Selection of an appropriate numeric data type

The PLS_INTEGER data type is a part of the NUMBER data type family. Most of the number or integer data types are generic ones and the reason behind this is to support code portability. However, PLS_INTEGER is specifically designed for performance. It was introduced in Oracle Database Release 7 to speed up the computation-intensive operations through native machine arithmetic instead of library arithmetic.

The following PL/SQL block compares the performance of the PLS_INTEGER and NUMBER variables:

```

/*Enable the SERVEROUTPUT to display block results*/
SET SERVEROUTPUT ON

/*Start the PL/SQL block*/
DECLARE
    l_pls_int PLS_INTEGER := 1;
    l_num     NUMBER:= 1;
    l_factor  PLS_INTEGER := 2;
    clock_in  NUMBER;
    clock_out NUMBER;
BEGIN

/*Capture the start time*/
    clock_in := DBMS_UTILITY.GET_TIME();

/*Begin the loop to perform a mathematical calculation*/
    FOR I IN 1..10000000
    LOOP
/*The mathematical operation increments a variable by one*/
        l_num := l_num + l_factor;
    END LOOP;
    clock_out := DBMS_UTILITY.GET_TIME();
/*Display the execution time consumed*/
    DBMS_OUTPUT.PUT_LINE('Time by NUMBER:' || TO_CHAR(clock_out-clock_in));

/*Capture the start time*/
    clock_in := DBMS_UTILITY.GET_TIME();

/*Begin the loop to perform a mathematical calculation*/
    FOR J IN 1..10000000
    LOOP

/*The mathematical operation increments a variable by one*/
        l_pls_int := l_pls_int + l_factor;
    END LOOP;
    clock_out := DBMS_UTILITY.GET_TIME();

/*Display the time consumed*/
    DBMS_OUTPUT.PUT_LINE('Time by PLS_INTEGER:' || TO_CHAR(clock_out-clock_in));

```

```
END;  
/  
  
Time by NUMBER:231  
Time by PLS_INTEGER:69  
  
PL/SQL procedure successfully completed.
```

The performance of the `PLS_INTEGER` variable is at least three times better than a `NUMBER` variable. An arithmetic expression having a mix of the `PLS_INTEGER` and `NUMBER` type variables will use library arithmetic and no performance gains will be obtained.

`PLS_INTEGER` is a 32-bit data type that can store values in the range of -2147483648 to 2147483647. It accepts integer values only. If a floating value is assigned to `PLS_INTEGER`, it is rounded to the nearest integer. If the `PLS_INTEGER` precision range is violated, Oracle raises an `ORA-01426: numeric overflow` exception. To resolve such scenarios, Oracle 11g introduced a new subtype of `PLS_INTEGER`, which is known as `SIMPLE_INTEGER`. The `SIMPLE_INTEGER` data type has the same range as that of `PLS_INTEGER`, but in the case of numeric overflows, it is automatically set to -2147483648 instead of raising an exception. As the overflow check is suppressed for `SIMPLE_INTEGER`, it is faster than `PLS_INTEGER`.



`SIMPLE_INTEGER` is a NOT NULL data type; therefore, all local variables must be initialized or defaulted to a definitive value.

The following PL/SQL anonymous block declares a `SIMPLE_INTEGER` variable and defaults it to 2147483646, that is, the last value in the precision range. In the program body, the variable is incremented by 1. Let's check what happens:

```
/*Enable the SERVEROUTPUT to display block results*/  
SET SERVEROUTPUT ON  
  
/*Start the PL/SQL block*/  
DECLARE  
    l_simple SIMPLE_INTEGER:= 2147483646;  
BEGIN  
    /*Increment the variable by 1*/  
    l_simple:= l_simple +1;  
    DBMS_OUTPUT.PUT_LINE('After 1st increment:'|| l_simple);  
  
    /*Re-Increment the variable by 1*/  
    l_simple:= l_simple +1;  
    DBMS_OUTPUT.PUT_LINE('After 2nd increment:'|| l_simple);  
EXCEPTION
```

```
WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Numeric Overflow exception occurred');
END;
/
```

```
After 1st increment:2147483647
After 2nd increment:-2147483648
```

PL/SQL procedure successfully completed.

Bulk processing in PL/SQL

Bulk data processing is one of the key features in performance management. Bulk SQL is a feature that reduces context switches between the SQL and PL/SQL processing engines. One of the biggest benefits of PL/SQL development is its seamless integration with the SQL language. The SQL statements can directly appear in the PL/SQL program body. In a PL/SQL unit with the SQL statements and PL/SQL constructs, the PL/SQL engine and SQL engine communicate through a context switch. The PL/SQL engine switches the program context to the SQL engine to execute the SQL statements and get the result. The frequent back-and-forth context switching between the processing engines causes a severe overhead and slows down the program execution. The row-by-row processing of records is quite slow compared to bulk processing.

Oracle implements bulk operations through the `BULK COLLECT` clause and the `FORALL` statement. The `BULK COLLECT` clause retrieves multiple records in a single fetch while the `FORALL` statement can process multiple DML statements in groups rather than row by row. In many implementations, the performance gains by the use of bulk processing features have been phenomenal.

BULK COLLECT

You can use `BULK COLLECT` in:

- The `SELECT...INTO` statement
- The `RETURNING INTO` clause with the `UPDATE` or `DELETE` statement
- The `FETCH INTO` clause with an explicit cursor

The bulk collect feature depends largely on collections and can be used with all the three forms of collections, that is, associative arrays, nested tables, and varrays. Multiple records can be fetched into a collection in a single fetch. This reduces the number of the context switches between the two processing engines. During the fetch operation, the collection variables get densely populated. In the case of no rows being fetched, the collections are left empty resulting in an empty collection.



Bulk operations are a CPU-intensive operation. They cannot be parallelized.

Until Oracle Database 9i, `BULK COLLECT` could be used only with static SQL statements, but starting with Oracle Database 10g, it can be used in dynamic SQL statements too.

The `CREATE TABLE` script creates the test table with the sample data from the `ALL_OBJECTS` dictionary view to be used in this illustration. The `ALL_OBJECTS` dictionary view contains details of the schema objects. During our illustration, we will work with the object id, object type, and object name columns of the table:

```
CREATE TABLE local_objects AS
SELECT * FROM all_objects
/
```

```
/*Query the record count*/
SELECT COUNT(*)
FROM local_objects
/
```

```
COUNT(*)
-----
       73673
```

The following PL/SQL block opens a cursor, fetches row by row, and counts the number of procedures that can be accessed by the `SCOTT` user:

```
SET SERVEROUTPUT ON
/*Start the PL/SQL block*/
DECLARE

/*Local PL/SQL variables*/
obj_id    local_objects.object_id%TYPE;
obj_type  local_objects.object_type%TYPE;
obj_name  local_objects.object_name%TYPE;

counter   NUMBER;
clock_in  NUMBER;
clock_out NUMBER;

/*Cursor to fetch the object details*/
CURSOR c IS
    SELECT object_id, object_type, object_name
```

```

FROM local_objects;
BEGIN

    /*Capture the start time*/
    clock_in := DBMS_UTILITY.GET_TIME();
    OPEN c;

    LOOP
        FETCH c INTO obj_id, obj_type, obj_name;
        EXIT WHEN c%NOTFOUND;
        /*Count the number of procedures in the test table*/
        IF obj_type = 'PROCEDURE' THEN
            counter := counter+1;
        END IF;

    END LOOP;
    CLOSE c;

    /*Capture the end time*/
    clock_out := DBMS_UTILITY.GET_TIME();
    DBMS_OUTPUT.PUT_LINE ('Time taken in row fetch:'||to_char (clock_
out-clock_in));

END;
/

```

Time taken in row fetch:369

PL/SQL procedure successfully completed.

The row-by-row fetch took 369 hsec to get executed. There were 73673 context switches made between the PL/SQL and SQL engines. Now, let's apply the bulk fetch techniques to the preceding block, and check the performance gains:

```

SET SERVEROUTPUT ON
/*Start the PL/SQL block*/
DECLARE

    /*Declare the local record and table collection*/
    TYPE obj_rec IS RECORD
    (obj_id local_objects.object_id%TYPE,
    obj_type local_objects.object_TYPE%TYPE,
    obj_name local_objects.object_name%TYPE);

    TYPE obj_tab IS TABLE OF obj_rec;
    t_all_objs obj_tab;

```

```
counter    NUMBER;
clock_in   NUMBER;
clock_out  NUMBER;
BEGIN

    /*Capture the start time*/
    clock_in := DBMS_UTILITY.GET_TIME();

    /*Select query to bulk fetch multi-record set in nested table
collection*/
    SELECT object_id, object_TYPE, object_name
    BULK COLLECT INTO t_all_objs
    FROM local_objects;

    /*Loop through the collection to count number of procedures*/
    FOR I IN t_all_objs.FIRST..t_all_objs.LAST
    LOOP
        IF (t_all_objs(i).obj_type = 'PROCEDURE') THEN
            counter := counter+1;
        END IF;
    END LOOP;

    /*Capture the end time*/
    clock_out := DBMS_UTILITY.GET_TIME();
    DBMS_OUTPUT.PUT_LINE ('Time taken in bulk fetch:'||to_char (clock_
out-clock_in));
END;
/

Time taken in bulk fetch:35

PL/SQL procedure successfully completed.
```

Well, the bulk SQL is around 10 times faster than the usual fetch operation. The reason is that there was only one context switch made between the SQL and PL/SQL engines. Note that the nested table collection variable is not required to be initialized for bulk operations.



The BULK COLLECT operation does not raise the NO_DATA_FOUND exception.

Bulk processing is a CPU-intensive operation, and fetching into a collection consumes your session memory. If the bulk collect fetches a large number of rows and the session memory is not large enough, you might experience a hung session or an abnormal termination. To resolve such cases, Oracle provides the `LIMIT` clause to control the number of records retrieved in a single fetch operation. The following PL/SQL block uses the `LIMIT` clause to control the number of records fetched during the `FETCH` operation:

```

SET SERVEROUTPUT ON
/*Start the PL/SQL block */
DECLARE

    /*Local block variables - object type record and nested table*/
    TYPE obj_rec IS RECORD
    (obj_id local_objects.object_id%TYPE,
     obj_type local_objects.object_TYPE%TYPE,
     obj_name local_objects.object_name%TYPE);

    TYPE obj_tab is table of obj_rec;
    t_all_objs obj_tab;

    counter NUMBER;
    p_rec_limit NUMBER := 100;
    clock_in NUMBER;
    clock_out NUMBER;

    /*Cursor to fetch object details from LOCAL_OBJECTS*/
    CURSOR C IS
    SELECT object_id, object_TYPE, object_name
    FROM local_objects;
BEGIN

    /*Capture the start time*/
    clock_in := DBMS_UTILITY.GET_TIME();
    /*Open the cursor*/
    OPEN c;
    LOOP

        /*Bulk fetch the records by specifying the limit*/
        FETCH c BULK COLLECT INTO t_all_objs LIMIT p_rec_limit;
        EXIT WHEN c%NOTFOUND;
        /*FOR loop to count the procedures */
        FOR I IN t_all_objs.FIRST..t_all_objs.LAST
            LOOP

```



```
        IF (t_all_objs(i).obj_type = 'PROCEDURE') THEN
            counter := counter+1;
        END IF;
    END LOOP;
    t_all_objs.DELETE;
END LOOP;
CLOSE c;

clock_out := DBMS_UTILITY.GET_TIME();

DBMS_OUTPUT.PUT_LINE ('Time taken in controlled fetch:' || to_char
(clock_out-clock_in));

END;
/

Time taken in controlled fetch:94

PL/SQL procedure successfully completed.
```

A controlled bulk operation, limited to 100 records per bulk fetch, took 94 hsec to get executed, which is still 4 times better than the row-by-row fetch operation. This time the program makes 737 context switches between the PL/SQL and SQL engines.

FORALL

More than iterative, FORALL is a declarative statement. The DML statements specified in FORALL are generated once, but they send in bulk to the SQL engine for processing, thus making only a single context switch. There can be only one DML statement in FORALL that can be INSERT, UPDATE, or DELETE, or the FORALL statement can be used, as per the following syntax:

```
FORALL index IN
[
    lower_bound...upper_bound |
    INDICES OF indexing_collection |
    VALUES OF indexing_collection
]
[SAVE EXCEPTIONS]
[DML statement]
```

If the DML statements contain bind variables, the bulk SQL binds all the statements in a single step. This is known as **bulk binding**.

Let's set up the environment for our performance test. The performance test will compare the execution time of a FOR loop versus a FORALL statement:

```

/*Create table T1 with basic object columns*/
CREATE TABLE T1
AS
SELECT object_id, object_type, object_name
FROM all_objects
WHERE 1=2
/

/*Create table T2 with basic object columns*/
CREATE TABLE T2
AS
SELECT object_id, object_type, object_name
FROM all_objects
WHERE 1=2
/

```

The following PL/SQL block bulk collects the data from the LOCAL_OBJECTS table, creates an interim collection (associative array) for only the synonym information, and inserts it into two different tables. For verification, we will query the records inserted in both the tables:

```

SET SERVEROUTPUT ON
/*Start the PL/SQL block*/
DECLARE

/*Local block variables - object type record and nested table
collection*/
TYPE obj_rec IS RECORD
(obj_id local_objects.object_id%TYPE,
 obj_type local_objects.object_TYPE%TYPE,
 obj_name local_objects.object_name%TYPE);

TYPE obj_tab IS TABLE OF obj_rec;
TYPE syn_tab IS TABLE OF obj_rec index by pls_integer;
t_all_objs obj_tab;
t_all_syn syn_tab;

counter NUMBER := 1;
clock_in NUMBER;
clock_out NUMBER;
BEGIN

```

```
    /*BULK COLLECT the data from LOCAL_OBJECTS in nested table
collection*/
    SELECT object_id, object_TYPE, object_name
    BULK COLLECT INTO t_all_objs
    FROM local_objects;
    FOR I IN t_all_objs.FIRST..t_all_objs.LAST
    LOOP
        /*Capture all synonyms in another collection*/
        IF (t_all_objs(i).obj_type = 'SYNONYM') THEN
            t_all_syn(t_all_syn.count+1) := t_all_objs(i);
        END IF;
    END LOOP;

    clock_in := DBMS_UTILITY.GET_TIME();

    /*FOR loop to insert the data in table T1*/
    FOR I IN t_all_syn.first..t_all_syn.last
    LOOP
        INSERT INTO t1 VALUES t_all_syn(i);
    END LOOP;

    clock_out := DBMS_UTILITY.GET_TIME();

    DBMS_OUTPUT.PUT_LINE ('Time taken by FOR loop:'||to_char (clock_
out-clock_in));

    clock_in := DBMS_UTILITY.GET_TIME();

    /*FORALL statement to insert the data in table T2*/
    FORALL I IN t_all_syn.first..t_all_syn.last
        INSERT INTO t2 VALUES t_all_syn(i);

    clock_out := DBMS_UTILITY.GET_TIME();

    DBMS_OUTPUT.PUT_LINE ('Time taken by FORALL:'||to_char (clock_out-
clock_in));
END;
/

Time taken by FOR loop:1637
Time taken by FORALL:29

PL/SQL procedure successfully completed.
```

```

/*Query the record count in T2*/
SQL> SELECT COUNT(*) FROM t2
/

```

```

COUNT(*)
-----
37031

```

```

/*Query the record count in T1*/
SQL> SELECT COUNT(*) FROM t1
/

```

```

COUNT(*)
-----
37031

```

This is phenomenal! The `FORALL` statement inserted more than 37000 records in less than a second, whereas the `FOR LOOP` statement took 16 sec to insert the same number of records. The reason is again the context switch between the processing engines; one switch against 37000 context switches.

In the case of a sparse collection, use `INDICES OF` or `VALUES OF` in the `FORALL` statement. It will use only those indexes of the collection that hold a value. The following PL/SQL block uses the `FORALL` statement to insert a sparse collection into the T2 table. To sparse the interim collection, the synonyms starting with `SYS%` are deleted from it:

```

/*Truncate table T2 for the current test*/
TRUNCATE TABLE t2
/
SET SERVEROUTPUT ON

/*Start the PL/SQL block*/
DECLARE

/*Local block variables - object type record and nested table
collection*/
TYPE obj_rec IS RECORD
(obj_id local_objects.object_id%TYPE,
 obj_type local_objects.object_TYPE%TYPE,
 obj_name local_objects.object_name%TYPE);

TYPE obj_tab IS TABLE OF obj_rec;
TYPE syn_tab IS TABLE OF obj_rec index by pls_integer;
t_all_objs obj_tab;

```

```
t_all_syn syn_tab ;

counter NUMBER := 1;
clock_in NUMBER;
clock_out NUMBER;
BEGIN

    /*BULK COLLECT the data from LOCAL_OBJECTS in nested table
collection*/
    SELECT object_id, object_TYPE, object_name
    BULK COLLECT INTO t_all_objs
    FROM local_objects;

    /*FOR loop to collect all synonyms in nested table collection*/
    FOR I IN t_all_objs.FIRST..t_all_objs.LAST
    LOOP
        IF (t_all_objs(i).obj_type = 'SYNONYM') THEN
            t_all_syn(t_all_syn.count+1) := t_all_objs(i);
        END IF;
    END LOOP;

    /*Delete all the synonyms whose names starts with SYS%*/
    FOR I in 1..t_all_syn.count
    LOOP
        IF t_all_syn(i).obj_name like 'SYS%' THEN
            t_all_syn.delete (i);
        END if;
    END LOOP;

    clock_in := DBMS_UTILITY.GET_TIME();

    /*Insert the sparse collection using INDICES of clause*/
    FORALL I IN INDICES OF t_all_syn
        INSERT INTO t2 VALUES t_all_syn(i);

    clock_out := DBMS_UTILITY.GET_TIME();

    DBMS_OUTPUT.PUT_LINE ('Time taken by FORALL: '||to_char (clock_out-
clock_in));

END;
/
```

```

Time taken by FORALL:30

PL/SQL procedure successfully completed.

/*Query the record count in table T2*/
SQL> select count(*) from t2;

COUNT(*)
-----
37025

```

There were six synonyms starting with SYS% that were not inserted in this run. If you hadn't used the `INDICES OF` clause, the block would have terminated with the `ORA-22160: element at index [3] does not exist exception`.

FORALL and exception handling

`FORALL` is specifically designed for bulk transactions. Any faulty transaction in the bulk operation may cause the entire bulk operation to abort with an exception, thereby rolling back all the changes made by the earlier transactions of the same DML statement. Oracle gracefully deals with such scenarios by saving the erroneous record and exception information separately and by continuing the DML execution. There are two ways to handle an exception in the `FORALL` statement:

1. Abort the `FORALL` execution but commit the changes made by the earlier transactions: Traditional exception handling but with a `COMMIT` in the exception handler. For example, the following exception handle will commit the transactions that have been executed already:

```

EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE (SQLERRM);
    COMMIT;
    RAISE;

```

2. Continue the `FORALL` execution and save the failed transactions: Use `SAVE EXCEPTIONS` with the `FORALL` statement. The feature is known as bulk exception handling. With `SAVE EXCEPTION`, Oracle stores the faulty DML details in the bulk exception logger known as `SQL%BULK_EXCEPTIONS`. After the `FORALL` execution is over, database administrators can look into the exception log and troubleshoot the defective records. The defective records are skipped and logged under the `SQL%BULK_EXCEPTIONS` pseudocolumn. For example, if `FORALL` generated 5000 update statements, out of which 13 were failed transactions, 4987 records will still be updated. The 13 defective transactions will be logged in the `SQL%BULK_EXCEPTIONS` array structure with the cursor index.

The `%BULK_EXCEPTIONS` attribute maintains two fields—`ERROR_INDEX` and `ERROR_CODE`. `ERROR_INDEX` stores the defect record index where the exception was raised, while `ERROR_CODE` records the exception message. `%BULK_EXCEPTIONS.COUNT` stores the count of exceptions raised during the execution of the `FORALL` statement. Note that the standard error code captured by the `ERROR_CODE` attribute is not prefixed with the negative (-) sign. Therefore, in order to fetch its equivalent error message, pass the error code multiplied by -1 to the `SQLERRM` function.

In the last illustration, we explicitly removed the values starting with `SYS%`. For the current demonstration, we will add a check constraint on the `OBJECT_NAME` column of the `T2` table. The following script adds the check constraint:

```
/*Truncate table T2 for current test*/
TRUNCATE TABLE t2
/

/*Add check constraint on OBJECT_NAME*/
ALTER TABLE t2
ADD CONSTRAINT t2_obj_name CHECK (object_name NOT LIKE 'SYS%')
/
```

Now, we will run the following PL/SQL block to insert the dense collection into the `T2` table. As `SAVE EXCEPTIONS` has been specified along with the `FORALL` statement, if there are any exceptions while loading, they will be saved in the bulk exception log:

```
SET SERVEROUTPUT ON

/*Start the PL/SQL block*/
DECLARE

/*Local block variables - object type record and nested table
collection*/
TYPE obj_rec IS RECORD
(obj_id local_objects.object_id%TYPE,
 obj_type local_objects.object_TYPE%TYPE,
 obj_name local_objects.object_name%TYPE);

TYPE obj_tab IS TABLE OF obj_rec;
TYPE syn_tab IS TABLE OF obj_rec index by pls_integer;
t_all_objs obj_tab;
t_all_syn syn_tab ;

counter NUMBER := 1;
clock_in NUMBER;
clock_out NUMBER;
```

```

/*Declare user defined exception for error number -24381*/
bulk_errors EXCEPTION;
PRAGMA EXCEPTION_INIT(bulk_errors, -24381);
BEGIN

    /*Bulk collect object details in nested table collection variable*/
    SELECT object_id, object_TYPE, object_name
    BULK COLLECT INTO t_all_objs
    FROM local_objects;

    /*Filter all synonyms and capture in another collection*/
    FOR I IN t_all_objs.FIRST..t_all_objs.LAST
    LOOP
        IF (t_all_objs(i).obj_type = 'SYNONYM') THEN
            t_all_syn(t_all_syn.count+1) := t_all_objs(i);
        END IF;
    END LOOP;

    /*FORALL statement to insert the records in table T2*/
    FORALL I IN 1..t_all_syn.count

    /*Save faulty records in bulk collection cursor attributes*/
    SAVE EXCEPTIONS
        insert into t2 values t_all_syn(i);
    EXCEPTION

    /*Exception handler to query the failed transactions*/
    WHEN BULK_ERRORS THEN
        FOR J IN 1..SQL%BULK_EXCEPTIONS.COUNT
        LOOP
            DBMS_OUTPUT.PUT_LINE('-----');
            DBMS_OUTPUT.PUT_LINE('Error in INSERT: ' || SQL%BULK_EXCEPTIONS(J).
            ERROR_INDEX);
            DBMS_OUTPUT.PUT_LINE('Error Message is: ' || sqlerrm('-' || SQL%BULK_
            EXCEPTIONS(J).ERROR_CODE));
        END LOOP;

    END;
/

```



```
-----
Error in INSERT:3
Error Message is: ORA-02290: check constraint (.) violated
-----
Error in INSERT:2089
Error Message is: ORA-02290: check constraint (.) violated
-----
Error in INSERT:2091
Error Message is: ORA-02290: check constraint (.) violated
-----
Error in INSERT:2093
Error Message is: ORA-02290: check constraint (.) violated
-----
Error in INSERT:2094
Error Message is: ORA-02290: check constraint (.) violated
-----
Error in INSERT:3545
Error Message is: ORA-02290: check constraint (.) violated

PL/SQL procedure successfully completed.

SQL> select count(*) from t2;

COUNT(*)
-----
      37025
```

The transactions that failed to get executed due to the check constraint violation were saved in the `BULK_EXCEPTIONS` cursor attribute. Besides these six records, all the other data was inserted, and this can be confirmed by a `SELECT` query on the table.

Summary

In this chapter, we have discussed the features that can tune your PL/SQL code to run faster. We have also seen how code compilation and optimization can speed up the performance of the PL/SQL programs. In the later half, we discussed the PL/SQL tuning features in detail with the help of demonstrations.

In the next chapter, we will focus on one of the major new features introduced in Oracle Database 11g. The feature is known as result caching that is primarily built to accelerate the performance of queries that are repeatedly executed. We will discuss the different flavors of result caching in the next chapter.

Practice exercise

- Identify the nature of the program that is best suited for the interpreted mode of compilation.
 1. The program unit contains multiple SQL statements.
 2. The program unit has just been developed and is in the debug stage.
 3. The program unit uses collections and bulk bind statements.
 4. The program unit is in the production phase.
- Choose the correct statements about the real native compilation mode in Oracle 11g.
 1. The compilation method uses the C compiler to convert the program into an equivalent C code.
 2. The compilation method mounts the shared libraries through the `PLSQL_NATIVE_LIBRARY_DIR` and `PLSQL_NATIVE_LIBRARY_SUBDIR_COUNT` parameters.
 3. The compilation does not use the C compiler but converts the program unit directly to the M code.
 4. The real native compilation is supported for RAC environments and participates in the backup recovery processes.
- Determine the behavior of the `PLSQL_OPTIMIZE_LEVEL` optimizer when it has been set to 3.
 1. The optimizer would inline the programs that are necessary.
 2. The optimizer would inline all the programs irrespective of the gains.
 3. The optimizer would inline only those subprograms which have `PRAGMA INLINE`.
 4. The setting has no effect on the inlining of the subprograms.
- Choose the correct statements about the compilation setting in Oracle.
 1. From Oracle 11g, the default value of `PLSQL_CODE_TYPE` is `NATIVE`.
 2. An object can be recompiled in a compilation mode that is different from the current database setting.
 3. During the database upgrade, `PLSQL_CODE_TYPE` must be modified in the `pfile` instance.
 4. In real native compilation, the libraries generated are stored in a secured file system.

- Identify the tuning tips in the following PL/SQL block:

```
DECLARE
CURSOR C IS
    SELECT ENAME, SAL, COMM
    FROM EMPLOYEES;
    L_COMM NUMBER;
BEGIN
    FOR I IN C
    LOOP
        L_COMM := I.SAL + ((I.COMM/100) * (I.SAL * 12));
        DBMS_OUTPUT.PUT_LINE (I.ENAME || ' earns ' || L_COMM || ' as
commission');
    END LOOP;
END;
/
```

1. Use BULK COLLECT to select the employee data.
 2. Declare L_COMM as NOT NULL.
 3. Use PLS_INTEGER for L_COMM.
 4. No tuning is required.
- Which of these statements are true about inlining in PL/SQL subprograms?
 1. The optimizer can inline only the standalone stored functions.
 2. The optimizer can inline only the locally declared subprograms.
 3. Inlining is always useful in the performance irrespective of the size of the subprogram.
 4. The optimizer cannot identify any subprogram for inlining when the optimizer level is set at 0.
 - Examine the following code and determine the output:

```
DECLARE
FUNCTION F_ADD (P_NUM NUMBER)
RETURN NUMBER
IS
BEGIN
    RETURN P_NUM + 10;
END;
BEGIN
FOR I IN 122..382
LOOP
    PRAGMA INLINE (F_ADD, 'YES');

```

```

        L_SUM := L_SUM + F_ADD (I);
    END LOOP;
    END;
/

```

1. PLSQL_OPTIMIZE_LEVEL is set to 2.
 2. The F_ADD local function would not be called inline unless PLSQL_OPTIMIZE_LEVEL is set to 3.
 3. The F_ADD local function may be called inline because PLSQL_OPTIMIZE_LEVEL is set to 2.
 4. The F_ADD local function would be called inline because PRAGMA INLINE marks it for inline.
 5. Inlining cannot be done for locally declared subprograms.
- The libraries generated from real native compilation are stored in the SYSAUX tablespace.
 1. True
 2. False
 - Suggest the tuning considerations in the following PL/SQL block:

```

DECLARE
    L_SUM NATURALN := 0;
    L_ID VARCHAR2(10);
BEGIN
    L_ID := 256;
    L_SUM := L_ID * 1.5;
END;

```

1. The datatype of L_SUM can be changed to NATURAL and nullability can be verified in the executable section.
2. L_SUM must not be initialized with zero.
3. The 1.5 multiple must be assigned to a variable.
4. L_ID must be of an appropriate datatype such as NUMBER or PLS_INTEGER.

- Identify the correct statements about PRAGMA INLINE.
 1. It is the fifth pragma in Oracle besides AUTONOMOUS_TRANSACTION, EXCEPTION_INIT, RESTRICT_REFERENCE, and SERIALLY_REUSABLE.
 2. It does not work for overloaded functions.
 3. It does not work for PLSQL_OPTIMIZE_LEVEL = 1.
 4. PRAGMA INLINE (<Function name>, ' YES') is meaningless at PLSQL_OPTIMIZE_LEVEL = 3, because the optimizer inlines all the subprograms.

9

Result Cache

In the last chapter, we learned quite a few techniques to tune PL/SQL code. By now, you must have got the idea that tuning is nothing less than an art that comes by practice and grows with experience. The better you understand the data and the application, the higher the probability of tuning the right areas. Most DBAs around the world are familiar with the commonly used tuning practices such as query rewriting, column indexing, instance optimization, materialized views, and PL/SQL code optimization.

As Oracle Database professionals, we might already be aware of multiple caches resident in the database instance architecture (to name a few: the buffer cache, library cache, dictionary cache, or recycle cache). Primarily, caches are meant to hold data so that the data access operations are served faster.

Oracle Database 11g Release 2 introduced a new cache component within the shared pool, known as the **Server Result Cache**, for a specific job. The **Result Cache** enables caching of results from an SQL query or even a PL/SQL function in the Server Result Cache. This chapter will discuss the Result Cache feature in detail. We will understand how to configure the result cache in a database and what makes it a brilliant feature. The outline of the chapter is as follows:

- Oracle Database 11g Result Cache:
 - What is Server Result Cache?
 - Configuring the Server Result Cache
 - Result Cache versus Buffer Cache
 - Result Cache versus Database In-Memory
 - Result Cache versus the In-Memory Database Cache
- SQL Result Cache

- PL/SQL function Result Cache
- OCI client Result Cache
- The `DBMS_RESULT_CACHE` package
- Result cache in Real Application Clusters

Oracle Database 11g Result Cache

Oracle Database 11g offered plenty of performance management features; one of those was server side result caching. Result caching implements a caching mechanism in an Oracle Database. Using this feature, you can cache the results of an SQL query or a PL/SQL function within a designated area in the **SGA (System Global Area)**, known as Server Result Cache.



The result caching feature is available in Oracle Database Enterprise Edition only.

Conventionally, when a query is executed for the first time, Oracle looks for the required data blocks in the buffer cache first. If the data blocks are already in the buffer cache (because previous `SELECT` queries had retrieved them), the current SQL query gets executed using that data. If not, Oracle performs physical I/Os to fetch the table data from the disk into the buffer cache and then moves ahead for the query processing. The next time the same `SELECT` query with the identical predicates and inputs is re-executed, in the same user session – Oracle performs logical I/Os to read the data from the buffer cache and execute the query, but still goes through the entire query processing cycle.

Result Cache is intended to tune the scenarios where a query is executed more than once for the same predicates and literals. Oracle stores the result of a query in the Server Result Cache during its first execution. On the subsequent execution of an identical `SELECT` query (with the same predicates and literals), the result is fetched direct from the server cache directly without re-executing the query, thereby achieving impressive performance gains. Where do these gains come from? Well, these gains are collectively contributed by the reduction in logical and physical I/Os, sorts, and, hence, the CPU consumption.

Result caching can be enabled at three levels:

- **SQL query Result Cache:** Results from SQL queries can be cached in the database server.
- **PL/SQL function Result Cache:** Results from PL/SQL functions can be cached in the database server.
- **OCI Result Cache:** Results from SQL queries can be cached in the client process cache. Within the scope of the chapter, we will restrict our discussion to the SQL and PL/SQL Result Cache only.

The Result Cache feature can bring huge benefits to warehouse and analytic workloads. Database applications, where the data does not change often but is frequently selected, are the right candidates to implement result caching. Another point to consider is that the server cache is a memory component, albeit a non-persistent one. It gets auto-flushed when the database instance crashes or restarts.

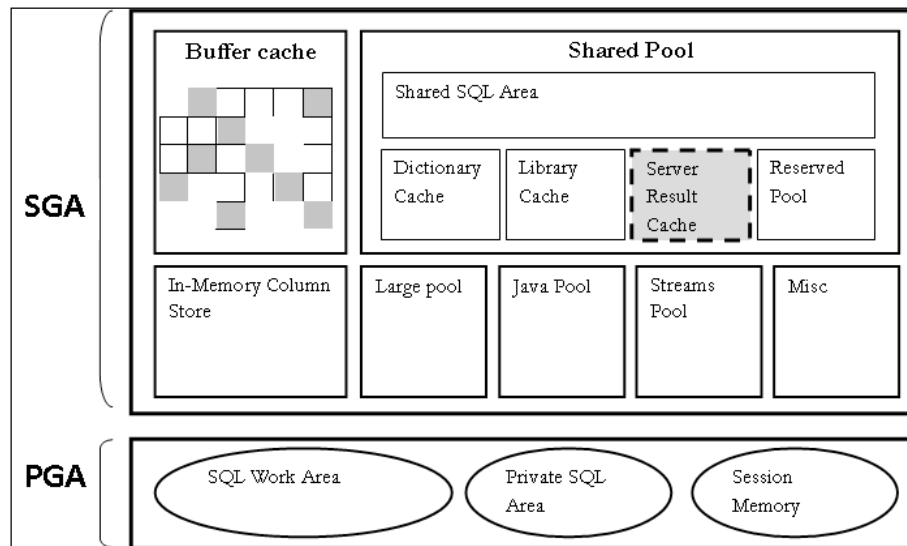
What is the Server Result Cache?

Starting with Oracle Database 11g, the Server Result Cache is a component of database instance memory, that resides within the shared pool of SGA. It is used to store the results of SQL queries and PL/SQL functions. It is further logically divided into two sub-pool components known as SQL query Result Cache and PL/SQL function Result Cache. The SQL query Result Cache stores the results SQL queries and their dependencies. The PL/SQL function Result Cache stores results from PL/SQL functions.

The Server Result Cache is pre-configured to use a small portion of the shared pool. If the database uses Automatic Memory Management (`memory_target`) for memory sizing, then 0.25 percent of `memory_target` is allocated to the Result Cache. In the case of automatic shared memory management (`sga_target`), 0.5 percent of `sga_target` is pre-allocated to the Result Cache. For manual shared memory management (`shared_pool_size`), the Result Cache is pre-allocated with 1 percent of `shared_pool_size`.

You can define your own Server Result Cache size using the initialization parameter `RESULT_CACHE_MAX_SIZE`. The user-supplied value to the parameter is rounded off to the nearest multiple of 32K. If the value of this parameter is zero, the result caching feature is disabled.

You must also note that, being a native part of SGA, the Server Result Cache is affected by **Automatic Memory Management (AMM)**. In addition, it abides by the regular cache features such as dynamic sizing and the **Least Recently Used (LRU)** algorithm to flush out the result sets. The following figure locates the server Result Cache component in the Oracle Database memory architecture:



Oracle Database memory architecture

Configuring the Server Result Cache

In this section, we will learn about the configuration of the server-side result cache feature. There are four initialization parameters that control the result cache feature in the Oracle Database:

- **RESULT_CACHE_MAX_SIZE:** This parameter determines if the caching is enabled or disabled on the database server. If it is zero, the caching feature is disabled. You must set this parameter to a value greater than zero to enable result caching on the server. Actually, it sets the maximum size of the server result cache in the shared pool. Oracle recommends to restrict the Server Result Cache up to 75 percent of the shared pool.

In RAC environments, this parameter must be set at each instance.

- **RESULT_CACHE_MAX_RESULT:** This parameter defines the size of a single result set in the server cache. It is expressed as a percentage value of **RESULT_CACHE_MAX_SIZE**. By default, its value is 5 percent.

- **RESULT_CACHE_MODE**: It determines whether an SQL query has to be served from the SQL query result cache. In other words, the parameter decides if the ResultCache operation has to be added in the query execution plan. The caching mode can be **MANUAL** or **FORCE**:
 - **MANUAL (default)**: The server caches the results of only the annotated queries that are marked with the **RESULT_CACHE** hint.
 - **FORCE**: The database enforces a cache lookup for all the queries and will attempt to store the results for all the queries that are executed on the server. However, you can use the **NO_RESULT_CACHE** hint to skip the cache lookup.



Use **FORCE** mode cautiously as it enforces caching for all SQL queries and this behavior may exhaust the server result cache.

- **RESULT_CACHE_REMOTE_EXPIRATION**: This parameter defines the retention time of a result cached from a remote object. It is expressed in minutes and its value is *zero*, by default.

For example, the following **ALTER SYSTEM** command sets the size of the Server Result Cache:

```
CONNECT sys/oracle as SYSDBA
ALTER SYSTEM SET result_cache_max_size=25M SCOPE=BOTH
/
```

Let's query the **V\$PARAMETER** view to confirm the effect of the preceding command. Note the **ISPDB_MODIFIABLE** column.

```
SELECT name,
       value,
       ispdb_modifiable
FROM v$parameter
WHERE name like 'result%'
/
```

NAME	VALUE	ISPDB
result_cache_mode	MANUAL	TRUE
result_cache_max_size	26214400	FALSE
result_cache_max_result	5	FALSE
result_cache_remote_expiration	0	TRUE

The preceding query output shows that, in a multitenant container database (Oracle Database 12c), the Server Result Cache is elastically shared by all the pluggable databases. The parameters that can be modified from the context of a pluggable database are `RESULT_CACHE_MODE` and `RESULT_CACHE_REMOTE_EXPIRATION`. A pluggable database administrator (PDBA) can either switch on automatic result caching for the database or use the manual mode of caching. In addition, the result set expiration time can vary for each pluggable database in a multitenant container database. Mission-critical databases or those at higher service levels can retain their result sets for longer than low-priority databases.

What happens when the result cache gets full? The least recently used result is automatically flushed from the cache to make room for new results.

Result Cache versus Buffer Cache

The Result cache and buffer cache are part of the database memory architecture, but both are meant to achieve different objectives. The Buffer cache is used to store the data blocks of a table and these blocks are not coupled to an SQL query. The Result cache stores the end results and not the blocks from the SQL queries and thus is tied to a particular SQL query. The Result cache doesn't work on the concepts of blocks or buffers.

Result Cache versus Oracle 12c Database In-Memory

The Server Result Cache stores the results from `SELECT` queries and these results are dependent on the underlying data sources. But Database In-Memory enables the columnization of row-format data and stores it within the In-Memory Column Store of the SGA. The columnar data in the In-Memory column store can be compressed while results in the server cache cannot be compressed.

Result Cache versus In-Memory Database Cache

The **In-Memory Database Cache (IMDB)** is a database option that uses Oracle's Times Ten database under the covers. It is deployed in the application tier to stage business-critical data closer to the application and, hence improve the response time. The application connects to the IMDB cache and performs transactions on the cached data; the changes are posted to the database tier through the synchronous or asynchronous methods. On the other hand, the Server Result Cache feature stores the results of SQL queries and PL/SQL functions, but not the table data. However, it is dependent on the table data and turns stale if the underlying data gets modified.

SQL query Result Cache

You can store the results of a SQL query in the SQL query result cache in three ways:

- **Enabling automatic result cache at the database level:** Automatic SQL result caching is enabled by setting the initialization parameter `RESULT_CACHE_MODE` to `FORCE`, which enforces caching for all the SQL statements. However, you can prevent specific SQL queries from exhausting the Server Result Cache by using the `NO_RESULT_CACHE` hint.
- **Enabling automatic result cache at the table level:** You can set `RESULT_CACHE` mode to `FORCE` for a particular table so that the results from the queries on this table will be cached in the query result cache.

```
ALTER TABLE my_objects RESULT_CACHE (MODE FORCE)
/
```

You can specify a `NO_RESULT_CACHE` hint to avoid the cache lookup operation for a query using the cache enforced table, and even rollback the caching mode to `MANUAL`.

- **Manually annotating selective SQL queries:** You can also specify a `RESULT_CACHE` hint in a frequently executed SQL query to cache its results. This is known as manual result caching. In this case, `RESULT_CACHE_MODE` must be set to `MANUAL`. Queries that scan large volumes of data and return a small result are good candidates for caching.



The `RESULT_CACHE` hint can also be used in sub-queries and in-line views.

How are the results stored in the SQL query result cache? Whenever a query executes, the database performs a cache lookup to search for a result from previous executions of the same query. If the result is found, the cached result is returned without further re-executing the SQL query. If the result is not found, the SQL query is executed as usual and the result is returned to the user. The result is then cached in the query result cache.

How is it different from materialized views? Result Caching and materialized views appear similar in concept as both improve query performance by maintaining a separate copy of a query result, but they are very different in implementation. A materialized view occupies database storage to store the data while the Server Result Cache consumes the instance memory. The data in materialized views gets periodically refreshed by re-executing the base query but the result cache gets automatically built on the subsequent run of the SQL query or PL/SQL function. Materialized view is persistent while the result cache is non-persistent.

A result set in the server cache can be used only if the same query is re-executed (ignoring blank spaces and indentation of the query). A result set is uniquely identified as a combination of dependent tables, predicates, literal inputs, and the result. If the query uses bind variables, Oracle stores a distinct result set for each unique combination of bind variable values.

Let us create a scenario to demonstrate SQL result caching. The following script creates a table from the ALL_OBJECTS dictionary view for our case study:

```
CONNECT scott/tiger
CREATE TABLE my_objects
AS SELECT * FROM all_objects
/
SELECT count (*)
FROM my_objects
/

COUNT(*)
-----
      89507
```

The following SQL gives the count of objects by their object type and status. Notice that the RESULT_CACHE hint has been used to add the ResultCache operator in the execution plan and do a cache lookup. Additionally, we are also adding GATHER_PLAN_STATISTICS to gather additional statistics for the query execution:

```
SET TIMING ON
SELECT /*+RESULT_CACHE GATHER_PLAN_STATISTICS*/
      object_type,
      status,
      count (*)
FROM my_objects
GROUP BY object_type, status
ORDER BY object_type, status
/

Elapsed: 00:00:00.92
```

The preceding query returns 43 rows. Now let us check the explain plan:

```
SELECT *
FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR(FORMAT=>'ALLSTATS LAST'))
/
```

PLAN_TABLE_OUTPUT										
SQL_ID d6p0gxb4qab8, child number 0										
select /*+result_cache gather_plan_statistics*/ object_type, status, count(*) from my_objects group by object_type, status										
Plan hash value: 3257108458										
Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	Buffers	OMem	lMem	Used-Mem
0	SELECT STATEMENT		1		43	00:00:00.33	348			
1	RESULT CACHE	br0p4fgjgh7jp128u0paxavkns	1		43	00:00:00.33	348			
2	HASH GROUP BY		1	54	43	00:00:00.33	348	1116K	1116K	1374K (0)
3	TABLE ACCESS FULL	MY_OBJECTS	1	89507	89507	00:00:00.18	348			
Result Cache Information (identified by operation id):										
1 -										
Note										
- automatic DOP: Computed Degree of Parallelism is 1 because of parallel threshold										
25 rows selected.										

The explain plan shows the RESULT CACHE operation, which confirms the result of the cache lookup operation. Since this is the first execution of the query, the result is cached in the query result cache by the ID 'br0p4fgjgh7jp128u0paxavkns'. The Starts column shows that all operations were carried out at least once in the current execution. The actual rows (A-Rows) show the count of rows scanned during the full table scan and grouped by row count.

Let us re-execute this query and check the explain plan:

```
SELECT /*+RESULT_CACHE GATHER_PLAN_STATISTICS*/
  object_type,
  status,
  count(*)
FROM my_objects
GROUP BY object_type, status
ORDER BY object_type, status
/

Elapsed: 00:00:00.01
```

Note that the second execution is significantly faster than the first execution.
Querying the explain plan:

```
SELECT *
FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR(FORMAT=>'ALLSTATS LAST'))
/
```

PLAN_TABLE_OUTPUT										
SQL_ID d6p0gxb4qab8, child number 0										
select /*+result cache gather plan statistics*/ object_type, status, count(*) from my_objects group by object_type, status										
Plan hash value: 3257108458										
Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	OMem	IMem	Used-Mem	
0	SELECT STATEMENT		1		43	00:00:00.01				
1	RESULT CACHE	br0p4fgjgh7jp128u0paxavkns	1		43	00:00:00.01				
2	HASH GROUP BY		0	54	0	00:00:00.01	1116K	1116K	1374K (0)	
3	TABLE ACCESS FULL	MY_OBJECTS	0	89507	0	00:00:00.01				
Result Cache Information (identified by operation id):										
1 -										
Note										
- automatic DOP: Computed Degree of Parallelism is 1 because of parallel threshold										
25 rows selected.										

The `Starts` column shows that the result is retrieved from the query result cache. It shows zero for the `HASH GROUP BY` and `TABLE ACCESS FULL` operations, which means that these operations were ignored. `A-rows` values are zero as there were no rows scanned in this execution.

Monitoring the SQL Result Cache

Oracle provides dynamic views to query the current state of the result cache. These views are `V$RESULT_CACHE_OBJECTS`, `V$RESULT_CACHE_STATISTICS`, `V$RESULT_CACHE_MEMORY`, and `V$RESULT_CACHE_DEPENDENCY`. Before querying these views, let us check what is in the result cache:

```
/*Monitor the result cache components in shared pool */
SELECT *
FROM v$sghostat
WHERE POOL='shared pool'
and name like 'Result%'
/
```

POOL	NAME	BYTES	CON_ID
shared pool	Result Cache: Memory Mgr	208	0
shared pool	Result Cache: Cache Mgr	256	0
shared pool	Result Cache: State Objs	2928	0
shared pool	Result Cache	163952	0
shared pool	Result Cache: Bloom Fltr	2048	0

From the preceding query output, it is evident that the result cache is a component of the shared pool and contains memory management drivers such as memory manager, cache manager, state objects, and bloom filters.

The RCBG background process coordinates the result cache management activities. The process plays a bigger role in Oracle RAC than in a single instance database:

```
/*Query the RCBG background process */
SELECT paddr, name, description
FROM v$bgprocess
WHERE name='RCBG'
/
```

PADDR	NAME	DESCRIPTION
00	RCBG	Result Cache: Background

Coming back to the cached results, the following query shows the properties of the result set cached in the preceding query:

```
/*Query the result cache objects */
SELECT status,
       type,
       build_time,
       depend_count,
       column_count,
       scan_count,
       row_count
FROM v$result_cache_objects
WHERE cache_id='br0p4fgjgh7jp128u0paxavkns'
/
```

STATUS	TYPE	BUILD_TIME	DEPEND_COUNT	COLUMN_COUNT	SCAN_COUNT
ROW_COUNT					
Published	Result	21	1	3	1
43					

In the preceding output:

- `BUILD_TIME` is the time spent in building the result (in hundredths of a second).
- `DEPEND_COUNT` is the count of dependent objects for this result. In this case, it is just the table (`MY_OBJECTS`).
- `COLUMN_COUNT` is the count of columns in the result.
- `SCAN_COUNT` is the number of times the result is used by an SQL query.
- `ROW_COUNT` is the number of rows contained in the result.
- `STATUS` is the current status of the result; `PUBLISHED`, in this case. Other values of the `STATUS` column, which you might see in different cases, can be:
 - `NEW`: An under-construction cache result
 - `PUBLISHED`: A valid result ready for use
 - `INVALID`: An invalidated and non-usable result
 - `EXPIRED`: A result which has crossed the expiration time
 - `BYPASS`: A bypass result is prevented from being used by a SQL query
- `TYPE` is `RESULT`. The other type can be `DEPENDENCY` for the objects while building up the result.



The `NAMESPACE` column in `V$RESULT_CACHE_OBJECTS` determines whether a result is from a SQL query cache or a PL/SQL function cache. The following SQL query lists the count of results in SQL and PL/SQL result caches in the shared pool:

```
SELECT namespace, count(*)
FROM v$result_cache_objects
WHERE namespace IS NOT NULL
GROUP BY namespace
/
```

The following query lists the result cache statistics. Note that the values against the statistic name match the result cache settings:

```
/*Query the result cache block statistics */
SELECT id, name, value
FROM V$RESULT_CACHE_STATISTICS
/
```

ID	NAME	VALUE
----	------	-------

```

-----
 1 Block Size (Bytes)          1024
 2 Block Count Maximum        25600
 3 Block Count Current         32
 4 Result Size Maximum (Blocks) 1280
 5 Create Count Success        1
 6 Create Count Failure        0
 7 Find Count                  0
 8 Invalidation Count          0
 9 Delete Count Invalid        0
10 Delete Count Valid          0
11 Hash Chain Length           1
12 Find Copy Count             0
13 Latch (Share)               0

```

13 rows selected.

Interpretation: The size of one result cache block in the server cache is 1 kilobyte. Therefore, a maximum of 25,600 blocks can be allowed in the cache and a single result size can contain a maximum of 1,280 blocks (5 percent of 25600). The current result cache has one result and in total 32 blocks are currently allocated.

You can also query these 32 blocks from `V$RESULT_CACHE_MEMORY`. Out of these 32 blocks, only 3 are occupied while the other 29 are free. The math for 3 occupied blocks is quite simple. SQL takes 2 blocks while the dependent table (`MY_OBJECTS`) consumes 1 block:

```

/* Query the count of occupied blocks in result cache memory*/
SELECT free, COUNT (*)
FROM v$result_cache_memory
GROUP BY free
/

FREE  COUNT(*)
-----
NO     3
YES    29

```

You can query the dependency matrix of the result set by querying the `V$RESULT_CACHE_DEPENDENCY` dictionary view:

```

/*Query the dependencies of cached results */
SELECT rc.result_id,
       o.object_name
FROM V$RESULT_CACHE_DEPENDENCY rc, user_objects o

```

```
WHERE rc.object_no = o.object_id
/
```

```
RESULT_ID OBJECT_NAME
-----
1          MY_OBJECTS
```

Invalidation of the SQL Result Cache

For a cached result to stay valid, the state of dependent tables and data must be preserved. If the table is altered or the underlying data is updated, the cached result gets invalidated. It's only after the next execution that the query result is rebuilt and cached as a new result. The following UPDATE statement modifies one row in the MY_OBJECTS table.

```
UPDATE my_objects
SET object_name=initcap (object_name)
WHERE object_id=30
/
COMMIT
/
```

The result in the query result cache gets invalidated:

```
/*Query the cache result status */
SELECT cache_id,status
FROM v$result_cache_objects
WHERE cache_id='br0p4fgjgh7jp128u0paxavkns'
/
```

```
CACHE_ID                                STATUS
-----
br0p4fgjgh7jp128u0paxavkns    Invalid
```

We will re-execute the group by querying the MY_OBJECTS table to rebuild the result in the query result cache:

```
/*Re-execute the SQL query to rebuild the result */
SELECT /*+RESULT_CACHE GATHER_PLAN_STATISTICS*/
  object_type,
  status,
  count(*)
FROM my_objects
GROUP BY object_type, status
```

```

ORDER BY object_type, status
/

/*Query the cache result status */
SELECT cache_id, status
FROM v$result_cache_objects
WHERE cache_id='br0p4fgjgh7jp128u0paxavkns'
/

```

CACHE_ID	STATUS
br0p4fgjgh7jp128u0paxavkns	Published
br0p4fgjgh7jp128u0paxavkns	Invalid

Note that the invalidated result is retained in the query result cache unless it is manually flushed out, ages out, or the database instance is restarted.

Read consistency of the SQL Result Cache

The result cache uses the **System Change Number (SCN)** to maintain the read consistency of SQL queries whose result is retrieved from the SQL query result cache. Each and every result added to the query result cache also stores the SCN until it is valid. A result will stay valid or consistent until an ongoing transaction is committed.

For example, in the preceding code listing, note the change in SCN while the result in the query result cache was invalidated and got rebuilt after the SQL was re-executed:

```

/*Query the SCN numbers for each query result cached */
SELECT cache_id, status, scn
FROM v$result_cache_objects
WHERE cache_id='br0p4fgjgh7jp128u0paxavkns'
/

```

CACHE_ID	STATUS	SCN
br0p4fgjgh7jp128u0paxavkns	Published	7713754
br0p4fgjgh7jp128u0paxavkns	Invalid	7680436

Limitations

The SQL result cache doesn't work with:

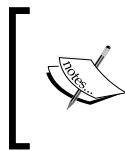
- Temporary tables
- SYS- or SYSTEM-owned objects

- Sequence pseudo columns (CURRVAL and NEXTVAL)
- Date and Time SQL functions
- The SYS_CONTEXT function with a non-constant variable

PL/SQL Function Result Cache

You must create a PL/SQL function with a `RESULT_CACHE` clause to add its result to the PL/SQL function result cache. When a cache-enabled PL/SQL function is invoked for the first time, the database looks into the PL/SQL result cache for its result with the matching arguments. If the result is found, it is returned to the calling environment without executing the function body. If the result is not found, the function body is executed and the result is stored in the PL/SQL function cache. Upon subsequent function calls for the same input parameters, the result is fetched directly from the cache.

Note that a result cache function doesn't need the dependent database tables to be result-cached.



Oracle Database 11g Release 1 used the `RELIES_ON` clause to specify the dependent data sources whose state would affect the status of the cached result. The clause was deprecated in Oracle Database 11g Release 2.

Does it sound similar to deterministic functions?

Developers who are familiar with deterministic functions in PL/SQL might be thinking that the function caching concept is quite close to deterministic behavior. Well, the idea is similar but function result caching comes with many more capabilities. A function is deterministic if it always returns the same output for the same input arguments in a given session only, when invoked from an SQL query. The PL/SQL function result cache overcomes the limitations of a deterministic function by being sharable across sessions of the same user and can be invoked from SQL as well as PL/SQL.

Differences between Result Cache and other caching techniques

Yes, that's a good point. Prior to the introduction of this feature in Oracle Database 11g, caching was implemented using package level collection variables. The two major drawbacks of that approach were that it served as a session level cache and consumed a substantial amount of **Process Global Area (PGA)**. Being a session-specific cache, the packaged variables were not sharable with other sessions. PGA usage can grow enormously for large collection variables, which may impact performance. The Server result cache wins by a wide margin on these points. It is sharable across a user's sessions, preserves data integrity, and most importantly, it uses the Oracle-managed SGA instead of throttling the PGA.

Illustration

We will create a PL/SQL function `F_COUNT_OBJ` with the `RESULT_CACHE` clause. The function returns the count of objects for a specific object type:

```
/*A function with the RESULT_CACHE clause */
CREATE OR REPLACE FUNCTION f_count_obj (obj_type VARCHAR2)
RETURN NUMBER RESULT_CACHE IS
  l_count NUMBER;
BEGIN
  /*Select query to fetch record count in a local variable */
  SELECT COUNT (object_type)
  INTO l_count
  FROM my_objects
  WHERE object_type = obj_type;

  RETURN l_count;
END;
/
```

We will invoke this function in a `SELECT` statement and check the statistics:

```
SET AUTOTRACE ON
/*Invoke function in SELECT statement for a valid input */
SELECT f_count_obj ('PROCEDURE')
FROM DUAL
/

M_PROC_COUNT
-----
      214
```

```
Statistics
-----
      2 recursive calls
      0 db block gets
    348 consistent gets
      0 physical reads
      0 redo size
    559 bytes sent via SQL*Net to client
    551 bytes received via SQL*Net from client
      2 SQL*Net roundtrips to/from client
      0 sorts (memory)
      0 sorts (disk)
      1 rows processed
```

Note the consistent gets in the preceding statistics. This is for the execution of the SELECT statement in the function body. Now let's run the function for the second time:

```
/*Invoke function in SELECT statement again for the same input */
SELECT f_count_obj ('PROCEDURE')
FROM DUAL
/
```

```
M_PROC_COUNT
```

```
-----
      214
```

```
Statistics
-----
      0 recursive calls
      0 db block gets
      0 consistent gets
      0 physical reads
      0 redo size
    559 bytes sent via SQL*Net to client
    551 bytes received via SQL*Net from client
      2 SQL*Net roundtrips to/from client
      0 sorts (memory)
      0 sorts (disk)
      1 rows processed
```

There were absolutely no consistent gets this time as the result is fetched directly from the PL/SQL cache. Now let's run the F_COUNT_OBJ function for a different input value and check the statistics:

```
/*Invoke function in SELECT statement for another input */
SELECT f_count_obj ('FUNCTION')
FROM DUAL
```

```

/

F_COUNT_OBJ('FUNCTION')
-----
                356

Elapsed: 00:00:00.09

Statistics
-----
      4 recursive calls
      0 db block gets
     348 consistent gets
      0 physical reads
      0 redo size
     558 bytes sent via SQL*Net to client
     551 bytes received via SQL*Net from client
      2 SQL*Net roundtrips to/from client
      0 sorts (memory)
      0 sorts (disk)
      1 rows processed

```

Once again, consistent gets appear in the statistics because a different input is supplied so the function body is executed and a new result set is added to the cache.

Monitoring the PL/SQL Result Cache

For monitoring, we can query the dynamic dictionary views and confirm the caching of results from the PL/SQL function. The following query lists the results present in the server cache:

```

SELECT cache_id, name, status, depend_count, scan_count
FROM v$result_cache_objects, dba_users
WHERE creator_uid=user_id
AND username='SCOTT'
/

```

CACHE_ID	NAME	STATUS	DEPEND_COUNT	SCAN_COUNT
SCOTT.MY_OBJECTS	SCOTT.MY_OBJECTS	Published	2	0
SCOTT.F_COUNT_OBJ	SCOTT.F_COUNT_OBJ	Published	2	0
0r7x7k9yu8n2d9x5qd94tk3p07	"SCOTT"."F_COUNT_OBJ"::8."F_COUNT_OBJ"#32fb3b6bdac49c05 #1	Published	2	0
0r7x7k9yu8n2d9x5qd94tk3p07	"SCOTT"."F_COUNT_OBJ"::8."F_COUNT_OBJ"#32fb3b6bdac49c05 #1	Published	2	0

From the preceding output, it is evident that there are two result sets in the cache (that is to say, for two different inputs). Also there are two dependent objects—the F_COUNT_OBJ function and the MY_OBJECTS table. Any change in the state of these two objects will result in the invalidation of the cached results.

Another point of interest is the cache ID. Yes, both the results have the same cache ID. In the case of the PL/SQL function cache, all the results have the same cache ID but different cache keys.

The following query retrieves two PL/SQL result cache statistics that determine how many results were created in the cache (Create Count Success) and how many times they were served (Find Count):

```
SELECT *
FROM v$result_cache_statistics
WHERE NAME IN ('Create Count Success','Find Count')
/
```

ID	NAME	VALUE	CON_ID
5	Create Count Success	2	0
7	Find Count	2	0

Invalidation of the PL/SQL Result Cache

The function result cache gets invalidated when:

- The underlying data sources are altered or the data is modified
- The function is recompiled

In the next invocation of the function, the result is rebuilt and cached in the PL/SQL function result cache. Note that the invalidated result set will still exist and gets flushed off by a manual flush operation or during an instance restart. By virtue of it being a cache, the results can also get flushed off by the least recently used (LRU) algorithm.

Limitation

The PL/SQL function result cache does not work for:

- Pipelined functions
- OUT and IN OUT parameters
- The IN parameter of BLOB, CLOB, NCLOB, ref cursor, collection, object, or record

- BLOB, CLOB, NCLOB returns, ref cursor, collection, object, or record
- A function declared locally in an anonymous PL/SQL block

OCI Client results cache

Applications with the **Oracle Call Interface (OCI)** client can largely benefit from the result cache feature. The results from an SQL query and a PL/SQL function can be cached within the OCI client process cache and not in the database server memory. Caching results in the client process cache helps in multiple ways. First, the query response time improves drastically as the results are served directly from the cache without even hitting the database server. Second, the reduced network roundtrips enhance application performance and the effective utilization of database resources.

How is the consistency of results in the client process cache maintained? The Oracle Database server is responsible for monitoring the consistency of cached results. If the underlying data is modified, the database server notifies the client process and invalidates the result set. On its next roundtrip to the server, the client process rebuilds the stale results.

The client result cache is independent of the database server cache and deduces that a result from an SQL query can be cached on the database server cache, the client process cache, or both.

To set up the client result cache, you are required to set the following parameters:

- `CLIENT_RESULT_CACHE`: This determines the maximum size of the client process cache, which must be more than 32KB.
- `CLIENT_RESULT_CACHE_LAG`: This specifies the time (in seconds) after which the OCI call to the database is forced to validate the cached results. The default value is 3,000 seconds.
- `COMPATIBLE`: This is the Oracle Database-compatible version.

The DBMS_RESULT_CACHE package

The Oracle-supplied package `DBMS_RESULT_CACHE` is used to regulate the Server Result Cache component of the shared pool. The package is owned by `SYS` and only privileged users should be granted the `EXECUTE` privilege.

The public constants used in the package are as follows:

DBMS_RESULT_CACHE constants (reference: Oracle documentation)	
STATUS_BYPS	CONSTANT VARCHAR(10) := 'BYPASS';
STATUS_DISA	CONSTANT VARCHAR(10) := 'DISABLED';
STATUS_ENAB	CONSTANT VARCHAR(10) := 'ENABLED';
STATUS_SYNC	CONSTANT VARCHAR(10) := 'SYNC';
STATUS_CORR	CONSTANT VARCHAR(10) := 'CORRUPT';

The subprograms used in the package are described in the following table:

DBMS_RESULT_CACHE subprograms (reference: Oracle documentation)	
BYPASS procedure	<ul style="list-style-type: none">• If set to TRUE, the result cache usage is bypassed in the current or every session.• Used when modifying and recompiling a PL/SQL function whose results have been cached earlier.• In RAC, since each instance has its own server cache, the BYPASS procedure must be run in all instances.
FLUSH function and procedure	<ul style="list-style-type: none">• Flushes out all the results from the server cache.• You can also retain or release memory and statistics. Default action is release.
INVALIDATE functions and procedures	<ul style="list-style-type: none">• Invalidates all the results that are dependent on a given object.• Returns the number of results invalidated.
INVALIDATE_OBJECT functions and procedures	<ul style="list-style-type: none">• Invalidates the specified result set in the server cache.
MEMORY_REPORT procedure	<ul style="list-style-type: none">• Generates the summary report for the server result cache.• Specify (detailed => TRUE) to generate a detailed version of the report.
STATUS function	<ul style="list-style-type: none">• Checks the status of the result cache.

Displaying the result cache memory report

You can generate a server cache memory report via the `DBMS_RESULT_CACHE` package. A sample cache memory report (summarizing results from our illustrations) looks as follows:

```
CONNECT sys/oracle as sysdba
SET SERVEROUTPUT ON

/*Generate the cache memory report*/
EXEC DBMS_RESULT_CACHE.MEMORY_REPORT
Result Cache Memory Report
[Parameters]
Block Size          = 1K bytes
Maximum Cache Size  = 25M bytes (25K blocks)
Maximum Result Size = 1280K bytes (1280 blocks)
[Memory]
Total Memory = 169392 bytes [0.044% of the Shared Pool]
... Fixed Memory = 5440 bytes [0.001% of the Shared Pool]
... Dynamic Memory = 163952 bytes [0.042% of the Shared Pool]
..... Overhead = 131184 bytes
..... Cache Memory = 32K bytes (32 blocks)
..... Unused Memory = 26 blocks
..... Used Memory = 6 blocks
..... Dependencies = 2 blocks (2 count)
..... Results = 4 blocks
..... SQL          = 2 blocks (1 count)
..... PLSQL        = 2 blocks (2 count)

PL/SQL procedure successfully completed.
```

The preceding report lists the maximum cache size, maximum result size, and standard block size. The report can be interpreted as follows:

- The result cache is allocated from the dynamic section of the shared pool. Currently, only 169,392 bytes are allocated and the maximum stretch is up to 25MB.
- Fixed memory is the memory consumed by Memory Mgr, Cache Mgr, State Objs and Bloom Fltr.
- Out of 32K blocks, only 6 blocks are used. The 6 used blocks include 2 for dependencies (1 each by `F_COUNT_OBJ` and `MY_OBJECTS`) and 4 by sub-pools (2 by the SQL query result cache and 2 by the PL/SQL function result cache—2 function calls).

You can also generate a detailed report by specifying the DETAILED parameter as TRUE in the MEMORY_REPORT procedure:

```
exec dbms_result_cache.memory_report (TRUE);
R e s u l t   C a c h e   M e m o r y   R e p o r t
[Parameters]
Block Size           = 1K bytes
Maximum Cache Size   = 25M bytes (25K blocks)
Maximum Result Size  = 1280K bytes (1280 blocks)
[Memory]
Total Memory = 169392 bytes [0.044% of the Shared Pool]
... Fixed Memory = 5440 bytes [0.001% of the Shared Pool]
..... Memory Mgr = 208 bytes
..... Cache Mgr   = 256 bytes
..... Bloom Fltr  = 2K bytes
..... State Objs  = 2928 bytes
... Dynamic Memory = 163952 bytes [0.042% of the Shared Pool]
..... Overhead = 131184 bytes
..... Hash Table   = 64K bytes (4K buckets)
..... Chunk Ptrs   = 24K bytes (3K slots)
..... Chunk Maps   = 12K bytes
..... Miscellaneous = 131184 bytes
..... Cache Memory = 32K bytes (32 blocks)
..... Unused Memory = 26 blocks
..... Used Memory  = 6 blocks
..... Dependencies = 2 blocks (2 count)
..... Results      = 4 blocks
..... SQL          = 2 blocks (1 count)
..... PLSQL        = 2 blocks (2 count)

PL/SQL procedure successfully completed.
```

Oracle Database 12c enhancements to the PL/SQL function Result Cache

Until Oracle Database 11g, results from a PL/SQL function created with invoker's rights (that is, the AUTHID CURRENT_USER clause) cannot be result-cached. The restriction was imposed from a security standpoint as the database users can have different access levels which may contradict the result caching mechanism. For instance, a user FIN executes a PL/SQL function to retrieve salary details of a department and adds the result to the cache. Another user CLERK invokes the same function and gets the results from the PL/SQL function result cache. Had it been an invoker's rights function, he would have received an exception.

Starting from Oracle Database 12c, the invoker's rights function can be result cached. The idea is quite simple. While performing a cache lookup, Oracle passes the current user along with the function signature. While adding a result to the cache, the user information is also cached. Next time, whenever the function is invoked, the database verifies whether the current user has invoked this function earlier. So now a result is tightly coupled with the user who has added it in the cache. Consequently, a result is sharable across the sessions of the same user.

For example, the following PL/SQL function is an invoker's rights function with the result cache feature:

```
/*Create an invoker right result cache function */
CREATE OR REPLACE FUNCTION f_count_myobj (obj_type VARCHAR2)
RETURN NUMBER
AUTHID CURRENT_USER
RESULT_CACHE IS
    l_count NUMBER;
BEGIN

    /*Capture the record count in a local variable */
    SELECT COUNT (object_type)
    INTO l_count
    FROM my_objects
    WHERE object_type = obj_type;

    RETURN l_count;
END;
/
```

Function created.

Result cache in Real Application Clusters

The Oracle Database **Real Application Cluster (RAC)** option allows the sharing of a cluster database across multiple database instances. It enhances the scalability and availability of database applications.

The Result cache, being the native component of database instance memory, is supported on RAC environments. The behavior and handling of result caching on RAC is the same as that of a single instance database. Here are the salient features of result caching in RAC:

- Each RAC instance maintains a local version of the Server Result Cache in the shared pool of the SGA
- A result is added to the instance result cache when a user executes a SQL query or a PL/SQL function on a database instance
- If the same user executes the same query from another instance, the cached result is copied to that instance's result cache over RAC interconnect
- The dynamic view `GV$RESULT_CACHE_OBJECTS` will show duplicate entries with the same cache ID, but for different instances (`INST_ID`)
- If the table data is modified or the table is altered, the result is invalidated in the Server Result Cache on all the instances of the cluster
- Whichever instance re-executes the query, the result is rebuilt and cached for that instance only

Summary

In this chapter, we have explored the caching mechanism in the Oracle Database server. We saw how server result caching can dramatically improve performance in SQL and PL/SQL applications. We learned how to configure the Server Result Cache, how to enable manual and automatic SQL query result caches, and how to work with the PL/SQL function result cache.

In the next chapter, we will learn about PL/SQL code profiling and tracing techniques.

Practice exercise

- The initialization parameter settings for your database are as follows:

```
MEMORY_TARGET = 500M
RESULT_CACHE_MODE = MANUAL
RESULT_CACHE_MAX_SIZE = 0
```

You execute a query by using the `RESULT_CACHE` hint. Which statement is true in this scenario?

- The query results are not stored in the cache because no memory is allocated for the result cache.
 - The query results are stored in the cache because Oracle implicitly manages the cache memory.
 - The query results are not stored in the cache because `RESULT_CACHE_MODE` is `MANUAL`.
 - The query results are stored in the cache automatically when `RESULT_CACHE_MODE` is `MANUAL`.
- You set the following initialization parameter settings for your database:

```
MEMORY_TARGET = 500M
RESULT_CACHE_MODE = FORCE
RESULT_CACHE_MAX_SIZE = 200M
```

You execute the following query:

```
SELECT /*+RESULT_CACHE*/ ENAME, DEPTNO
FROM EMPLOYEES
WHERE EMPNO = 7844
/
```

Which of the following statements are true?

- The query results are cached because SQL uses the `RESULT_CACHE` hint.
 - The query results are cached because the result cache mode is `FORCE`.
 - The query results are not cached because SQL uses the `RESULT_CACHE` hint.
 - The `RESULT_CACHE` hint is ignored when the result cache mode is `FORCE`.
- The cached query result becomes invalid when the data accessed by the query gets modified.
 - True.
 - False.

- The SQL query result cache is persistent only for the current session.
 1. True.
 2. False.
- Which of the following PL/SQL object results cannot be cached?
 1. A Standalone function.
 2. A Procedure.
 3. A function local to a procedure.
 4. A Packaged function.
- The `RELIES_ON` clause in the PL/SQL function result cache can be used to specify the dependent tables or views whose state would affect the cached result.
 1. True.
 2. False.
- Server settings are as follows:

```
MEMORY_TARGET = 500M
RESULT_CACHE_MODE = FORCE
RESULT_CACHE_MAX_SIZE = 200M
```

Identify the SQL queries whose results cannot be cached by the server.

1. `SELECT ename, sal FROM employees WHERE empno = 7900;`
 2. `SELECT seq_empid.nextval FROM DUAL;`
 3. `SELECT ename, sysdate, hiredate FROM employees;`
 4. `SELECT dname, loc FROM departments WHERE deptno = 10;`
- Identify the correct statements about the PL/SQL function result cache.
 1. The PL/SQL function result cache requires additional server configuration.
 2. The PL/SQL function result cache cannot be operated on procedures.
 3. The PL/SQL function result cache works with all categories of functions.
 4. The PL/SQL function result cache features can work with functions that take collection type arguments.

- Identify the admissible value of the `STATUS` column in `V$RESULT_CACHE_OBJECTS`.

- PUBLISHED.
- INVALID.
- USED.
- UNUSED.

- Choose the correct statement about the following sample cache memory statistics report:

ID	NAME	VALUE

1	Block Size (Bytes)	1024
2	Block Count Maximum	204800
3	Block Count Current	32
4	Result Size Maximum (Blocks)	40960
5	Create Count Success	1
6	Create Count Failure	0
7	Find Count	0
8	Invalidation Count	0
9	Delete Count Invalid	0
10	Delete Count Valid	0
11	Hash Chain Length	1

- Create Count Success is the count of successfully cached results.
- Find Count is the count of the successfully cached results found and used in the queries.
- Invalidation Count is the count of the invalidated cached results.
- Block Count Maximum is the static value of total blocks available in the cache memory.

10

Analyzing, Profiling, and Tracing PL/SQL Code

During the database development stage and even after, the developers are required to analyze and maintain the database objects. Analyzing the PL/SQL code is an essential exercise that enables you to draw out key information about a program. PL/SQL code analysis can help the developers in: tracking object dependencies, unused variables, retrieving compilation settings, tracking program execution flow, and building the performance profile of an object.

Oracle provides a powerful set of metadata sources, known as dictionary views, to reveal the metadata of PL/SQL objects. For all the objects that are created, modified, or compiled in a database, Oracle captures the metadata and continues to update it at each action. This chapter will focus on how to analyze a PL/SQL code unit, how to trace the program execution, and how to profile it in a very simplistic way. In this chapter, we will learn techniques to:

- Analyze PL/SQL metadata information through dictionary views
- Trace PL/SQL program execution flow
- Profile PL/SQL code for performance

A sample PL/SQL program

Before we plunge into code analysis techniques, let us write down a standard PL/SQL program for demonstrating code analysis, profiling, and tracing. The following PL/SQL procedure calculates the score of a user in an exam (Note that negative scoring is applicable). The procedure `P_CALC_USER_POINTS` declares a local function and a procedure to calculate the points:

```
/*Create a PL/SQL procedure*/  
CREATE OR REPLACE PROCEDURE p_calc_user_points
```

```
(p_user VARCHAR2 DEFAULT USER, p_correct NUMBER, p_wrong NUMBER)
IS
  l_num NUMBER;

/*A local function F_CALC_POINTS */
FUNCTION f_calc_points (p_ques NUMBER, p_factor NUMBER)
RETURN NUMBER
IS
BEGIN
  RETURN (p_ques*p_factor);
END;

/*A local procedure */
PROCEDURE P_NET_CALC (p_net_points OUT NUMBER) IS
BEGIN
  p_net_points := f_calc_points (p_correct,4) + f_calc_points (p_
wrong,-2);
END;

/*Main procedure body */
BEGIN
  p_net_calc (l_num);
  DBMS_OUTPUT.PUT_LINE (USER||' earned '||TO_CHAR (l_num)||' points');
END;
/
```

We will now query the Oracle-supplied dictionary views to find ways to track PL/SQL program properties such as argument details, identifier usage, and object compilation settings.

Tracking PL/SQL coding information

The Oracle-supplied dictionary views are a great source of information for performing drill-down analysis of PL/SQL code. Although there are several dictionaries that store PL/SQL object information, the important ones are `USER_ARGUMENTS`, `USER_OBJECTS`, `USER_SOURCE`, `USER_PROCEDURES`, and `USER_DEPENDENCIES`. These views also have their `ALL_*` and `DBA_*` counterparts. For your reference, the `USER`, `ALL`, and `DBA` category views are described as follows:

- `USER`: Contains only the objects that are owned by a user
- `ALL`: Contains the objects that can be accessed by a user
- `DBA`: Contains all the objects accessible by the `SYS` user or a user with `DBA` privileges

You can query the data dictionary views from the `DICTIONARY` view and their column structure from `DICT_COLUMNS` view. Let us query the metadata information of the procedure `P_CALC_USER_POINTS` in these dictionary views.

USER_ARGUMENTS

The `USER_ARGUMENTS` view captures the argument information of a PL/SQL program. The argument information includes the position of an argument in the parameter list, its data type, defaults, and pass mode.



Usage of a `NOCOPY` hint in an `OUT` or `IN OUT` parameter is not tracked in `USER_ARGUMENTS`

The following query lists `argument_name`, its position, pass mode, `data_type`, and default.

```
/*Query the arguments of the procedure P_CALC_USER_POINTS*/
SELECT argument_name, data_type, defaulted, position, data_level, in_
out, pls_type
FROM user_arguments
WHERE object_name IN ('P_CALC_USER_POINTS')
ORDER BY position
/
```

ARGUMENT_NAME	DATA_TYPE	D	POSITION	DATA_LEVEL	IN_OUT	PLS_TYPE
P_USER	VARCHAR2	Y	1	0	IN	VARCHAR2
P_CORRECT	NUMBER	N	2	0	IN	NUMBER
P_WRONG	NUMBER	N	3	0	IN	NUMBER

In the preceding query result, the `DATA_LEVEL` represents the level of nesting in a composite data type. Since the procedure had scalar data types only, `DATA_LEVEL` is zero.

USER_OBJECTS

The `USER_OBJECTS` dictionary view is the primary source to check the validity of a PL/SQL object. The view also includes relevant details such as creation and last update timestamps, namespace, and edition.

```
/*Query the object properties of P_CALC_USER_POINTS */
SELECT object_id, object_type, status, namespace
FROM user_objects
```

```
WHERE object_name='P_CALC_USER_POINTS'
/
```

OBJECT_ID	OBJECT_TYPE	STATUS	NAMESPACE
81410	PROCEDURE	VALID	1

The `ORACLE_MAINTAINED` column of the dictionary view determines if a given object is managed by the Oracle Database or the user. All the Oracle-supplied packages (such as `STANDARD`, `DBMS_OUTPUT` and all others) are maintained implicitly by Oracle and can be queried from `ALL_OBJECTS`.

USER_OBJECT_SIZE

The `USER_OBJECT_SIZE` view provides the size (in bytes) of the source code, parsed code and compiled code of a PL/SQL object. The size information can be quite relevant, if the program invocation results in out-of-memory issues. Database developers can determine if a large program can be broken down into smaller subprograms or multiple modules can be clubbed under a single PL/SQL unit.

The following query shows the size of the `P_CALC_USER_POINTS` procedure:

```
/*Query the code size of procedure P_CALC_USER_POINTS */
SELECT type, source_size, parsed_size, code_size, error_size
FROM user_object_size
WHERE name='P_CALC_USER_POINTS'
/
```

TYPE	SOURCE_SIZE	PARSED_SIZE	CODE_SIZE	ERROR_SIZE
PROCEDURE	472	1096	827	0

In the preceding output, `ERROR_SIZE` is the size of the error messages raised during code compilation.

USER_SOURCE

The `USER_SOURCE` dictionary view provides the text source code of a PL/SQL object. This view can be handy in building up a text-based code search engine.

The following query shows the program body of the procedure `P_CALC_USER_POINTS`:

```
/*Query the source code of P_CALC_USER_POINTS*/
SELECT line, text
```

```

FROM user_source
WHERE name='P_CALC_USER_POINTS'
ORDER BY line
/

```

```

LINE TEXT
-----
1  PROCEDURE p_calc_user_points
2  (p_user VARCHAR2 DEFAULT USER, p_correct NUMBER, p_wrong NUMBER)
3  IS
4  l_num NUMBER;
5  FUNCTION f_calc_points (p_ques NUMBER, p_factor NUMBER)
6  RETURN NUMBER
7  IS
8  BEGIN
9    RETURN (p_ques*p_factor);
10 END;
11
12 PROCEDURE P_NET_CALC (p_net_points OUT NUMBER) IS
13 BEGIN
14   p_net_points := f_calc_points (p_correct,4) + f_calc_points
(p_wrong,-2);
15 END;
16
17 BEGIN
18   p_net_calc (l_num);
19   DBMS_OUTPUT.PUT_LINE (USER||' earned '||TO_CHAR (l_num)||'
points');
20 END;

```

20 rows selected.

The following SELECT query finds the objects that have customized exception handling through RAISE_APPLICATION_ERROR:

```

/*Query to build text based code search */
SELECT DISTINCT owner, name
FROM all_source
WHERE UPPER(text) LIKE '%RAISE_APPLICATION_ERROR%'
/

```

OWNER	NAME
SYS	UTL_SMTP
SYS	DBMS_STANDARD


```
SYS          UTL_HTTP
APEX_040200   WWV_FLOW_ERROR_API
APEX_040200   WWV_FLOW_ESCAPE
```

5 rows selected.

USER_PROCEEDURES

The `USER_PROCEEDURES` dictionary view captures the subprogram properties of an object. Contrary to its name (which just says "procedures"), it displays the behavioral aspects such as aggregate, pipelined, parallel, deterministic, or privilege authentication of not just procedures, but also of PL/SQL functions and PL/SQL packages.

The procedural properties of the `P_CALC_USER_POINTS` subprogram can be queried from the view as follows:

```
/*Query the subprogram properties of P_CALC_USER_POINTS*/
SELECT object_type,
       aggregate,
       pipelined,
       parallel,
       interface,
       deterministic,
       authid
FROM   user_procedures
WHERE  object_name='P_CALC_USER_POINTS'
/
```

```
OBJECT_TYPE  AGG PIP PAR INT DET AUTHID
-----
PROCEDURE    NO  NO  NO  NO  NO  DEFINER
```

USER_PLSQL_OBJECT_SETTINGS and USER_STORED_SETTINGS

The `USER_PLSQL_OBJECT_SETTINGS` and `USER_STORED_SETTINGS` views are used to display the compilation parameter values of a PL/SQL object. The difference between the two views is that the `USER_PLSQL_OBJECT_SETTINGS` view contains a column for each compilation parameter, while the `USER_STORED_SETTINGS` view stores parameter values as a key-value pair (the parameter name as the key) for a PL/SQL program.

Let's check the value of compilation parameters from both of the views. The following SELECT pivot query shows the compilation parameter values for the procedure P_CALC_USER_POINTS:

```
/*Query the compilation parameters for P_CALC_USER_POINTS */
SELECT name, pname, pval
FROM user_plsql_object_settings
UNPIVOT INCLUDE NULLS
(
  pval FOR (pname) IN
  (
    PLSQL_CODE_TYPE as 'PLSQL_CODE_TYPE',
    PLSQL_DEBUG as 'PLSQL_DEBUG',
    PLSQL_WARNINGS as 'PLSQL_WARNINGS',
    NLS_LENGTH_SEMANTICS as 'NLS_LENGTH_SEMANTICS',
    PLSQL_CCFLAGS as 'PLSQL_CCFLAGS',
    PLSCOPE_SETTINGS as 'PLSCOPE_SETTINGS'
  )
)
WHERE name='P_CALC_USER_POINTS'
/
```

NAME	PNAME	PVAL
P_CALC_USER_POINTS	PLSQL_CODE_TYP	NATIVE
P_CALC_USER_POINTS	PLSQL_DEBUG	FALSE
P_CALC_USER_POINTS	PLSQL_WARNINGS	DISABLE:ALL
P_CALC_USER_POINTS	NLS_LENGTH_SEMANTICS	BYTE
P_CALC_USER_POINTS	PLSQL_CCFLAGS	
P_CALC_USER_POINTS	PLSCOPE_SETTINGS	IDENTIFIERS:ALL

6 rows selected.

```
/*Query the compilation parameters for P_CALC_USER_POINTS */
SELECT object_name, param_name, param_value
FROM user_stored_settings
WHERE object_name = 'P_CALC_USER_POINTS'
/
```

OBJECT_NAME	PARAM_NAME	PARAM_VALUE
P_CALC_USER_POINTS	plsql_optimize_level	2
P_CALC_USER_POINTS	plsql_code_type	NATIVE
P_CALC_USER_POINTS	plsql_debug	FALSE

P_CALC_USER_POINTS	nls_length_semantics	BYTE
P_CALC_USER_POINTS	plsql_warnings	DISABLE:ALL
P_CALC_USER_POINTS	plsql_ccflags	
P_CALC_USER_POINTS	plscope_settings	IDENTIFIERS:ALL
P_CALC_USER_POINTS	plsql_compiler_flags	NATIVE,NON_DEBUG

8 rows selected.

USER_DEPENDENCIES

The USER_DEPENDENCIES dictionary view gives you the list of objects on which a given PL/SQL subprogram is dependent. If any of these dependent objects gets invalidated, the PL/SQL program will also be invalidated. The following query lists the objects that are referenced in the P_CALC_USER_POINTS function:

```
/*Query the dependent objects of P_CALC_USER_POINTS */
SELECT referenced_owner,
       referenced_name rname,
       referenced_type rtype,
       dependency_type dtype
FROM user_dependencies
WHERE name='P_CALC_USER_POINTS'
/
```

OWNER	RNAME	RTYPE	DTYP
-----	-----	-----	-----
SYS	STANDARD	PACKAGE	HARD
SYS	SYS_STUB_FOR_PURITY_ANALYSIS	PACKAGE	HARD
PUBLIC	DBMS_OUTPUT	SYNONYM	HARD

The DBMS_DESCRIBE package

The DBMS_DESCRIBE package is an Oracle supplied package that is used to describe the structure of a PL/SQL object. In terms of functionality, the DBMS_DESCRIBE package gives information that is similar to USER_ARGUMENTS or USER_PROCEDURES, but it is used where the program structure has to be exposed based on a client request. It is owned by SYS and its public synonym is available to all users.

The DBMS_DESCRIBE package contains one subprogram DESCRIBE_PROCEDURE and uses associative arrays to capture the PL/SQL object structure.

```
TYPE VARCHAR2_TABLE IS TABLE OF VARCHAR2(30)
INDEX BY BINARY_INTEGER;
TYPE NUMBER_TABLE IS TABLE OF NUMBER
INDEX BY BINARY_INTEGER;
```

The following PL/SQL block retrieves the signature of the PL/SQL procedure

P_CALC_USER_POINTS:

```

SET SERVEROUTPUT ON
DECLARE

/*Declare the local variables of associative array type*/

    v_overload  DBMS_DESCRIBE.NUMBER_TABLE;
    v_position  DBMS_DESCRIBE.NUMBER_TABLE;
    v_level     DBMS_DESCRIBE.NUMBER_TABLE;
    v_arg_name  DBMS_DESCRIBE.VARCHAR2_TABLE;
    v_datatype  DBMS_DESCRIBE.NUMBER_TABLE;
    v_def_value DBMS_DESCRIBE.NUMBER_TABLE;
    v_in_out    DBMS_DESCRIBE.NUMBER_TABLE;
    v_length    DBMS_DESCRIBE.NUMBER_TABLE;
    v_precision DBMS_DESCRIBE.NUMBER_TABLE;
    v_scale     DBMS_DESCRIBE.NUMBER_TABLE;
    v_radix     DBMS_DESCRIBE.NUMBER_TABLE;
    v_spare     DBMS_DESCRIBE.NUMBER_TABLE;
BEGIN

/*Call the procedure DESCRIBE_PROCEDURE for P_CALC_USER_POINTS */
    DBMS_DESCRIBE.DESCRIBE_PROCEDURE
    (object_name=> 'P_CALC_USER_POINTS',
      reserved1 => null,
      reserved2 => null,
      overload  => v_overload,
      position  => v_position,
      level     => v_level,
      argument_name => v_arg_name,datatype => v_datatype,
      default_value => v_def_value,
      in_out     => v_in_out,
      length     => v_length,
      precision  => v_precision,
      scale      => v_scale,
      radix      => v_radix,
      spare      => v_spare,
      include_string_constraints => null);

    FOR i IN v_arg_name.FIRST .. v_arg_name.LAST
    LOOP
        DBMS_OUTPUT.PUT_LINE (RPAD('_',40, '_'));
    
```

```
/*Check if the position if zero or not*/
  IF v_position(i) = 0 THEN

/*Zero position is reserved for RETURN types*/
    DBMS_OUTPUT.PUT_LINE (' RETURN type of the function: ');
    DBMS_OUTPUT.PUT_LINE (rpad('Function Return type:',30,' ')||v_
datatype(i));
    ELSE

/*Print the argument name*/
    DBMS_OUTPUT.PUT_LINE (RPAD('Parameter name:',30,' ')||v_arg_
name(i));
    END IF;

/*Display the position, type and mode of parameters*/
    DBMS_OUTPUT.PUT_LINE(rpad('Parameter position:',30,' ')||
v_position(i));
    DBMS_OUTPUT.PUT_LINE(rpad('Parameter data type:',30,' ')||
case v_datatype(i)
  when 1 then 'VARCHAR2'
  when 2 then 'NUMBER'
  when 3 then 'BINARY_INTEGER'
  when 8 then 'LONG'
  when 11 then 'ROWID'
  when 12 then 'DATE'
  when 23 then 'RAW'
  when 24 then 'LONG RAW'
  when 58 then 'OPAQUE TYPE'
  when 96 then 'CHAR'
  when 106 then 'LONG RAW'
  when 121 then 'OBJECT'
  when 122 then 'NESTED TABLE'
  when 123 then 'VARRAY'
  when 178 then 'TIME'
  when 179 then 'TIME WITH TIME ZONE'
  when 180 then 'TIMESTAMP'
  when 230 then 'PL/SQL RECORD'
  when 251 then 'PL/SQL TABLE'
  when 252 then 'PL/SQL BOLLEAN'
end);
    DBMS_OUTPUT.PUT_LINE(rpad('Parameter default:',30,' ')||
case v_def_value(i)
  when 0 then 'No Default'
  when 1 then 'Defaulted'
```

```

end);

DBMS_OUTPUT.PUT_LINE(rpad('Parameter pass mode:',30,' ')||
                      case v_in_out(i)
                        when 0 then 'IN mode'
                        when 1 then 'OUT mode'
                        when 2 then 'IN OUT mode'
                      end);
DBMS_OUTPUT.PUT_LINE (rpad('_',40,'_'));
END LOOP;
END;
/

```

Parameter name:	P_USER
Parameter position:	1
Parameter data type:	VARCHAR2
Parameter default:	Defaulted
Parameter pass mode:	IN mode

Parameter name:	P_CORRECT
Parameter position:	2
Parameter data type:	NUMBER
Parameter default:	No Default
Parameter pass mode:	IN mode

Parameter name:	P_WRONG
Parameter position:	3
Parameter data type:	NUMBER
Parameter default:	No Default
Parameter pass mode:	IN mode

PL/SQL procedure successfully completed.

Tracking the program execution subprogram call stack

The call stack information lets you determine the stages of program execution. A program call stack includes the nested information of the referenced subprograms. You can track the call stack for a program using the `DBMS_UTILITY.FORMAT_CALL_STACK` function or `UTL_CALL_STACK` package.

The function `FORMAT_CALL_STACK` returns the current call stack information, which includes the object handle number, line number, and the name of the subprogram.

Let's create three standalone procedures: P1, P2, and P3. P3 invokes P2, while P2 invokes P1. By the time the program control reaches P1, the call stack includes P1, P3 and P2.

```
/*Create the procedure P1*/
CREATE OR REPLACE PROCEDURE P1
IS
BEGIN
    DBMS_OUTPUT.PUT_LINE ('Executing P1..');
    DBMS_OUTPUT.PUT_LINE (RPAD ('~',15,'~'));
    DBMS_OUTPUT.PUT_LINE (dbms_utility.format_call_stack);
END;
/

/*Create the procedure P2*/
CREATE OR REPLACE PROCEDURE P2
IS
BEGIN
    DBMS_OUTPUT.PUT_LINE ('Executing P2..');
    DBMS_OUTPUT.PUT_LINE ('Calling P1..');
    DBMS_OUTPUT.PUT_LINE (RPAD ('~',15,'~'));
/*Call procedure P1*/
    P1;
END;
/

/*Create the procedure P3*/
CREATE OR REPLACE PROCEDURE P3
IS
BEGIN
    DBMS_OUTPUT.PUT_LINE ('Executing P3..');
    DBMS_OUTPUT.PUT_LINE ('Calling P2..');
    DBMS_OUTPUT.PUT_LINE (RPAD ('~',15,'~'));
/*Call procedure P2*/
    P2;
END;
/

/*Enable the serveroutput to display the error messages*/
SET SERVEROUTPUT ON

/*Start a PL/SQL block to invoke P3*/
```

```

BEGIN
/*Call P3*/
  P3;
END;
/

Executing P3..
Calling P2..
~~~~~
Executing P2..
Calling P1..
~~~~~
Executing P1..
~~~~~
----- PL/SQL Call Stack -----
   object handle      line number   object name
   -----
0xec4c4ce0             6           procedure SCOTT.P1
0xec438220             8           procedure SCOTT.P2
0xe4e00448             8           procedure SCOTT.P3
0xe4c98298             1           anonymous block

PL/SQL procedure successfully completed.

```

In the preceding program output, you can check the program execution flow from P3 to P2, followed by P1. But one issue with the `FORMAT_CALL_STACK` output is the inability to extract the drill-down up to local routine information and use it for debugging tasks.

Oracle Database 12c introduces a new API, known as `UTL_CALL_STACK`, to give a structured call stack as well as error stack. The structured call stack output allows developers to extract the information and use it for debug purposes.

The procedure `SHOW_CALL_STACK` uses the `UTL_CALL_STACK` subprograms to display the structured call stack information:

```

/*Create the procedure to print call stack using UTL_CALL_STACK*/
CREATE OR REPLACE PROCEDURE show_call_stack
is
  lvl PLS_INTEGER;
BEGIN

  /*Retrieve the dynamic depth of the call stack */
  lvl := UTL_CALL_STACK.DYNAMIC_DEPTH();

  /*Iterate the call depth structure */

```



```
FOR i IN 1..lvl LOOP
    DBMS_OUTPUT.put_line(
        RPAD('Call depth:' || UTL_CALL_STACK.lexical_depth(i), 15) ||
        RPAD('Line:' || TO_CHAR(UTL_CALL_STACK.unit_line(i), '99'), 15) ||
        UTL_CALL_STACK.CONCATENATE_SUBPROGRAM
        (UTL_CALL_STACK.SUBPROGRAM(i))
    );
END LOOP;
END;
/
```

In the preceding procedure:

- DYNAMIC_DEPTH denotes the number of subprograms in the call stack
- LEXICAL_DEPTH is the relative call depth within a subprogram
- UNIT_LINE is the line number where the call statement is made
- SUBPROGRAM returns the name of the subprogram currently under execution
- CONCATENATE_SUBPROGRAM concatenates a subprogram name to its calling program

P_CALC_POINT_CALLSTACK invokes SHOW_CALL_STACK to print the structured call stack:

```
/*Create a procedure to invoke SHOW_CALL_STACK */
CREATE OR REPLACE PROCEDURE p_calc_point_callstack
(p_user VARCHAR2, p_correct NUMBER, p_wrong NUMBER)
IS
    l_num NUMBER;

    FUNCTION f_calc_points (p_ques NUMBER, p_factor NUMBER)
    RETURN NUMBER
    IS
    BEGIN
        /*Invoke SHOW_CALL_STACK */
        show_call_stack;
        RETURN (p_ques*p_factor);
    END;

    PROCEDURE p_net_calc (p_net_points OUT NUMBER) IS
    BEGIN
        p_net_points := f_calc_points (p_correct,4) + f_calc_points (p_
wrong,-2);
    END;
BEGIN
```

```

    p_net_calc (l_num);
    DBMS_OUTPUT.PUT_LINE (USER||' earned '||TO_CHAR (l_num)||' points');
end;
/

```

Execute the preceding procedure in a PL/SQL block:

```

SET SERVEROUTPUT ON
BEGIN
    p_calc_point_callstack (user, 10, 3);
END;
/

Call depth:1   Line: 10       P_CALC_POINT_CALLSTACK.F_CALC_POINTS
Call depth:1   Line: 15       P_CALC_POINT_CALLSTACK.P_NET_CALC
Call depth:0   Line: 18       P_CALC_POINT_CALLSTACK
Call depth:0   Line:  2       __anonymous_block
Call depth:0   Line:  6       SHOW_CALL_STACK
Call depth:1   Line: 10       P_CALC_POINT_CALLSTACK.F_CALC_POINTS
Call depth:1   Line: 15       P_CALC_POINT_CALLSTACK.P_NET_CALC
Call depth:0   Line: 18       P_CALC_POINT_CALLSTACK
Call depth:0   Line:  2       __anonymous_block
SCOTT earned 34 points

```

PL/SQL procedure successfully completed.

The preceding output shows the call stack in last-in-first-out order. To read the stack from top to bottom, you can loop the dynamic depth in REVERSE order.

Tracking propagating exceptions in PL/SQL code

Traditionally, database developers are keen to use `SQLERRM` and `SQLCODE` to find an exception code and message. In a modular programming approach, PL/SQL programs invoke routines and it may happen that a routine call fails with an exception. The exception propagates to the calling block and traverses through multiple blocks in search of its handler.

If you want to track the path of propagating exceptions, you can either use the `FORMAT_ERROR_BACKTRACE` function from `DBMS_UTILITY` or the new `UTL_CALL_STACK` package.

The PL/SQL procedure here raises user-defined exceptions to demonstrate the working of back-tracing the exceptions:

```
/*Create a procedure to trace print error stack */
CREATE OR REPLACE PROCEDURE p_calc_point_errStack
(p_user VARCHAR2, p_correct NUMBER, p_wrong NUMBER)
IS
    l_num      NUMBER;

/*Declare user defined exceptions */
myFunExp      EXCEPTION;
myProcExp     EXCEPTION;
myBlkExp      EXCEPTION;
PRAGMA EXCEPTION_INIT(myFunExp, -20020);
PRAGMA EXCEPTION_INIT(myProcExp, -20021);
PRAGMA EXCEPTION_INIT(myBlkExp, -20022);

/*Explicitly raise the user defined exception in the local function*/
FUNCTION f_calc_points (p_ques NUMBER, p_factor NUMBER)
RETURN NUMBER
IS
BEGIN
    RAISE myFunExp;
    RETURN (p_ques*p_factor);
EXCEPTION
WHEN myFunExp THEN
    RAISE myProcExp;
END;

/*Explicitly raise the user defined exception in the local procedure
*/
PROCEDURE p_net_calc (p_net_points OUT NUMBER) IS
BEGIN
    p_net_points := f_calc_points (p_correct,4) + f_calc_points (p_
wrong,-2);
EXCEPTION
WHEN myProcExp THEN
    RAISE myBlkExp;
END;

/*Explicitly raise the user defined exception in the program body*/
BEGIN
    p_net_calc (l_num);
    DBMS_OUTPUT.PUT_LINE (USER||' earned '||to_char (l_num)||' points');
EXCEPTION
```

```

    WHEN myBlkExp THEN
        DBMS_OUTPUT.PUT_LINE( DBMS_UTILITY.FORMAT_ERROR_BACKTRACE );
    END;
/

```

Upon execution, the block gives the following output:

```

SET SERVEROUTPUT ON
BEGIN
    p_calc_point_errstack (USER, 10, 3);
END;
/
ORA-06512: at "SCOTT.P_CALC_POINTS_ERRSTACK", line 28
ORA-06512: at "SCOTT.P_CALC_POINTS_ERRSTACK", line 20
ORA-06512: at "SCOTT.P_CALC_POINTS_ERRSTACK", line 32

```

PL/SQL procedure successfully completed.

Note that, since the exception in the inner block is not handled in the exception sections of the outer blocks, it propagates through the line numbers 19 (F_CALC_POINTS), 27 (P_NET_CALC), and 31 (P_CALC_POINTS_ERRSTACK).

Once again, FORMAT_ERROR_BACKTRACE presents an output which is hard to parse and doesn't include the names of local subprograms. Let's re-write the program using UTL_CALL_STACK.

The following procedure DISPLAY_ERROR_STACK shows the error stack along with the error numbers raised in the error block:

```

/*Create a procedure to print error stack using UTL_CALL_STACK*/
CREATE OR REPLACE PROCEDURE display_error_stack AS
    l_depth PLS_INTEGER;
BEGIN

    /*Retrieve the count of the error stack */
    l_depth := UTL_CALL_STACK.ERROR_DEPTH;

    /*Iterate the error stack structure to print errors */
    FOR i IN 1..l_depth LOOP
        DBMS_OUTPUT.put_line(
            RPAD(i, 10) ||
            RPAD('ORA-' || LPAD(UTL_CALL_STACK.error_number(i), 5, '0'), 10) ||
            UTL_CALL_STACK.error_msg(i)
        );
    END LOOP;
END;
/

```

In the preceding program code:

- ERROR_DEPTH is the count of errors in the error stack
- ERROR_NUMBER and ERROR_MSG are the error number and statement of the error in the stack

```
/*Rewrite the procedural logic to invoke DISPLAY_ERROR_STACK*/
CREATE OR REPLACE PROCEDURE p_calc_point_errStack
(p_user VARCHAR2, p_correct NUMBER, p_wrong NUMBER)
IS
  l_num NUMBER;

  /*Declare user-defined exceptions */
  myFunExp EXCEPTION;
  myProcExp EXCEPTION;
  myBlkExp EXCEPTION;
  PRAGMA EXCEPTION_INIT(myFunExp,-20020);
  PRAGMA EXCEPTION_INIT(myProcExp,-20021);
  PRAGMA EXCEPTION_INIT(myBlkExp,-20022);

  /*Explicitly raise the user defined exception in the local function*/
  FUNCTION f_calc_points (p_ques NUMBER, p_factor NUMBER)
  RETURN NUMBER
  IS
  BEGIN
    RAISE myFunExp;
    RETURN (p_ques*p_factor);
  EXCEPTION
  WHEN myFunExp THEN
    RAISE myProcExp;
  END;

  /*Explicitly raise the user defined exception in the local
  procedure*/
  PROCEDURE p_net_calc (p_net_points OUT NUMBER) IS
  BEGIN
    p_net_points := f_calc_points (p_correct,4) + f_calc_points (p_
    wrong,-2);
  EXCEPTION
  WHEN myProcExp THEN
    RAISE myBlkExp;
  END;
```

```

/*Explicitly raise the user defined exception in the program body*/
BEGIN
  p_net_calc (l_num);
  DBMS_OUTPUT.PUT_LINE (USER||' earned '||to_char (l_num)||' points');
EXCEPTION
  WHEN myBlkExp THEN
    display_error_stack;
END;
/

```

Execute the procedure P_CALC_POINT_ERRSTACK:

```

SET SERVEROUTPUT ON
BEGIN
  p_calc_point_errStack (user, 10, 3);
END;
/

1   ORA-20022

2   ORA-06512 at "SCOTT.P_CALC_POINT_ERRSTACK", line 27

3   ORA-20021

4   ORA-06512 at "SCOTT.P_CALC_POINT_ERRSTACK", line 19

5   ORA-20020

PL/SQL procedure successfully completed.

```

Determining identifier types and usages

A lexical unit in a PL/SQL program code is built up using literals, identifiers, delimiters, and comments. All items that are declared in a PL/SQL program as variables, cursors, constants, and subprogram names are identifiers. Identifiers can be reserved words (such as BEGIN and END), predefined (declared globally within STANDARD package), or quoted.

USER_IDENTIFIERS

The USER_IDENTIFIERS dictionary view reports the usage of identifiers in a PL/SQL program unit. The view includes information about an identifier's name, its type, and usage by line number in a PL/SQL program.

Tracking identifier details for all the subprograms would be additional task during code compilation and therefore, it is collected only for enabled PL/SQL objects.

The structure of the `USER_IDENTIFIERS` view is as follows:

Name	Null?	Type
-----	-----	-----
NAME		VARCHAR2 (128)
SIGNATURE		VARCHAR2 (32)
TYPE		VARCHAR2 (18)
OBJECT_NAME	NOT NULL	VARCHAR2 (128)
OBJECT_TYPE		VARCHAR2 (13)
USAGE		VARCHAR2 (11)
USAGE_ID		NUMBER
LINE		NUMBER
COL		NUMBER
USAGE_CONTEXT_ID		NUMBER
ORIGIN_CON_ID		NUMBER

Key points to note:

- The `SIGNATURE` is the unique code of an identifier and differentiates identifiers with the same name
- Use of an identifier can be `DECLARATION`, `DEFINITION`, `ASSIGNMENT`, `CALL`, or `REFERENCE`
- Use of an identifier in a program unit is uniquely identified by its Usage ID or the combination of line, column, and usage
- The `USAGE_CONTEXT_ID` establishes a self-referencing integrity with the usage ID

The PL/Scope tool

The PL/Scope tool was introduced in Oracle Database 11g to capture the use of identifiers in a PL/SQL program. The identifier details are stored in the `SYSAUX` tablespace and can be queried from the `USER_IDENTIFIERS` dictionary view (or its `ALL_*` or `DBA_*` views).



You can use the PL/Scope tool from SQL Developer.



Key features of the PL/Scope tool are as follows:

- The feature can be enabled at the database instance or session level. A PL/SQL subprogram can be compiled for PL/Scope.
- The PL/Scope tool cannot capture information for obfuscated subprograms.
- The PL/Scope tool cannot collect the identifier information if the `SYSAUX` tablespace is absent. However, no error is raised but a warning is stored in the `USER_ERRORS` view.

The `PLSCOPE_SETTINGS` parameter

The compilation parameter `PLSCOPE_SETTINGS` enables the collection of identifier data in a PL/SQL program. By default, the parameter is disabled (`IDENTIFIERS:NONE`). It should be set to `IDENTIFIERS:ALL` in order to enable identifier tracking.

If you enable this parameter at the instance level, Oracle will capture database wide identifier information. You must be careful while setting this parameter at the database instance level as it might impact the compilation performance of large packages.

Setting `PLSCOPE_SETTINGS` at system or session level:

```
ALTER [SYSTEM | SESSION]
SET PLSCOPE_SETTINGS = ['IDENTIFIERS:ALL' | 'IDENTIFIERS:NONE']
```

Being a compilation parameter, you can also compile a PL/SQL subprogram for identifier tracking by specifying the `PLSCOPE_SETTINGS` parameter:

```
ALTER [PROGRAM NAME] COMPILE
PLSCOPE_SETTINGS = ['IDENTIFIERS:ALL' | 'IDENTIFIERS:NONE']
```

You can query the current setting of the `PLSCOPE_SETTINGS` parameter from the `V$PARAMETER` dynamic dictionary view.

```
/*View the current setting of PLSCOPE_SETTINGS parameter*/
SELECT value
FROM v$parameter
WHERE name='plscope_settings'
/

VALUE
-----
IDENTIFIERS:NONE
```


You can view the space consumed by the PL/Scope tool in SYSAUX tablespace by running the following query as SYSDBA:

```
/*Verify the PL/Scope occupancy in SYSAUX tablespace*/
SELECT occupant_desc, schema_name, space_usage_kbytes
FROM v$sysaux_occupants
WHERE occupant_name='PL/SCOPE'
/
```

OCCUPANT_DESC	SCHEMA_NAME	SPACE_USAGE_KBYTES
-----	-----	-----
PL/SQL Identifier Collection	SYS	2496

Let us recompile the procedure P_CALC_USER_POINTS with the PLScope_SETTINGS parameter:

```
ALTER PROCEDURE p_calc_user_points COMPILE PLScope_
SETTINGS=' IDENTIFIERS:ALL'
/
```

Query the USER_PLSQL_OBJECT_SETTINGS view to verify whether the identifier information has been captured:

```
SELECT plscope_settings
FROM user_plsql_object_settings
WHERE name='P_CALC_USER_POINTS'
/
```

```
PLSCOPE_SETTINGS
-----
IDENTIFIERS:ALL
```

Oracle Database documentation provides a handy script for generating a hierarchical report of program identifiers by its usage. The following script generates the PL/Scope identifier report from the USER_IDENTIFIERS dictionary view:

```
WITH v AS
(
  SELECT line, col,
         name,
         LOWER(type) Type,
         LOWER(usage) Usage,
         usage_id,
         usage_context_id
  FROM user_identifiers
  WHERE object_name = 'P_CALC_USER_POINTS'
  AND object_type = 'PROCEDURE'
)
```

```

SELECT line, RPAD(LPAD(' ', 2*(level-1)) || name, 25, ' ') || ' ' ||
RPAD(Type, 15) || RPAD(Usage, 15) IDENTIFIER_USAGE_CONTEXTS
FROM v
START WITH usage_context_id = 0
CONNECT BY PRIOR usage_id = usage_context_id
ORDER SIBLINGS BY line, col
/
LINE IDENTIFIER_USAGE_CONTEXTS
-----
1  P_CALC_USER_POINTS..... procedure      declaration
1  P_CALC_USER_POINTS..... procedure      definition
1  P_USER..... formal in                  declaration
1  P_USER..... formal in                  assignment
1  USER..... function                    call
1  VARCHAR2..... character datatype      reference
1  P_CORRECT..... formal in              declaration
1  NUMBER..... number datatype          reference
1  P_WRONG..... formal in                declaration
1  NUMBER..... number datatype          reference
3  L_NUM..... variable                   declaration
3  NUMBER..... number datatype          reference
4  F_CALC_POINTS..... function            declaration
4  F_CALC_POINTS..... function            definition
4  P_QUES..... formal in                  declaration
4  NUMBER..... number datatype          reference
4  P_FACTOR..... formal in                declaration
4  NUMBER..... number datatype          reference
5  NUMBER..... number datatype          reference
9  P_QUES..... formal in                  reference
9  P_FACTOR..... formal in                reference
12 P_NET_CALC..... procedure              declaration
12 P_NET_CALC..... procedure              definition
12 P_NET_POINTS..... formal out           declaration
1  NUMBER..... number datatype          reference
14 P_NET_POINTS..... formal out           assignment
1  F_CALC_POINTS..... function            call
14 P_CORRECT..... formal in              reference
14 F_CALC_POINTS..... function            call
14 P_WRONG..... formal in                reference
17 P_NET_CALC..... procedure              call
17 L_NUM..... variable                   reference


32 rows selected.

```

In a similar way to the preceding query, application developers can explore multiple use cases where identifier information can be useful.

The DBMS_METADATA package

The DBMS_METADATA package was introduced in Oracle 9i. The API enables the extraction of object metadata from database dictionaries that can be optionally manipulated and re-executed on a database server. The package is owned by SYS whose public synonym is used by all users. A user with SELECT_CATALOG_ROLE can directly access the DBMS_METADATA package.

 DBMS_METADATA is a key enabler of metadata export functionality in Data Pump

The package pulls the object's metadata in XML form from the database dictionary and provides transform handlers to build it in the desired form. The formatted XML can then be re-executed in the database.

DBMS_METADATA data types and subprograms

As we said earlier, the DBMS_METADATA package uses the public synonyms of SYS-owned data structures. The following list shows SYS-owned object types:

- SYS.KU\$_PARSED_ITEM: An object type used to capture the attributes of an object's metadata. The object type structure is as follows:
CREATE TYPE sys.ku\$_parsed_item AS OBJECT
(
 item VARCHAR2(30),
 value VARCHAR2(4000),
 object_row NUMBER
)

ITEM, VALUE form the attribute name-value pair for OBJECT_ROW.
- SYS.KU\$_PARSED_ITEMS: A nested table of SYS.KU\$_PARSED_ITEM used to hold the object metadata attributes for multiple objects.
- SYS.KU\$_DDL: An object type used to capture the DDL of an object along with its parsed item information. The object type structure is as follows:

```
CREATE TYPE sys.ku$_ddl AS OBJECT  
(  
    ddlText CLOB,  
    parsedItem sys.ku$_parsed_items  
)
```

The parsed object information is stored in PARSEDITEM.

- `SYS.KU$_DDL`: A nested table of `SYS.KU$_DDL` returned by the `FETCH_DDL` subprogram used to hold the metadata of an object transformed into multiple DDL statements.
- `SYS.KU$_MULTI_DDL`: An object type used to hold the DDL for an object in multiple transforms.
- `SYS.KU$_MULTI_DDLS`: A nested table of `SYS.KU$_MULTI_DDL` returned by the `CONVERT` subprogram.
- `SYS.KU$_ERRORLINE`: An object type used to capture error information. The object type structure is as follows:

```
CREATE TYPE sys.ku$_ErrorLine IS OBJECT
(
    errorNumber NUMBER,
    errorText VARCHAR2(2000)
)
```

- `SYS.KU$_ERRORLINES`: A nested table of the `SYS.KU$_ERRORLINE` object type used to hold bulk error information during extraction of each DDL statement.
- `SYS.KU$_SUBMITRESULT`: An object type used to capture the complete error information incurred in a DDL statement. The object type structure is as follows:

```
CREATE TYPE sys.ku$_SubmitResult AS OBJECT
(
    ddl sys.ku$_ddl,
    errorLines sys.ku$_ErrorLines
)
```

- `SYS.KU$_SUBMITRESULTS`: A nested table of the `SYS.KU$_SUBMITRESULT` object type used to hold multiple DDL statements and associated error information.



A complete list of subprograms can be found on Oracle documentation at https://docs.oracle.com/database/121/ARPLS/d_metada.htm

The following table shows the frequently used subprograms of the DBMS_METADATA package:

Subprogram	Remarks
ADD_TRANSFORM function	Specifies a transform that FETCH_[XML DDL CLOB] applies to the XML representation of the retrieved objects
CLOSE procedure	Invalidates the handle returned by OPEN and cleans up the associated state
CONVERT functions and procedures	Convert an XML document to DDL
FETCH_[XML DDL CLOB] functions and procedures	Returns metadata for objects meeting the criteria established by OPEN, SET_FILTER, SET_COUNT, ADD_TRANSFORM, and so on
GET_[XML DDL CLOB] functions	Fetches the metadata for a specified object as XML or DDL in a single call
GET_QUERY function	Returns the text of the queries that are used by FETCH_[XML DDL CLOB]
OPEN function	Specifies the type of object to be retrieved, the version of its metadata, and the object model
OPENW function	Opens a write context
PUT function	Submits an XML document to the database
SET_COUNT procedure	Specifies the maximum number of objects to be retrieved in a single FETCH_[XML DDL CLOB] call
SET_FILTER procedure	Specifies restrictions on the objects to be retrieved—for example, the object name or schema
SET_PARSE_ITEM procedure	Enables output parsing by specifying an object attribute to be parsed and returned
SET_TRANSFORM_PARAM and SET_REMAP_PARAM procedures	Specifies parameters to the XSLT style sheets identified by transform_handle

Out of the preceding list, the subprograms can be grouped based on their work function:

Subprograms used to retrieve multiple objects from the database	Subprograms used to submit XML metadata to the database
ADD_TRANSFORM function	ADD_TRANSFORM function
CLOSE procedure	CLOSE procedure 2
FETCH_[XML DDL CLOB] functions and procedures	CONVERT functions and procedures
GET_QUERY function	OPENW function
GET_[XML DDL CLOB] functions	PUT function
OPEN function	SET_PARSE_ITEM procedure
SET_COUNT procedure	SET_TRANSFORM_PARAM and SET_REMAP_PARAM procedures
SET_FILTER procedure	
SET_PARSE_ITEM procedure	
SET_TRANSFORM_PARAM and SET_REMAP_PARAM procedures	

Parameter requirements

You must note that the parameters to the metadata API are case-sensitive and should be passed in their respective position only (no named notation).

The DBMS_METADATA transformation parameters and filters

As listed in the preceding API list, the SET_TRANSFORM_PARAM subprogram is used to format and control the DDL output. It is used for both retrieval and submission of metadata from or to the database. It is an overloaded procedure with the following syntax:

```
DBMS_METADATA.SET_TRANSFORM_PARAM
(
    transform_handle IN NUMBER,
    name IN VARCHAR2,
    value IN VARCHAR2,
    object_type IN VARCHAR2 DEFAULT NULL
);
DBMS_METADATA.SET_TRANSFORM_PARAM
```

```
(
  transform_handle IN NUMBER,
  name IN VARCHAR2,
  value IN BOOLEAN DEFAULT TRUE,
  object_type IN VARCHAR2 DEFAULT NULL
);
DBMS_METADATA.SET_TRANSFORM_PARAM
(
  transform_handle IN NUMBER,
  name IN VARCHAR2,
  value IN NUMBER,
  object_type IN VARCHAR2 DEFAULT NULL
);
```

In the preceding program signature:

- **TRANSFORM_HANDLE:** A handler either from `ADD_TRANSFORM`, or a generic handler constant `SESSION_TRANSFORM` used to affect the whole session
- **NAME:** Name of the parameter to be modified
- **VALUE:** New value of the parameter

The following is a list of the common set of parameters that are applicable to all objects in a schema:

Parameter	Value	Meaning
PRETTY	TRUE FALSE (default value is TRUE)	If TRUE, produces properly indented output
SQLTERMINATOR	TRUE FALSE (default value is FALSE)	If TRUE, appends SQL terminator (; or /) after each DDL
DEFAULT	TRUE FALSE	If TRUE, resets all parameters to their default state
INHERIT	TRUE FALSE	If TRUE, inherits session -level settings

Transform handlers applicable for tables and views are as follows:

Parameter	Value	Meaning
SEGMENT_ATTRIBUTES	TRUE FALSE (default value is TRUE)	If TRUE, includes segment, tablespace, logging, and physical attributes
STORAGE	TRUE FALSE (default value is FALSE)	If TRUE, includes storage clause

Parameter	Value	Meaning
TABSPACE	TRUE FALSE	If TRUE, includes tablespace specification
CONSTRAINTS	TRUE FALSE	If TRUE, includes table constraints
REF_CONSTRAINTS	TRUE FALSE	If TRUE, includes referential constraints
CONSTRAINTS_AS_ALTER	TRUE FALSE	If TRUE, includes constraints in the ALTER TABLE statements
OID	TRUE FALSE	If TRUE, includes the object table OID
SIZE_BYTE_KEYWORD	TRUE FALSE	If TRUE, includes the BYTE keywords in string type column specifications
FORCE	TRUE FALSE	If TRUE, creates view with the FORCE option

The `DBMS_METADATA.SET_FILTER` procedure is used to specify the filters on the schema objects. It accepts the metadata handle, filter name, and its value as input arguments.

```
PROCEDURE set_filter(
  handle   IN NUMBER,
  name     IN VARCHAR2,
  value    IN VARCHAR2|BOOLEAN|NUMBER,
  object_type_path VARCHAR2
)
```

Frequently used filters can be schema, user, object dependencies, table data, tables, indexes, constraints, and so on. There are more than 70 filters available in Oracle 11g. It can be set as follows:

```
DBMS_METADATA.SET_FILTER(handle, 'SCHEMA', 'SCOTT');
DBMS_METADATA.SET_FILTER(handle, 'NAME', 'DEPARTMENTS');
```

Demonstration

Let us retrieve the DDL of the entire EMP table in the SCOTT schema. The SQL query returns a CLOB output:

```
/*Query to retrieve DDL of EMP table */
SELECT dbms_metadata.get_ddl('TABLE', 'EMP', 'SCOTT')
FROM DUAL
/

CREATE TABLE "SCOTT"."EMP"
```



```
    ("EMPNO" NUMBER(4,0),
    "ENAME" VARCHAR2(10),
    "JOB" VARCHAR2(9),
    "MGR" NUMBER(4,0),
    "HIREDATE" DATE,
    "SAL" NUMBER(7,2),
    "COMM" NUMBER(7,2),
    "DEPTNO" NUMBER(2,0),
    "EMAIL" VARCHAR2(50),
    CONSTRAINT "PK_EMP" PRIMARY KEY ("EMPNO")
USING INDEX
PCTFREE 10
INITTRANS 2
MAXTRANS 255
COMPUTE STATISTICS
STORAGE(INITIAL 65536
    NEXT 1048576
    MINEXTENTS 1
    MAXEXTENTS 2147483645
    PCTINCREASE 0
    FREELISTS 1
    FREELIST GROUPS 1
    BUFFER_POOL DEFAULT
    FLASH_CACHE DEFAULT
    CELL_FLASH_CACHE DEFAULT)
TABLESPACE "USERS" ENABLE,
    CONSTRAINT "FK_DEPTNO"
    FOREIGN KEY ("DEPTNO")
    REFERENCES "SCOTT"."DEPT" ("DEPTNO") ENABLE)
SEGMENT CREATION IMMEDIATE
PCTFREE 10 PCTUSED 40 INITTRANS 1 MAXTRANS 255
NOCOMPRESS LOGGING
STORAGE(INITIAL 65536
    NEXT 1048576
    MINEXTENTS 1
    MAXEXTENTS 2147483645
    PCTINCREASE 0
    FREELISTS 1
    FREELIST GROUPS 1
    BUFFER_POOL DEFAULT
    FLASH_CACHE DEFAULT
    CELL_FLASH_CACHE DEFAULT)
TABLESPACE "USERS"
```

In the preceding output, if you wish to skip the storage clause specification for the table, you can do so by setting the transform parameters:

```
/*Anonymous block to set transform handles */
BEGIN
  DBMS_METADATA.SET_TRANSFORM_PARAM (DBMS_METADATA.SESSION_
TRANSFORM, 'STORAGE', false);
  DBMS_METADATA.SET_TRANSFORM_PARAM (DBMS_METADATA.SESSION_
TRANSFORM, 'SEGMENT_ATTRIBUTES', false);
  DBMS_METADATA.SET_TRANSFORM_PARAM (DBMS_METADATA.SESSION_
TRANSFORM, 'PRETTY', true);
  DBMS_METADATA.SET_TRANSFORM_PARAM (DBMS_METADATA.SESSION_TRANSFORM, 'S
QLTERMINATOR', true);
  DBMS_METADATA.SET_TRANSFORM_PARAM (DBMS_METADATA.SESSION_
TRANSFORM, 'REF_CONSTRAINTS', false);
  DBMS_METADATA.SET_TRANSFORM_PARAM (DBMS_METADATA.SESSION_
TRANSFORM, 'TABLESPACE', false);
  DBMS_METADATA.SET_TRANSFORM_PARAM (DBMS_METADATA.SESSION_
TRANSFORM, 'SIZE_BYTE_KEYWORD', false);
END;
/
```

Now, if you execute the GET_DDL function to retrieve the DDL of the EMP table, the output will be much neater and cleaner:

```
/*Query to retrieve DDL of EMP table */
SELECT dbms_metadata.get_ddl('TABLE', 'EMP', 'SCOTT')
FROM DUAL
/

CREATE TABLE "SCOTT"."EMP"
  ("EMPNO" NUMBER(4,0),
  "ENAME" VARCHAR2(10),
  "JOB" VARCHAR2(9),
  "MGR" NUMBER(4,0),
  "HIREDATE" DATE,
  "SAL" NUMBER(7,2),
  "COMM" NUMBER(7,2),
  "DEPTNO" NUMBER(2,0),
  "EMAIL" VARCHAR2(50),
  CONSTRAINT "PK_EMP" PRIMARY KEY ("EMPNO")
  USING INDEX ENABLE
);
```

You can use the GET_XML function to retrieve the XML format of the EMP table.

The following SQL query retrieves the object grants of the EMP table using the GET_DEPENDENT_DDL function:

```
/*Query to retrieve DDL of object grants of EMP */
SELECT
  DBMS_METADATA.GET_DEPENDENT_DDL ('OBJECT_GRANT','EMP','SCOTT') OBJ_
GRANTS
FROM DUAL
/

GRANT SELECT ON "SCOTT"."EMP" TO "SALES";
GRANT SELECT ON "SCOTT"."EMP" TO "MGR";
GRANT SELECT ON "SCOTT"."EMP" TO "CLERK";
GRANT INSERT ON "SCOTT"."EMP" TO "SALES";
GRANT INSERT ON "SCOTT"."EMP" TO "MGR";
GRANT INSERT ON "SCOTT"."EMP" TO "CLERK";
```

You can also retrieve the DDL for the indexes on the table columns. The following SELECT query displays the DDL for the unique index on the EMPNO column of the EMP table:

```
/*Query to retrieve DDL of primary key in EMP table */
SELECT DBMS_METADATA.GET_DDL('INDEX','PK_EMP','SCOTT')
FROM DUAL
/

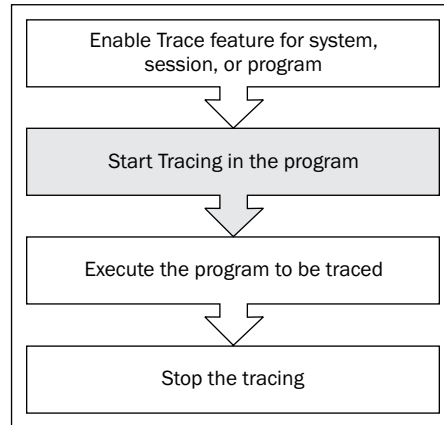
CREATE UNIQUE INDEX "SCOTT"."PK_EMP" ON "SCOTT"."EMP" ("EMPNO")
;
```

Tracing PL/SQL programs using DBMS_TRACE

Tracing program execution is an important exercise in a scrum based database development environment. Sometimes, in a modular programming model, it becomes difficult to track the program execution path. Oracle provides the DBMS_TRACE package to trace PL/SQL program code.

DBMS_TRACE is an Oracle supplied package that can be used to enable and disable tracing in database sessions. The program execution path is traced when you execute a PL/SQL program in a trace-enabled session. The trace information is captured and stored in database tables. These trace tables can be further analyzed to examine the execution path of a PL/SQL program.

The following figure shows the steps involved in PL/SQL code tracing:



PL/SQL code tracing cannot be done in a shared server environment

Installing the DBMS_TRACE package

You can install the DBMS_TRACE package by running the following scripts from the \$ORACLE_HOME/rdbms/admin folder:

- dbmspbt.sql: This script creates the DBMS_TRACE package specification
- prvtpbt.plb: This script creates the DBMS_TRACE package body



The scripts must be executed as the SYS user and in the same order.

DBMS_TRACE subprograms

The DBMS_TRACE package contains trace constants and member subprograms. The trace constants are used to specify the tracing level in a session.



For complete details, please refer to the Oracle documentation at https://docs.oracle.com/database/121/ARPLS/d_trace.htm.

The DBMS_TRACE constants are summarized as follows:

DBMS_TRACE constant	Default	Remarks
TRACE_ALL_CALLS	1	Traces all calls
TRACE_ENABLED_CALLS	2	Traces calls that are enabled for tracing
TRACE_ALL_EXCEPTIONS	4	Traces all exceptions
TRACE_ENABLED_EXCEPTIONS	8	Traces exceptions that are enabled for tracing
TRACE_ALL_SQL	32	Traces all SQL statements
TRACE_ENABLED_SQL	64	Traces SQL statements that are enabled for tracing
TRACE_ALL_LINES	128	Traces each line
TRACE_ENABLED_LINES	256	Traces lines that are enabled for tracing
TRACE_PAUSE	4096	Pauses tracing (controls the tracing process)
TRACE_RESUME	8192	Resume tracing (controls the tracing process)
TRACE_STOP	16384	Stops tracing (controls the tracing process)
TRACE_LIMIT	16	Limits the trace information (controls the tracing process)
TRACE_MINOR_VERSION	0	Administers the tracing process
TRACE_MAJOR_VERSION	1	Administers the tracing process
NO_TRACE_ADMINISTRATIVE	32768	Prevents tracing of administrative events such as: <ul style="list-style-type: none">• PL/SQL Trace Tool started• Trace flags changed• PL/SQL Virtual Machine started• PL/SQL Virtual Machine stopped
NO_TRACE_HANDLED_EXCEPTIONS	65536	Prevents tracing of handled exceptions

The subprograms contained in the DBMS_TRACE package are as follows:

DBMS_TRACE subprogram	Remarks
CLEAR_PLSQL_TRACE procedure	Stops trace data dumping in session
GET_PLSQL_TRACE_LEVEL function	Gets the trace level

DBMS_TRACE subprogram	Remarks
GET_PLSQL_TRACE_RUNNUMBER function	Gets the current sequence of execution of trace
PLSQL_TRACE_VERSION procedure	Gets the version number of the trace package
SET_PLSQL_TRACE procedure	Starts tracing in the current session
COMMENT_PLSQL_TRACE procedure	Includes comment on the PL/SQL tracing
INTERNAL_VERSION_CHECK function	Has a value as 0, if the internal version check has not been done
LIMIT_PLSQL_TRACE procedure	Sets limit for the PL/SQL tracing
PAUSE_PLSQL_TRACE procedure	Pauses the PL/SQL tracing
RESUME_PLSQL_TRACE procedure	Resumes the PL/SQL tracing

In the preceding list, the key subprograms are:

- SET_PLSQL_TRACE: It kicks off the PL/SQL tracing session. For example, DBMS_TRACE.SET_PLSQL_TRACE (DBMS_TRACE.TRACE_ALL_SQL) traces all SQL in the program.
- CLEAR_PLSQL_TRACE: It stops the tracing session.

The PLSQL_TRACE_VERSION returns the current trace version as the OUT parameter.



The trace level that controls the tracing process (stop, pause, resume, and limit) cannot be used in combination with other trace levels

Compiling a PL/SQL program for debugging

A PL/SQL subprogram can be traced only after it is compiled in debug mode. You must compile the program to be traced by specifying the PLSQL_OPTIMIZE_LEVEL as 1. (In earlier releases, the compilation parameter PLSQL_DEBUG was used to compile a program for debug.)



The PLSQL_DEBUG parameter has been deprecated in Oracle Database 12c, but retained for backward-compatibility. Oracle recommends to use PLSQL_OPTIMIZE_LEVEL as 1 to compile a PL/SQL program for debugging.

Viewing the PL/SQL trace information

The trace information is logged into the trace tables that can be created by invoking the `$ORACLE_HOME/rdbms/admin/tracetab.sql` script from the SYS account. The successful execution of the script will create the following two tables:

- **PLSQL_TRACE_RUNS:** The table stores execution-specific information. RUNID is a unique run identifier that derives its value from a sequence, PLSQL_TRACE_RUNNUMBER. The RUN_DATE and RUN_END columns specify the start and end time of the run respectively. The RUN_SYSTEM_INFO and SPARE1 columns are currently unused columns in the table.
- **PLSQL_TRACE_EVENTS:** This table displays the traces of the program execution path. The table structure is described as follows:
 - The RUNID column references the RUNID column of the PLSQL_TRACE_RUNS table
 - The EVENT_SEQ is the unique event identifier within a single run
 - The EVENT_UNIT, EVENT_UNIT_KIND, EVENT_UNIT_OWNER, and EVENT_LINE columns capture the program unit information (such as name, type, owner, and line number) that initiates the trace event
 - The PROC_NAME, PROC_UNIT, PROC_UNIT_KIND, PROC_OWNER, and PROC_LINE columns capture the procedure information (such as name, type, owner, and line number) that is currently traced
 - The EXCP and USER_EXCP columns apply to the exceptions occurring during the trace
 - The EVENT_COMMENT column gives a user-defined comment or the actual event description
 - The MODULE, ACTION, CLIENT_INFO, CLIENT_ID, ECID_ID, and ECID_SEQ columns capture information about the session running on a SQL*Plus client
 - The CALLSTACK and ERRORSTACK columns store the call stack information

Once the script has been executed, the DBA can optionally create their public synonyms that can be accessed by all users.

```
/*Connect as SYSDBA*/
Conn sys/oracle as SYSDBA

/*Create synonym for PLSQL_TRACE_RUNS*/
CREATE PUBLIC SYNONYM plsqli_trace_runs FOR plsqli_trace_runs
/
```

```

/*Create synonym for PLSQL_TRACE_EVENTS*/
CREATE PUBLIC SYNONYM plsqli_trace_events FOR plsqli_trace_events
/

/*Create synonym for PLSQL_TRACE_RUNNUMBER sequence*/
CREATE PUBLIC SYNONYM plsqli_trace_runnumber FOR plsqli_trace_runnumber
/

/*Grant privileges on the PLSQL_TRACE_RUNS*/
GRANT select, insert, update, delete ON plsqli_trace_runs TO PUBLIC
/

/*Grant privileges on the PLSQL_TRACE_EVENTS*/
GRANT select, insert, update, delete ON plsqli_trace_events TO PUBLIC
/

/*Grant privileges on the PLSQL_TRACE_RUNNUMBER*/
GRANT select ON plsqli_trace_runnumber TO PUBLIC
/

```

Steps to trace PL/SQL program execution

PL/SQL tracing is demonstrated in the following steps:

1. Compile the procedure P_CALC_USER_POINTS for debug by specifying PLSQL_OPTIMIZE_LEVEL as 1:

```

ALTER PROCEDURE p_calc_user_points COMPILE PLSQL_OPTIMIZE_LEVEL=1
/

```

2. Enable tracing in the session:

Note the trace levels while enabling tracing in the current session. The following PL/SQL block will enable tracing of all calls excluding the tracing of administrative events.

```

BEGIN
/*Enable tracing for all calls in the session*/
  DBMS_TRACE.SET_PLSQL_TRACE(
    DBMS_TRACE.TRACE_ALL_CALLS +
    DBMS_TRACE.NO_TRACE_ADMINISTRATIVE);
END;
/

```

3. Execute the procedure P_CALC_USER_POINTS:

```

SET SERVEROUTPUT ON
BEGIN

```



```
p_calc_user_points (USER, 10, 3);  
END;  
/
```

4. Disable tracing in the session:

```
BEGIN  
/*Stop the trace session*/  
  DBMS_TRACE.CLEAR_PLSQL_TRACE;  
END;  
/
```

5. Query the trace log tables PLSQL_TRACE_RUNS and PLSQL_TRACE_EVENTS:

Query the RUNID of the current trace from PLSQL_TRACE_RUNS

```
SELECT MAX(runid)  
FROM PLSQL_TRACE_RUNS  
WHERE run_owner='SCOTT'  
/
```

```
MAX(RUNID)  
-----  
13
```

For RUNID=13, query the trace events from PLSQL_TRACE_EVENTS

```
SELECT event_seq,  
       event_comment,  
       event_unit_owner||'.'||event_unit unit,  
       proc_name,  
       proc_unit,  
       proc_unit_kind  
FROM PLSQL_TRACE_EVENTS  
WHERE RUNID=13  
ORDER BY EVENT_SEQ  
/
```

Profiling PL/SQL code

Oracle enables database developers to perform dynamic analysis of their PL/SQL code through tracing and profiling. As an application developer, you write a multi-line PL/SQL subprogram, which may include SQL statements, PL/SQL constructs, calls to routine subprograms, exceptions, and many more items. You can follow the execution flow of the program by using the DBMS_TRACE package, but it doesn't reveal the time consumed at each step. You can profile the PL/SQL program to check its performance aspects. The performance profile reveals how much time is spent at each line of code in a PL/SQL program.

Profiling is a vital exercise in the development stage of a database, as you can identify the areas in your PL/SQL program code that can be fine-tuned for performance. Oracle provides two built-in utility packages to profile PL/SQL code: `DBMS_PROFILER` and `DBMS_HPROF`. The `DBMS_PROFILER` package gathers the performance metrics and produces a flat profiler output. On the other hand, `DBMS_HPROF` gathers the profiler data and writes the profile information into interactive HTML reports.

In the scope of this chapter, we will discuss the `DBMS_HPROF` package.

The `DBMS_HPROF` package

The `DBMS_HPROF` package was introduced in Oracle Database 11g Release 1. It enables hierarchical profiling of a PL/SQL program and provides a detailed performance analysis of time spent by subprogram calls, by namespace, and by call descendants through the HTML reports. It is known as a "hierarchical profiler" because of its ability to drill-down to descendant and sub-tree levels of a subprogram execution. The distinctive features of a hierarchical profiler are as follows:

- It gathers and stores profiler information into database tables
- It is supported by SQL Developer
- It reports the performance of SQL and PL/SQL execution in the program
- The number of distinct subprograms calls and time spent in each one of them
- The subprogram call hierarchy

The `DBMS_HPROF` package is installed by default and is executed with the invoker's privileges. A user should have the `EXECUTE` privilege on the SYS-owned `DBMS_HPROF` tool.

Differences between `DBMS_PROFILER` and `DBMS_HPROF`

`DBMS_PROFILER` profiles a given PL/SQL program at the line level, while `DBMS_HPROF` builds a call level profile of a program. `DBMS_HPROF` is easy to use and requires no additional efforts when run on critical database environments. `DBMS_PROFILER` produces flat output while the raw data from `DBMS_HPROF` can be further analyzed and written into HTML reports.

DBMS_HPROF subprograms

The DBMS_HPROF package contains the subprograms to collect the profile data (**Data Collector**) and analyze it (**Analyzer**).

- **Data Collector:** The following subprograms open and close the profiling window. All the PL/SQL programs executed within this window will be profiled and written in raw format to a file.
 - START_PROFILING procedure
 - STOP_PROFILING procedure
- **Analyzer:** The ANALYZE subprogram reads and analyzes the raw profiler data and populates the profiler tables. You can analyze either the complete raw profiler data or even a particular subprogram call. Oracle also lets you profile the allocation of **UGA (User Global Area)** and **PGA (Program Global Area)** for a function during the program execution.

Collecting raw profile data

We will list the steps to profile the execution of the P_CALC_USER_POINTS procedure:

1. Create a database directory. Oracle will create the raw profiler data output file in this location:

```
connect sys/oracle as SYSDBA
CREATE DIRECTORY dir_profiles
AS '/u01/app/oracle/diag/profiles/'
/
```

2. Make the necessary grants to the SCOTT user. This step is an important prerequisite for a user to use the DBMS_HPROF package:

```
connect sys/oracle as SYSDBA
GRANT READ,WRITE ON DIRECTORY dir_profiles TO SCOTT
/
GRANT EXECUTE ON DBMS_HPROF TO SCOTT
/
```

3. Enable profiling in the current session:

```
connect scott/tiger
BEGIN
    DBMS_HPROF.START_PROFILING ('DIR_PROFILES', 'hprof_p_calc.log');
END;
/
```

4. Execute the P_CALC_USER_POINTS procedure:

```
connect scott/tiger
SET SERVEROUT ON
BEGIN
    P_CALC_USER_POINTS (USER, 10, 3);
END;
/
```

5. Stop profiling in the session

```
connect scott/tiger
BEGIN
/*Stop the profiling */
    DBMS_HPROF.STOP_PROFILING;
END;
/
```

6. Check the raw profiler data created at the specified directory location:

Oracle gathers the profiler data in text format but the raw format is hard to interpret and draw conclusions from. Following are the first initial lines of the raw profiler data:

```
P#V PLSHPROF Internal Version 1.0
P#! PL/SQL Timer Started
P#C PLSQL."".""."__plsql_vm"
P#X 14
P#C PLSQL."".""."__anonymous_block"
P#X 334
P#C SQL."SYS"."STANDARD"::11."__static_sql_exec_line180" #180
P#X 3504
P#R
P#X 132056
P#C PLSQL."SCOTT"."P_CALC_USER_POINTS"::7."P_CALC_USER_
POINTS"#9d831f6c5a526d3e #1
P#X 224
P#C PLSQL."SCOTT"."P_CALC_USER_POINTS"::7."P_CALC_USER_POINTS.P_
NET_CALC"#e17d780a3c3eae3d #12
P#X 1
```

Interpreting the raw profiler data

Although the raw profiler data is a bit tricky to understand and interpret, you can get first-level indications from the following information.

Event indicators — each line of the profiler starts with an event indicator, which carries a meaning. Here are the distinct event indicators from the profiler output:

- P#V: PLSHPROF banner information
- P#!: Comment
- P#C: Call to SQL or PL/SQL subprogram:
Determines the call namespace, line number, calling and called subprogram name, and hash signature.
- P#X: Elapsed time between the two events
- P#R: Return from a subprogram call

Operational functions — Functions to execute a PL/SQL program can be the following:

- `__anonymous_block` indicates anonymous block execution
- `__dyn_sql_exec_line` indicates dynamic SQL execution at line#
- `__pkg_init` indicates PL/SQL package initialization
- `__plsql_vm` indicates PL/SQL virtual machine call
- `__sql_fetch_line` indicates fetch operation at line#
- `__static_sql_exec_line` indicates static SQL execution at line#

Analyzing profiler data

After the raw profiler data is collected, we will analyze it and populate the profiler tables. Let us now create the profiler tables.

Creating the profiler tables

By default, the profiler tables are not created. Therefore, either the SYS DBA or the user with DBA privileges will have to run the `$ORACLE_HOME/rdbms/admin/dbmshtab.sql` script to create the profiler tables. On execution of this script, the following three tables are created:

- `DBMSHP_RUNS`: Maintains flat information about each command executed during profiling
- `DBMSHP_FUNCTION_INFO`: Contains information about the profiled function
- `DBMSHP_PARENT_CHILD_INFO`: Contains parent-child profiler information

A database administrator can create a public synonym on the preceding tables or grant the `SELECT` privilege to the user who intends to use the profiler.

Analyzing the profiler output

Before invoking the subprogram to analyze the raw profiler data, first grant object privileges to the SCOTT user:

```
connect sys/oracle as sysdba
GRANT select, insert on DBMSHP_RUNS to scott
/
GRANT select, insert on DBMSHP_FUNCTION_INFO to scott
/
GRANT select, insert on DBMSHP_PARENT_CHILD_INFO to scott
/
```

The following PL/SQL anonymous block invokes the ANALYZE subprogram to interpret the trace file passed as the parameter:

```
connect scott/tiger
/*Start the PL/SQL block*/
DECLARE
    l_runid NUMBER;
BEGIN

    /*Invoke the analyzer API*/
    l_runid := DBMS_HPROF.analyze
        (location      => 'DIR_PROFILES',
        FILENAME       => 'hprof_p_calc.log',
        run_comment    => 'Analyzing the execution of P_CALC_USER_POINTS');

    DBMS_OUTPUT.put_line('l_runid=' || l_runid);
END;
/
```

Querying the profiler tables

You can query profiler data from the DBMSHP_RUNS, DBMSHP_FUNCTION_INFO, and DBMSHP_PARENT_CHILD_INFO tables. The following SQL query selects the most recent RUNID from the DBMSHP_RUNS table:

```
SELECT runid, total_elapsed_time, run_comment
FROM dbmshp_runs
ORDER BY runid
/

RUNID TOTAL_EL RUN_COMMENT
-----
2 19973 Analyzing the execution of P_CALC_USER_POINTS
```

The following SQL query selects the subprogram information from DBMSHP_FUNCTION_INFO:

```
SELECT namespace,
       function,
       module,
       calls,
       function_elapsed_time "time_ms"
FROM dbmshp_function_info
WHERE runid = 2
/
```

NAMES	FUNCTION	MODULE	CALLS
time_ms			
-----	-----	-----	-----
PLSQL	__anonymous_block		2
132558			
PLSQL	__plsql_vm		2
23			
PLSQL	P_CALC_USER_POINTS	P_CALC_USER_POINTS	1
2489			
PLSQL	P_CALC_USER_POINTS.F_CALC_POINTS	P_CALC_USER_POINTS	2
14			
PLSQL	P_CALC_USER_POINTS.P_NET_CALC	P_CALC_USER_POINTS	1
6			
PLSQL	STOP_PROFILING	DBMS_HPROF	1
0			
PLSQL	PUT_LINE	DBMS_OUTPUT	1
3			
SQL	__static_sql_exec_line180	STANDARD	2
3727			

8 rows selected.

The preceding output shows the time elapsed in each program function along with the namespace to which it belongs. The profiler data in the tables can be used to build custom reports in development tools.

The plshprof utility

Oracle provides a command-line utility tool, called `plshprof`, to perform in-depth analysis of raw profiler data and generate neat HTML reports. It can also be used to generate a difference report between two raw profiler output files. The tool is not dependent on the analyzer phase or the profiler table's data.

The `plshprof` utility syntax is as follows:

```
[oracle@packt ~]$ plshprof
PLSHPROF: Oracle Database 12c Enterprise Edition Release 12.1.0.2.0 -
64bit Production
Usage: plshprof [<option>...] <tracefile1> [<tracefile2>]
Options:
  -trace <symbol>    (no default) specify function name of tree root
  -skip <count>      (default=0) skip first <count> invocations
  -collect <count>   (default=1) collect info for <count> invocations
  -output <filename> (default=<symbol>.html or <tracefile1>.html)
  -summary                               print time only
```

In the following example, the `plshprof` utility summarizes the raw profiler data from the `log hprof_p_calc.log`.

```
[oracle@profiles]$ plshprof -summary hprof_p_calc.log
PLSHPROF: Oracle Database 12c Enterprise Edition Release 12.1.0.2.0 -
64bit Production
Total subtree time: 138820 microsecs (elapsed time)
```

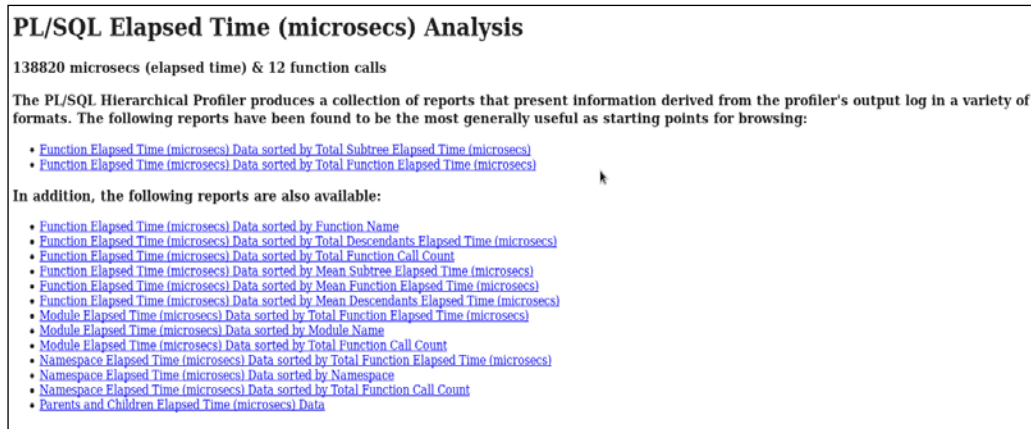
Note that the summary only gives you the total elapsed time. Let us now fully analyze the profiler data and generate the HTML output:

```
[oracle@profiles]$ plshprof -output HPROF_PCALC hprof_p_calc.log
PLSHPROF: Oracle Database 12c Enterprise Edition Release 12.1.0.2.0 -
64bit Production
[8 symbols processed]
[Report written to 'HPROF_PCALC.html']
```

Once the `plshprof` utility is processed, the following HTML files are generated in the directory location:

```
[oracle@ profiles]$ ls *.html
HPROF_PCALC_2c.html  HPROF_PCALC_mf.html  HPROF_PCALC_tc.html
HPROF_PCALC_2f.html  HPROF_PCALC_ms.html  HPROF_PCALC_td.html
HPROF_PCALC_2n.html  HPROF_PCALC_nsc.html  HPROF_PCALC_tf.html
HPROF_PCALC_fn.html  HPROF_PCALC_nsf.html  HPROF_PCALC_ts.html
HPROF_PCALC.html     HPROF_PCALC_nsp.html
HPROF_PCALC_md.html  HPROF_PCALC_pc.html
```


Here, HPROF_PCALC.html is the report index file and links all other profiler reports. The main index page is shown in the following screenshot:



What do these reports reveal?

The various reports give a detailed breakdown of elapsed time by each program function, module, descendent, and namespace. Let's explore the main reports from the complete set:

- **Function Elapsed Time (microsecs) Data sorted by Total Subtree Elapsed Time (microsecs):** This report provides a flat view of the raw profiler data. It includes the total call count, self time, subtree time, and descendants of each function.
- **Function Elapsed Time (microsecs) Data sorted by Total Function Elapsed Time (microsecs):** This module-level summary report shows the total time spent in each module and the total calls to the functions in the module.
- **Namespace Elapsed Time (microsecs) Data sorted by Namespace:** This report provides the distribution of time spent by the PL/SQL and SQL engines. SQL and PL/SQL are the two namespace categories available for a block. It is very useful in reducing the disk I/O and hence enhancing block performance. The net sum of the distribution is always 100 percent.

Summary

In this chapter, we learned various techniques to maintain PL/SQL program code. A skill in analyzing a PL/SQL program is required in order to troubleshoot code issues or diagnose performance bottlenecks. There are many dictionary views that can provide lot of metadata about a program, but developers need to pick the right one for the job. Tracing and profiling are the techniques with which database developers should be familiar. In large scale developments, tracing and profiling can be of great help in identifying opportunities to improve performance.

In the next chapter, we will discuss how to safeguard your database applications against SQL injection attacks.

Practice exercise

- Which of the following dictionary views is used to get information about subprogram arguments?

1. ALL_OBJECTS
2. ALL_ARGUMENTS
3. ALL_DEPENDENCIES
4. ALL_PROGRAMS

- The tablespace information on a database server is as follows:

```
SELECT tablespace_name
FROM DBA_TABLESPACES
/
```

```
TABLESPACE_NAME
-----
SYSTEM
UNDOTBS1
TEMP
USERS
EXAMPLE
```

You execute the following command in the session:

```
SQL> ALTER SESSION SET PLSCOPE_SETTINGS = 'IDENTIFIERS:ALL';
Session altered.
```

Identify the correct statements:

1. The identifier information would be captured by PL/Scope for the program created or compiled in the session.
 2. The identifier information would not be captured by PL/Scope as `IDENTIFIERS:ALL` can be enabled only at the `SYSTEM` level.
 3. The identifier information would be captured by PL/Scope only for the programs that are created in the session.
 4. The identifier information would not be captured by PL/Scope since the `SYSAUX` tablespace is not available.
- The parameters specified in `DBMS_METADATA` are case-sensitive:
 1. True
 2. False
 - `DBMS_UTILITY.FORMAT_CALL_STACK` accomplishes which of the following objectives:
 1. Captures exceptions in a PL/SQL block.
 2. Prepares the stack of sequential calls.
 3. Prepares the stack of execution actions.
 4. Prepares the stack of block profiler.
 - Which of the following does the `DBMS_METADATA` package achieve:
 1. Generates report of invalidated objects in a schema.
 2. Generates DDL for a given or all object(s) in a schema.
 3. Generates an object-to-table dependency report in a schema
 4. Generates a report of object statistics in a schema
 - The PL/Scope tool can store identifier data only in the `USERS` tablespace.
 1. True
 2. False
 - Which of the following are the valid parameter values of `SET_TRANSFORM_PARAM` for tables alone?
 1. `STORAGE`
 2. `FORCE`
 3. `PRETTY`
 4. `INHERIT`

11

Safeguarding PL/SQL Code against SQL injection

An incidence of a security breach involves: a hacker and a vulnerable system. A hacker can be an insider or an outsider, who attacks the system to expose and access confidential information, which may lead to fatal consequences. A system could be vulnerable to attacks because of low coding standards and a half-baked understanding of technologies. The steep growth of web application users and sharp rise in social media interactions has widened the attack surface area. The systems that are a hacker's paradise are those which contain personal identifiable information, financial information, government data, and business transactions. The hazardous consequences of a security breach have pushed many organizations to look seriously after data security. As a first layer of protection, organizations must bolt the network penetration through the adoption of products like **Audit Vault and Database Firewall (AVDF)** and protect data access through strong data access policies, encryption or redaction.

SQL injection is a technique to break through the application design and extract sensitive data. In 1998, an author by pen name **Rain Forest Puppy (rfp)** first identified the technology vulnerabilities in his paper *NT Web Technology Vulnerabilities* for Phrack magazine and evangelized the best practices of writing code to dilute such acts. The chapter outline is as follows:

- What is SQL injection?
- Preventing SQL injection attacks
- Testing the code for SQL injection flaws

What is SQL injection?

A database application on the server side contains the programmable logic embedded within the PL/SQL packages and subprograms. These PL / SQL program units may contain SQL statements, which are intended to perform specific operations. The SQL statements, whose query text is built at runtime (dynamically derived) and based on client-supplied inputs, open ways for SQL injection. A malicious user can supply a manipulated input that can break through the PL/SQL program logic by replacing the SQL syntax and perform arbitrary execution.

The reason it is known as **Injection** is because the manipulated text, which replaces or appends to the original SQL text in a PL/SQL program unit, is parsed along with the original SQL statement. The undetected attacker's code is legally executed by the SQL engine, along with the original programmed SQL.

For example, a string type malicious input from the client is executed as legal code by the SQL engine; thus, exploiting a server-side SQL statement. As a consequence, an attacker can gain access to sensitive and restricted information from the database. The data, considered to be attack-prone, can be personal information, credit card information, an organization's internal data, or government data.

Although there are multiple platform targets, techniques and remedies for SQL injection, this chapter restricts the scope of discussion to the attacks on PL/SQL units with the best practices to avoid them.

SQL injection targets

In many of the cases reported to date, the client-supplied inputs were identified as the major cause of SQL injection attacks. The dynamic SQL, which works directly with the client-supplied inputs, is more prone to injection attacks. After the code vulnerabilities are discovered, an attacker can exploit the code to any extent. By extracting restricted data, they can update or delete sensitive information. Not only the data modification, data definitions can also be modified through SQL injection.

Apart from the client-supplied inputs, a definer PL/SQL program unit can indirectly lead to code injection because it executes with the privileges of the unit owner and not the invoker. Therefore, an invoker can execute a PL/SQL program with an elevated set of privileges.

How to exploit the PL/SQL code?

There are numerous forms of SQL injection. An attacker can create a function to update sensitive information with `PRAGMA AUTONOMOUS_TRANSACTION`. He can gain access to the DBA privileges and make irrevocable changes to the database.

Let's understand the SQL injection through a credit card case study. The following `CREATE TABLE` statement creates an `EMP_CREDIT_BAL` table, which contains the credit card details of employees:

```
/*Test table to generate credit card details of employees */
CREATE TABLE emp_credit_bal
AS
SELECT empno card_holder,
       deptno,
       ename,
       job,
       sal,
       REGEXP_REPLACE(str, '(\d{4}) (\d{4}) (\d{4}) (\d{4}) ', '\1-\2-\3-\4') card_
no,
       TRUNC(DBMS_RANDOM.VALUE(sal, 5*sal)) credit_balance
FROM emp E ,
     (SELECT TO_CHAR(
           TRUNC(
             DBMS_RANDOM.VALUE(10000000000000000,
                               9999999999999999)) str
         FROM dual
        )
      /

/*Verify the data in the table */
SELECT ename,
       card_no,
       credit_balance
FROM emp_credit_bal
/
```

ENAME	CARD_NO	CREDIT_BALANCE
SMITH	4020-3009-1084-0345	2844
ALLEN	9205-3519-4278-7993	2034
WARD	7838-5801-5970-8060	4038
JONES	7556-7454-2248-0850	4336
MARTIN	6689-9703-5168-5282	1690
BLAKE	3662-7700-4489-3187	5443

CLARK	6195-2171-8232-3240	11152
SCOTT	1961-7153-0620-9366	14337
KING	7930-1567-7029-0943	19497
TURNER	8166-4431-5647-3579	2286
ADAMS	8155-2229-9107-3634	2301
JAMES	4788-5716-1863-6983	1540
FORD	8683-1130-1302-1498	9750
MILLER	5522-2843-9553-5188	3585

14 rows selected.

A `P_REP_CC_BAL` procedure is a low standard program that accepts the last four digits of the credit card and prints the credit balance report of the employee:

```
/*Procedure to print the credit card info of a card holder */
CREATE OR REPLACE PROCEDURE p_rep_cc_bal (p_card_no VARCHAR2)
IS

    /*Declare ref cursor variable */
    TYPE c_ref IS REF CURSOR;
    cc_bal C_REF;

    /*Declare a record of EMP_CREDIT_CARD row type */
    emp_cc_rec emp_credit_bal%rowtype;
BEGIN

    /*Open the ref cursor variable for a SELECT query */
    OPEN cc_bal FOR 'SELECT * FROM emp_credit_bal
                    WHERE substr(card_no,-4,4)='||p_card_no;

    LOOP

        /*Iterate through the result set to fetch the record */
        FETCH cc_bal INTO emp_cc_rec;
        EXIT WHEN cc_bal%NOTFOUND;

        /*Print the credit card information */
        DBMS_OUTPUT.PUT_LINE (RPAD('_',50,'_'));
        DBMS_OUTPUT.PUT_LINE ('Emp Name/Title:'||emp_cc_rec.ename||' |
' || INITCAP(emp_cc_rec.job));
        DBMS_OUTPUT.PUT_LINE ('Emp CCard:'||emp_cc_rec.card_no);
        DBMS_OUTPUT.PUT_LINE ('CCard Balance:'||emp_cc_rec.credit_balance);
    END LOOP;
    CLOSE cc_bal;
END;
/
/*Enable the SERVEROUTPUT to print the program output */
SET SERVEROUTPUT ON
```

Let's test the preceding procedure with a sample input:

```
EXEC p_rep_cc_bal('0345');
```

```
Emp Name/Title:SMITH | Clerk
Emp CCard:4020-3009-1084-0345
CCard Balance:2844
```

PL/SQL procedure successfully completed.

Let's check how a hacker can tweak the string input to extract the details of the PRESIDENT:

```
EXEC p_rep_cc_bal(''XXX'' OR JOB=''PRESIDENT'');
```

```
Emp Name/Title:KING | President
Emp CCard:7930-1567-7029-0943
CCard Balance:19497
```

PL/SQL procedure successfully completed.

What we have just seen is how a manipulated input bypasses the procedural logic by supplying a manipulated value to the existing predicate(s). It appends a new predicate to the SQL that fetches the president's credit details. The following procedure call fetches the credit card details of employees from department 10:

```
EXEC p_rep_cc_bal(''XXX'' OR DEPTNO=10');
```

```
Emp Name/Title:CLARK | Manager
Emp CCard:6195-2171-8232-3240
CCard Balance:11152
```

```
Emp Name/Title:KING | President
Emp CCard:7930-1567-7029-0943
CCard Balance:19497
```

```
Emp Name/Title:MILLER | Clerk
Emp CCard:5522-2843-9553-5188
CCard Balance:3585
```

PL/SQL procedure successfully completed.

Likewise, an attacker can generate the report for all the employees by bypassing all the filters and predicates:

```
EXEC p_rep_cc_bal(''XXX'' OR 1=1');
```

```
Emp Name/Title:SMITH | Clerk
```

```
Emp CCard:4020-3009-1084-0345
```

```
CCard Balance:2844
```

```
Emp Name/Title:ALLEN | Salesman
```

```
Emp CCard:9205-3519-4278-7993
```

```
CCard Balance:2034
```

```
Emp Name/Title:WARD | Salesman
```

```
Emp CCard:7838-5801-5970-8060
```

```
CCard Balance:4038
```

```
Emp Name/Title:JONES | Manager
```

```
Emp CCard:7556-7454-2248-0850
```

```
CCard Balance:4336
```

```
Emp Name/Title:MARTIN | Salesman
```

```
Emp CCard:6689-9703-5168-5282
```

```
CCard Balance:1690
```

```
Emp Name/Title:BLAKE | Manager
```

```
Emp CCard:3662-7700-4489-3187
```

```
CCard Balance:5443
```

```
Emp Name/Title:CLARK | Manager
```

```
Emp CCard:6195-2171-8232-3240
```

```
CCard Balance:11152
```

```
Emp Name/Title:SCOTT | Analyst
```

```
Emp CCard:1961-7153-0620-9366
```

```
CCard Balance:14337
```

```
Emp Name/Title:KING | President
```

```
Emp CCard:7930-1567-7029-0943
```

```
CCard Balance:19497
```

```
Emp Name/Title:TURNER | Salesman
```

```
Emp CCard:8166-4431-5647-3579
```

```
CCard Balance:2286
```

```
Emp Name/Title:ADAMS | Clerk
Emp CCard:8155-2229-9107-3634
CCard Balance:2301
```

```
Emp Name/Title:JAMES | Clerk
Emp CCard:4788-5716-1863-6983
CCard Balance:1540
```

```
Emp Name/Title:FORD | Analyst
Emp CCard:8683-1130-1302-1498
CCard Balance:9750
```

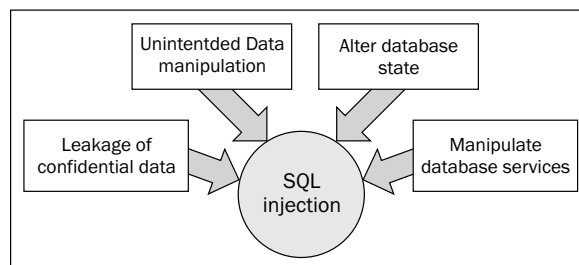
```
Emp Name/Title:MILLER | Clerk
Emp CCard:5522-2843-9553-5188
CCard Balance:3585
```

PL/SQL procedure successfully completed.

Now, you can realize how the good old arguments can be exploited to leak confidential information. A badly coded procedure, which is not too uncommon, is vulnerable to such attacks.

The preceding case study is a classic example of a **First Order Attack**, where the data loses its confidentiality after it is attacked. If the data attack does not happen through program execution, it is called a **Second Order Attack**.

The following diagram branches the impacts of SQL injection:



The impact of SQL injection

Preventing SQL injection attacks

SQL injection is not a design bug, but an intentional malicious practice. Database developers must adopt best practices while writing PL/SQL code. If code modification is not possible, the application interface layer may also work to reduce the attack surface area.

Let's take a brief look at the precautionary measures to minimize injection attacks:

- **Check your dynamic SQL:** Dynamic SQL query text, which is constructed at runtime and directly uses the user-supplied inputs, creates a pregnable hitch point in the application. You can protect them against injection attacks through either of these techniques:
 - Reduce the direct exposure of client inputs to dynamic SQL: You can sanitize and validate the client-supplied inputs before they are used in dynamic SQL. Oracle provides the `DBMS_ASSERT` package to verify the inputs.
 - Use bind arguments in dynamic SQL: Database developers are encouraged to make use of bind arguments for multiple reasons and one of them is security. Bind arguments nearly eliminate the possibility of attacks. Manipulated argument values would end in an exception, thereby terminating the program execution.
- **Monitor a user's object privileges:** Irregularity in object privileges can be a potential threat to an application. A user can invoke a definer's program unit which may be at a higher privilege level. From a security standpoint, a user at a lower privilege level can gain access to unauthorized datasets objects. The use of the invoker's program units should be encouraged to regulate object privileges. Oracle Database 12c enhances the security of the definer's program units by allowing the granting of roles to program units.

Sanitizing inputs using `DBMS_ASSERT`

Oracle 10g Release 2 introduced the `DBMS_ASSERT` package to validate user inputs before they are consumed by the server-side program units. The package asserts or postulates an input for a certain fact such as quoted and unquoted identifiers, or object validity, and so on. Upon assertion, the input is returned as the actual instance. If the assertion fails, the `VALUE_ERROR` exception will be raised.

The `DBMS_ASSERT` package is owned by `SYS` and contains seven packaged functions. The package subprograms are demonstrated as follows:

Subprograms	Description
<code>ENQUOTE_LITERAL</code> function	Encloses a string literal within single quotes
<code>ENQUOTE_NAME</code> function	Encloses the input string in double quotes
<code>NOOP</code> functions	An overloaded function returns the value without any checking; does no operation

QUALIFIED_SQL_NAME function	<ul style="list-style-type: none"> • Verifies if the input string is a qualified SQL name • Raises ORA-44004 if the input string is not a valid qualified SQL name
SCHEMA_NAME function	<ul style="list-style-type: none"> • Verifies if the input string is a valid schema name • Raises ORA-44001 if the input string is an invalid schema name
SIMPLE_SQL_NAME function	<ul style="list-style-type: none"> • Verifies if the input string is a simple identifier (quoted and unquoted) • Raises ORA-44003 if the input string is an invalid SQL name
SQL_OBJECT_NAME function	<ul style="list-style-type: none"> • Verifies if the input parameter string is a valid object in the database • Raises ORA-44002 if the input string is an invalid object name

Choose the right subprogram for the right identifier

The DBMS_ASSERT subprogram aims to verify the properties of identifiers and literals. An identifier is used to denote each and every item used in the database. All the object names or variable names in a PL/SQL block are identifiers. An identifier can be quoted, unquoted, or literal.

Unquoted identifiers

This identifier abides by the naming convention of the Oracle Database: it must begin with an alphabet followed by numbers or a set of defined special characters (_). The verification algorithm of unquoted identifiers is basic and it checks whether the identifier follows a proper naming convention.

You can use SIMPLE_SQL_NAME to check whether an unquoted identifier contains any nonadmissible characters and starts with an alphabet.

Note that it doesn't verify the validity status of the input identifier.

In the following example, SIMPLE_SQL_NAME confirms the input identifier as it correctly follows the naming convention:

```

/*Verify an unquoted identifier using SIMPLE_SQL_NAME*/
SELECT DBMS_ASSERT.SIMPLE_SQL_NAME('emp_credit_bal')
FROM DUAL
/

DBMS_ASSERT.SIMPLE_SQL_NAME('EMP_CREDIT_BAL')
-----
emp_credit_bal

```

Any violation in the naming convention would throw an exception:

```
/*Verify an unquoted identifier using SIMPLE_SQL_NAME*/
SELECT DBMS_ASSERT.SIMPLE_SQL_NAME('1emp_credit_bal')
FROM DUAL
/

*
ERROR at line 1:
ORA-44003: invalid SQL name
ORA-06512: at "SYS.DBMS_ASSERT", line 206
```

An identifier with more than one SQL name should be validated through `QUALIFIED_SQL_NAME`. A qualifier may include a schema name, an object name, or a database link:

```
/*Verify an unquoted qualifier using SIMPLE_SQL_NAME*/
SELECT DBMS_ASSERT.SIMPLE_SQL_NAME('scott.emp_credit_bal')
FROM DUAL
/

SELECT DBMS_ASSERT.SIMPLE_SQL_NAME('scott.emp_credit_bal') FROM
DUAL

*
ERROR at line 1:
ORA-44003: invalid SQL name
ORA-06512: at "SYS.DBMS_ASSERT", line 206

/*Verify an unquoted qualifier using QUALIFIED_SQL_NAME*/
SELECT DBMS_ASSERT.QUALIFIED_SQL_NAME('scott.emp_credit_bal') FROM
DUAL
/

DBMS_ASSERT.QUALIFIED_SQL_NAME('SCOTT.EMP_CREDIT_BAL')
-----
scott.emp_credit_bal
```

Schema names and object names can be distinctly verified using the `SCHEMA_NAME` and `SQL_OBJECT_NAME` subprograms of `DBMS_ASSERT`. For example, the following `SELECT` query verifies the schema name and object name before sanitizing the qualifier:

```
/*Verify an unquoted qualifier using QUALIFIED_SQL_NAME*/
SELECT DBMS_ASSERT.QUALIFIED_SQL_NAME(
    DBMS_ASSERT.SCHEMA_NAME('SCOTT') || '.' ||
    DBMS_ASSERT.SQL_OBJECT_NAME('emp_credit_bal')) obj
FROM DUAL
/
```

```

OBJ
-----
SCOTT.emp_credit_bal

```



SCHEMA_NAME is a case-sensitive subprogram. A lower case string input will not be verified as a schema name.

Quoted identifiers

A quoted identifier is always enclosed within double quotes and follows no naming convention. It may start with a number or even contain any special character. You can use SIMPLE_SQL_NAME to check the sanity of quoted identifiers:

```

/*Unquoted identifier with special characters raise exception */
SELECT DBMS_ASSERT.SIMPLE_SQL_NAME('***emp_credit_bal***')
FROM DUAL
/

ERROR at line 1:
ORA-44003: invalid SQL name
ORA-06512: at "SYS.DBMS_ASSERT", line 206

/*Quoted identifier with special characters return value */
SELECT DBMS_ASSERT.SIMPLE_SQL_NAME('***emp_credit_bal***') FROM DUAL
/
DBMS_ASSERT.SIMPLE_SQL_NAME('***EMP_CREDIT_BAL***')
-----
"***emp_credit_bal***"

```

You can also use the ENQUOTE_NAME subprogram to enclose a constant value or identifier in double quotes. Quoted identifiers, which are qualifiers, can be verified using the QUALIFIED_SQL_NAME subprogram.



The Oracle-supplied DBMS_UTILITY.NAME_TOKENIZE subprogram helps in differentiating a simple SQL name from a qualified SQL name.

Literals

A literal can be any fixed constant used in a SQL query or a PL/SQL program:

```

/*Demonstrate the use of a literal in a SELECT query */
SELECT *
FROM emp

```

```
WHERE ename = 'KING'
/
/*Use (EMP) as a literal*/
SELECT *
FROM user_tables
WHERE table_name='EMP'
/
```

While working with literals, you can use `ENQUOTE_LITERAL` to sanitize the client-supplied inputs and avoid additional predicates or the union `SELECT` query. Let's look at a user input that was used earlier to inject and extract the credit card information of employees:

```
/*Sanitize the input before supplying it to procedure call*/
DECLARE
  l_cc VARCHAR2 (4000) := ''XXX' OR DEPTNO=10';
BEGIN
  /*Sanitize using DBMS_ASSERT */
  l_cc := DBMS_ASSERT.ENQUOTE_LITERAL(l_cc);
  p_rep_cc_bal (l_cc);
END;
/

DECLARE
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error
ORA-06512: at "SYS.DBMS_ASSERT", line 409
ORA-06512: at "SYS.DBMS_ASSERT", line 493
ORA-06512: at line 4
```

The preceding block throws an exception because the client input could not be verified by the `DBMS_ASSERT` package.

DBMS_ASSERT – limitations

The limitations of the `DBMS_ASSERT` package are listed as follows:

- No validation for the `TNS` connection strings.
- No validation for string lengths or buffer overflow attacks.
- No validation for the validity of SQL identifiers.
- No validation for object privileges or the unintended use of privileges. For privilege evaluation, you can use the Oracle Database 12c privilege analysis feature to determine whether to retain or revoke object privileges.

Use of bind variables to prevent injection attacks

Using bind variables is a good programming practice. From a performance standpoint, bind variables eliminate hard parses of SQL statements and improve the performance of a PL/SQL block. From a security standpoint, a bind variable avoids the concatenation of literal values to a dynamic SQL statement.

Let's revisit the credit card case study and rewrite the procedure with the help of bind variables. Note the use of the `B_CARD_NO` bind variable in the `ref` cursor:

```
/*Re-write the procedure P_REP_CC_BAL to use bind variables */
CREATE OR REPLACE PROCEDURE p_rep_cc_bal (p_card_no VARCHAR2)
IS

    /*Declare ref cursor variable */
    TYPE c_ref IS REF CURSOR;
    cc_bal C_REF;

    /*Declare record structure of EMP_CREDITBAL row type */
    emp_cc_rec emp_credit_bal%rowtype;
BEGIN

    /*Open the ref cursor variable with a bind variable */

    OPEN cc_bal FOR 'SELECT * FROM emp_credit_bal
                     WHERE substr(card_no,-4,4)=:b_card_no'
                     USING p_card_no;

    LOOP

        /*Iterate the result set and print the credit card info */
        FETCH cc_bal INTO emp_cc_rec;
        EXIT WHEN cc_bal%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE (RPAD('_',50,'_'));
        DBMS_OUTPUT.PUT_LINE ('Emp Name/Title: ' || emp_cc_rec.ename || ' | ' |
                               ' || INITCAP(emp_cc_rec.job));
        DBMS_OUTPUT.PUT_LINE ('Emp CCard: ' || emp_cc_rec.card_no);
        DBMS_OUTPUT.PUT_LINE ('CCard Balance: ' || emp_cc_rec.credit_balance);
    END LOOP;
    CLOSE cc_bal;
END;
/
SET SERVEROUTPUT ON
```


You can check the procedure output with a test input:

```
/*Verify with Input-1 */  
EXEC p_rep_cc_bal('0345');
```

```
Emp Name/Title:SMITH | Clerk  
Emp CCard:4020-3009-1084-0345  
CCard Balance:2844
```

PL/SQL procedure successfully completed.

Let's invoke the procedure with the manipulated inputs, which were earlier used to access restricted data:

```
/*Verify with Input-2 */  
EXEC p_rep_cc_bal(''XXX'' OR JOB='''PRESIDENT''');
```

PL/SQL procedure successfully completed.

```
/*Verify with Input-3 */  
EXEC p_rep_cc_bal(''XXX'' OR DEPTNO=10');
```

PL/SQL procedure successfully completed.

```
/*Verify with Input-4 */  
EXEC p_rep_cc_bal(''XXX'' OR 1=1');
```

PL/SQL procedure successfully completed.

Well, none of them returned an output, because the bind variable value or the tweaked input didn't create a valid SQL cursor. Therefore, it could not be executed.

Best practices to avoid SQL injection

There are multiple best practices to mitigate the effect of SQL injection. The objective of these best practices is to reduce the attack surface area by covering the vulnerable areas of the database code. The listed best practices are safe to follow and easy to remember:

- Avoid dynamically constructed SQL query text in the PL/SQL program units. Use static SQL wherever possible, as it avoids the code vulnerability. With dynamic SQL, bind variables should be used.
- Expose database program units to the client through the API units only.
- Monitor and control the object privileges carefully before granting the EXECUTE privilege on an object to a user.

- Encourage the use of the `DBMS_ASSERT` subprogram in policing; not just for user inputs, but also dynamic SQL texts and placeholder values in a SQL statement. Application clients can also programmatically restrict arbitrary inputs and allow only the expected ones.
- Examine and evaluate the use of `PUBLIC` privileges.
- Encourage the adoption of data security solutions such as encryption, data redaction, database vault, audit vault, and database firewall.

Testing the code for SQL injection flaws

So far, we have discussed the symptoms and remedies of SQL injection. We demonstrated the programming recommendations to mitigate the effects of code injections and smuggles. Assuring code quality during the testing cycle plays a crucial role towards preventing code attacks. The code testing resources must adopt a concrete strategy to discover and hit upon the code's vulnerabilities before it invites an attacker to exploit the database. Now, we will discuss some of the testing considerations to test the code for SQL injection flaws.

Test strategy

A logical and efficient test strategy must be employed to discover injection flaws. Of course, there is no magic bullet to filter out all the vulnerabilities of a piece of code.

Usual code reviews are a part of static testing while testing programs with sample data and inputs come under dynamic testing. These days, static testing has been absorbed into the development stage, where developers and their peers and seniors review the code. Major syntactical errors, logical issues, code practices, and injection bugs can be traced at this level. The Dry Run concept can even check multiple scenarios and ensure bug-free application submission to the quality assurance team.

An effective code review

As a code reviewer, the first and foremost step is to measure the attack surface area. The code reviewer must verify the exposure of the database program units in the application interface layer. In addition, they must check the privileges available to the database users. Once these steps are passed with the right justification, the PL/SQL code can be reviewed to identify the vulnerable areas. In PL/SQL-based applications, always be careful to look for:

- Dynamic SQL using:
 - `EXECUTE IMMEDIATE`

- REF CURSOR queries
- DBMS_SQL
- DBMS_SYS_SQL
- Check for the appropriate usage of bind arguments
- Parameter sanitization using DBMS_ASSERT

Similarly, in a Java or C client architecture, the reviewer must look for dynamic callable statement preparation.

Static code analysis

SQL injection attacks are mostly due to coding unawareness and dynamic SQL. Therefore, static code analyzers cannot easily trace an application's vulnerability. The Oracle documentation defines static code analysis as follows:

Static code analysis is the analysis of computer software that is performed without executing programs built from that software. In most cases, the analysis is performed on some version of the source code and in other cases, some form of the object code. The term is usually applied to the analysis performed by an automated tool.

It is advisable that such analysis tools should not be considered as the testing benchmark and confirmatory tools. Instead, they can be used for white box testing, where the application is tested for a smooth logical flow and the program executions for different nature of input data.

Fuzz tools

Fuzz testing is a rough testing that is not based on any preset logic or use case. It measures the application's sustainability against junk and malicious inputs. Without any preconception of the system or program behavior, it uses raw inputs to check the program semantics. The environment for fuzz testing tools can be explicitly made by modifying the context values and manipulating the test data to all odds.

The bugs reported in fuzz testing may not always be real threats to the application, but they may provide a clue to the vulnerability attacks.

Generating test cases

The last and the final call is the preparation of test cases. Although this is skipped during the development stage, the test cases serve as the proof of testing at later stages. Test cases can measure the robustness of database programs, application security, and the validation of client inputs.

Summary

In this chapter, we learned about a malicious hacking concept – SQL injection. We discussed the causes of a code attack and its impact on the database. We covered the techniques to safeguard an application against the injection attacks through demonstrations and illustrations. At the end of the chapter, we discussed some of the testing considerations to expose the vulnerable areas in the code.

Practice exercise

- Which method would you employ to protect the PL/SQL code against SQL injection attacks?
 1. Replace Dynamic SQLs with Static SQLs.
 2. Replace concatenated inputs in Dynamic SQL with bind arguments.
 3. Declare the PL/SQL program to be executed by its invoker's rights.
 4. Remove string type parameters from the procedure.
- You should use static SQL to avoid SQL injection when all Oracle identifiers are known at the time of code execution.
 1. True.
 2. False.
- Choose the impact of SQL injection attacks:
 1. Malicious string inputs can extract confidential information.
 2. Unauthorized access can drop a database.
 3. It can insert the ORDER data in to the EMPLOYEES table.
 4. A procedure executed by owners, (SYS) rights can change the password of a user.

- Pick the correct strategies to fight against SQL injection:
 1. Sanitize the malicious inputs from the application layer with DBMS_ASSERT.
 2. Remove string concatenated inputs from the Oracle subprogram.
 3. Dynamic SQL should be removed from the stage.
 4. Execute a PL/SQL program with its creator's rights.
- Statistical code analysis provides an efficient technique to trace the application's vulnerability by using ideal and expected parameter values.
 1. True.
 2. False.
- The fuzz tool technique is a harsh and rigorous format of testing which uses raw inputs and checks a programs' sanctity.
 1. True.
 2. False.
- Choose the objectives that can be addressed by the DBMS_ASSERT package to prevent SQL injection
 1. Enclose a given string in single quotes.
 2. Enclose a given string in double quotes.
 3. Verify a schema object name.
 4. Verify a simple SQL and qualified SQL identifier.
- Identify the nature of the table name in the following SELECT statement:

```
SELECT TOTAL
FROM "ORDERS"
WHERE ORD_ID = P_ORDID
/
```

 1. A unquoted identifier.
 2. A quoted identifier.
 3. A literal.
 4. A placeholder.

- Which of the following DBMS_ASSERT subprograms modifies the input value?
 1. SIMPLE_SQL_NAME.
 2. ENQUOTE_LITERAL.
 3. QUALIFIED_SQL_NAME.
 4. NOOP.
- The code reviews must identify certain vulnerable key areas for SQL injection. Select the correct ones from the following list:
 1. DBMS_SQL.
 2. BULK COLLECT.
 3. EXECUTE IMMEDIATE.
 4. REF CURSOR.
- The AUTHID CURRENT_USER clause achieves which of the following purposes?
 1. The code executes with the invoker's rights.
 2. The code executes with the current logged in user.
 3. It eliminates SQL injection vulnerabilities.
 4. The code executes with the creator's rights.

12

Working with Oracle SQL Developer

SQL Developer is an integrated development environment from Oracle that serves as a one-stop solution for major Oracle Database activities, thereby standardizing the development tasks and enhancing the productivity of database professionals. As a free **graphical user interface (GUI)** tool, SQL Developer offers a wide set of features and solutions for database development, administration, and management, in on-premise and cloud deployments. The SQL Developer tool supports Oracle Database 10g, 11g, and 12c.

For database developers, SQL Developer offers rich editors for working with core and advanced functionalities, such as execute, debug, and test, using SQL, PL/SQL, XML, Stored Java procedures, query tuning, and much more. Database administrators can use SQL Developer to perform various operations like data export and import via data pump, database backup and recovery tasks via RMAN, resource monitor and management, performance analysis and diagnosis, use data modeler and third-party migration.

In this chapter, we will describe the benefits of SQL Developer, for database administrators, and architects. Note that the chapter does not intend to provide a step-by-step tutorial for various tool operations. The outline of the chapter is as follows:

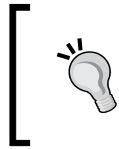
- Overview of SQL Developer
 - Key differentiators
 - History and background
 - SQL Developer for DBA
 - SQL Developer for Developers

- Getting started with SQL Developer
- New features of SQL Developer 4.0 and 4.1

An overview of SQL Developer

SQL Developer is a free integrated development environment from Oracle that allows the database community to develop PL/SQL applications and, perform administrative tasks, and enables management activities such as data modeling, versioning, scripting, and third-party migration. The latest version of SQL Developer (as of the time of writing this book) is 4.1.1.

The tool is built using Java (on JDeveloper Framework), which enables it to run on Windows, Linux, and Mac OS X. By default, it uses the thin JDBC (Type IV Java driver) driver to connect to the database, thus reducing the requirements of any further database clients. In addition, the extensible architecture of SQL Developer allows the developer community to incorporate their own custom extensions in to the tool. You can leverage Oracle JDeveloper's Extension **SDK (Software Development Kit)** to develop and add the required functionality as an extension to the SQL Developer tool.



Visit the SQL Developer Extensions Exchange (<http://www.oracle.com/technetwork/developer-tools/sql-developer/extensions-083825.html>) to look for extensions that were built within and outside Oracle.

Key differentiators

The following are the key factors that differentiate SQL Developer from other contemporary tools:

- SQL Developer is a free tool and shipped along with the Oracle Database software. No separate installation is required.
- Oracle support is available for the customers who have licensed the Oracle Database.
- It supports traditional on-premise as well as cloud deployments of Oracle Database.
- It supports database products such as Data Miner, Times Ten, Spatial, and Graph.
- It is developer-rich and administrator-friendly too. The DBA console provides a wide set of features for administrative tasks.

- There are pre-built reports for providing information about the database management, database schemas, and performance. There is provision for creating custom reports and creating multilevel with predefined reports.
- There is logical, relational, and physical modelling enabled through the Data Modeler extension.
- It has the ability to connect to not just the Oracle Database, but also non-Oracle databases such as SQL server, Sybase, and DB2.
- Build on the JDeveloper framework. There is provision for including custom-built extensions in the tool.
- There is a wide community of SQL Developer users who interact through various channels, such as Oracle Technology Network forums, developer communities, and blogs. Users are encouraged to participate in the Oracle SQL Developer Exchange program (<https://sqldeveloper.oracle.com>), where you can register a feature request and submit code snippets or tooltips.

History and background

The first release of SQL Developer was made in March 2006. The initial release included the basic functionalities, such as executing SQL queries and PL/SQL blocks, and invoking SQL* Plus scripts. Thereafter, there were a few quick patches and an important release of SQL Developer 1.1 was made in December 2006. Since then, the tool has evolved with a broad spectrum of features and support for database options.

Let's take a quick glance at the earlier releases of the SQL Developer tool:

SQL Developer release version	Release date	Key new features
SQL Developer 1.0	March 2006	<ul style="list-style-type: none"> • Initial release enabled the execution of SQL and PL/SQL statements
SQL Developer 1.1	December 2006	<ul style="list-style-type: none"> • File-based PL/SQL editing. • Patches SQL Developer 1.1.1 (Jan 2007), SQL Developer 1.1.2 (March 2007), and SQL Developer 1.1.3 (May 2007)
SQL Developer 1.2	June 2007	<ul style="list-style-type: none"> • Developer migration workbench • Support for Oracle Application Express 3.0.1
SQL Developer 1.5	April 2008	<ul style="list-style-type: none"> • Support for version control

SQL Developer release version	Release date	Key new features
SQL Developer 2.1	December 2009	<ul style="list-style-type: none">• Third-party database migration support• PL/SQL unit testing• Integrated data modeler viewer
SQL Developer 2.1.1	March 2010	<ul style="list-style-type: none">• Patch release
SQL Developer 3.0	March 2011	<ul style="list-style-type: none">• Query builder• DBA navigator• Schedule builder
SQL Developer 3.1	February 2012	<ul style="list-style-type: none">• Support for Recovery Manager (RMAN)• Support for Data Pump• PDF reporting
SQL Developer 3.2	November 2012	<ul style="list-style-type: none">• Support for managing the APEX listener• Improved Database Diff and DB DOC features
SQL Developer 4.0	December 2013	<ul style="list-style-type: none">• Support for Oracle Database 12c Multitenant and Database as a Service platform• Stable release with updated framework
SQL Developer 4.1	May 2015	<ul style="list-style-type: none">• Optimal memory footprint• Requirement for Java 7 JDK• Support subversion 1.7

SQL Developer for Developers

SQL Developer provides a powerful editor for running and analyzing SQL queries and developing PL/SQL programs. As a database developer, you can browse schema objects, view object attributes, and compile stored subprograms. You can run a SQL query (or invoke a script), retrieve the result, export it into an XML, XLS, or PDF, and examine the query explain plan. You can create, edit, debug, compile, or drop stored PL/SQL subprograms.

SQL Developer for Database Administrators

In its initial days, SQL Developer was a developer-centric tool. However, the later releases of the tool broadened its administrative offerings. Database administrators can perform their routine database activities including storage management, backup and recovery operations, data export and import, database auditing, diagnostics pack features, and database resource management.

The latest version of SQL Developer supports pluggable database operations in a multitenant container database. You can create, drop, unplug, plug, or clone, a pluggable database.

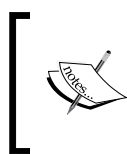
The **Manage Database** option in the connection tree allows a user with SYSDBA privilege to report the usage of database tablespaces. Besides, there are a variety of pre-defined Database Administration reports, which furnish a great deal of information from a database-management standpoint. A new dedicated DBA panel is available for database administration activities. Existing DBA connections can be copied from the connection tree to the DBA panel and can be used to manage the Oracle Database.

Not only for developers and administrators, SQL Developer successfully connects the architect community as well. Database architects can perform data modeling with the **SDDM (SQL Developer Data Modeler)** module, integrated with SQL Developer.

SQLcl – The new SQL command line

SQLcl is the modernized and rebranded version of the good old SQL* Plus. SQL* Plus has been the primary command-line interface for SQL execution for many years.

Oracle SQL Developer, as we are all aware, provides multiple-user interface features for application developers to format and edit SQL text, export query result in desired file formats, object name completion, maintains SQL history, and does much more. SQLcl, being a command-line interface utility, inherits the developer-friendly formatting and editing elements of SQL worksheet from the SQL Developer. With SQLcl, the developer and user experiences will be improved by an order of magnitude.



You can download the SQLcl - Early Adopter binaries from the downloads page of SQL Developer (<http://www.oracle.com/technetwork/developer-tools/sql-developer/downloads/index.html>).

Currently, the utility is available for standalone download, but in future it will be part of the SQL Developer 4.1 and Oracle Database 12c Release 2 software packages. For installation, you must have **Java Runtime Environment (JRE)** 1.7 or higher on your system.



For the SQLcl overview, watch the YouTube video at <https://www.youtube.com/watch?v=HApdy-o525A>.


Let's briefly discuss some of the key features and commands of SQLcl:

- Database connection: SQLcl supports database connection through EZConnect, TNS from the Oracle client or the `TNS_ADMIN` environment variable, and LDAP. You can store the database connections using the `NET` command.
- Object-name/Command completion: Pressing the *Tab* key while you type an object name will auto-complete the object name in uppercase.
- Multiline edits: Instead of using an editor to make changes to the SQL query, SQLcl allows you to work with the keyboard keys (*Enter*, *Backspace*, forward and backward) to edit the SQL query.
- ALIAS: SQLcl enables you to create an alias for an SQL query or a PL/SQL block. To invoke the alias, simply type the alias name in the SQLcl prompt.
- CD: Similarly to the OS command, you can change the directory from the SQLcl interface by using the `CD` command. This command will prevent SQL developers from including OS paths in the scripts.
- CTAS: This acronym can be used directly in place of `CREATE TABLE AS SELECT` to generate a DDL script of a new table from an existing one. You can transform the command output by setting the transformation parameters using the `DBMS_METADATA` package.
- DDL: This command generates the DDL script for a given object (it optionally writes the output to a file too).
- SQLFORMAT: You can apply different formats to the query results. It may be applying a different color in `ansiconsole`, JSON, XML, CSV, HTML, insert, loader, delimited, or text.
- SQL history: SQLcl stores the last 100 commands executed by all the database users.
- INFORMATION: The new SQLcl command is an advanced version of the `DESCRIBE` command. Besides listing the object structure or a program signature, the `INFO` and `INFO+` commands provide more detailed information.

Getting started with SQL Developer

To start working with the SQL Developer tool, you must follow the following steps:

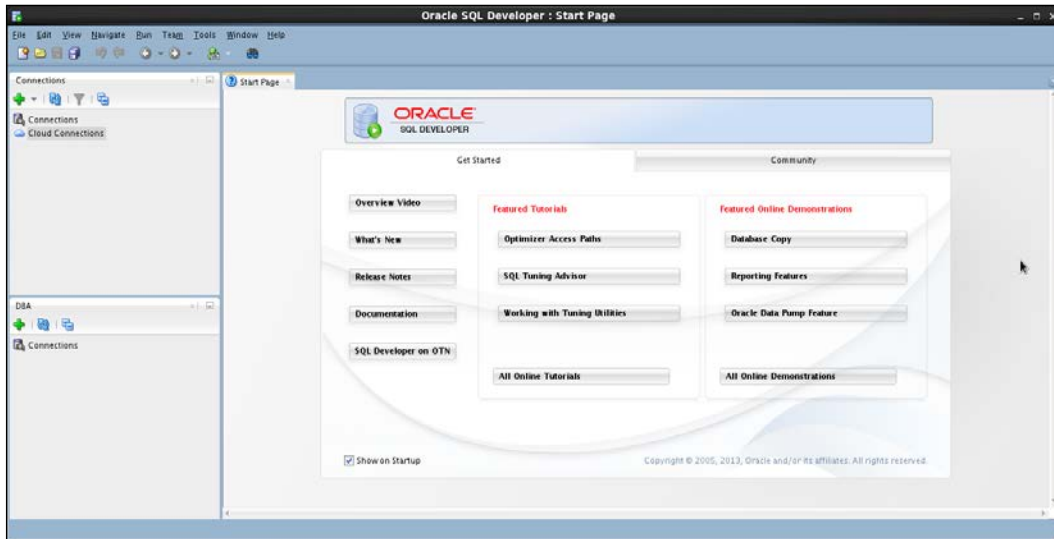
1. Download the latest binaries of the SQL Developer tool from the Oracle Technology Network.

 Go to http://www.oracle.com/technology/products/database/sql_developer for the most recent standalone version of the tool. This tool can be downloaded for free.

2. Alternatively, you may also find the software binaries shipped along with the Oracle Database or Oracle JDeveloper software media.
3. Ensure that the **Java Development Kit (JDK)** is installed on the target system. Usually, the SQL Developer product doesn't include the JDK except on Microsoft Windows. On Windows, a version of product binaries also includes the JDK. After startup, the correct JDK path must be specified for SQL Developer.
4. Run the `sqldeveloper` executable from the unzipped SQL Developer folder. Specify the full path of the JDK:

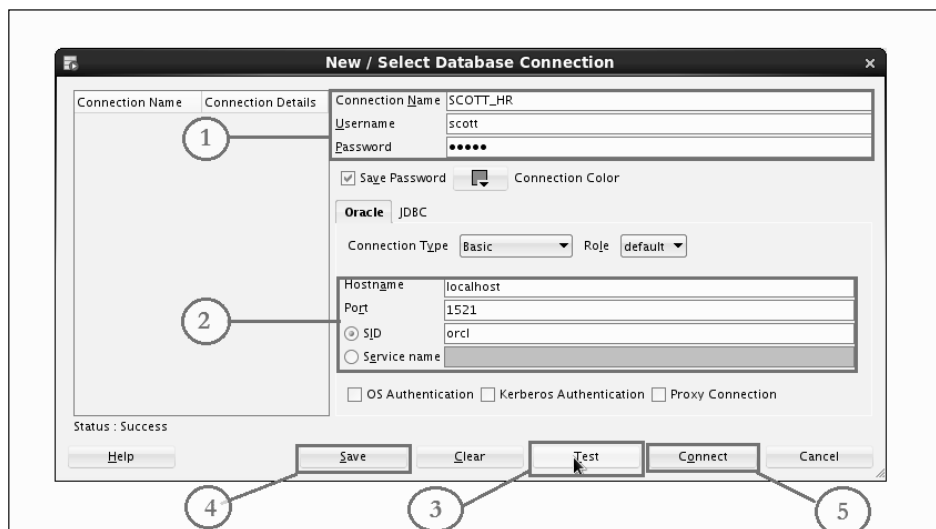


The SQL Developer homepage includes multiple tutorials to help you get started, and it assists you with the key functionalities of the tool:



Creating a database connection

If you know the login credentials of a database user, you can save the connection details within SQL Developer. The login credentials include the username, password, server IP address, and database SID. The following image shows how to create a database connection in SQL Developer:



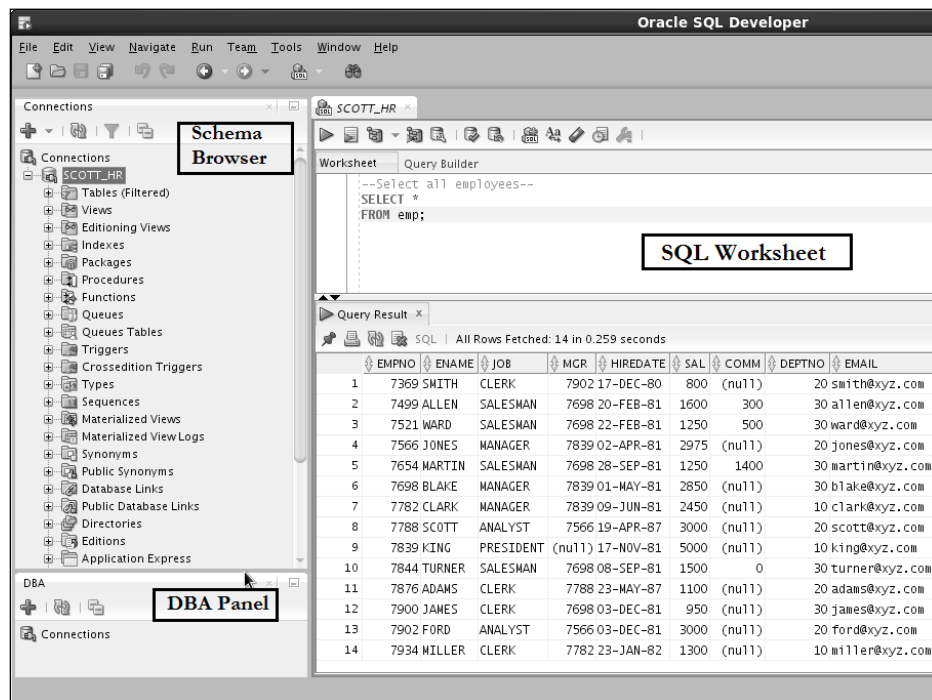
The steps for creating a database connection are as follows:

1. Provide a **Connection Name**, database **Username**, and **Password**.
2. Provide the connection type, role (default/ SYSDBA), and database server details such as the machine name or IP address, **Port**, and database **SID**.
3. Click on **Test** to verify the connection details. Status: Success will confirm the connection properties.
4. Click on **Save** to store the connection properties for future use.
5. Click on **Connect** to connect to the database using the current connection details.

Using the SQL worksheet

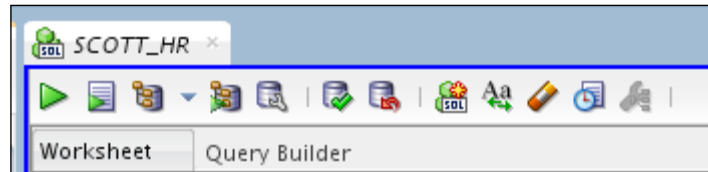
After establishing the connection to the database, SQL Developer opens up an SQL worksheet to execute SQL queries and PL/SQL programs.

For example, the following screenshot from SQL Developer shows a SELECT query on the EMP table and the query output in the following grid. Note the schema browser and the DBA panel in the left pane of the tool:



SQL Worksheet in the SQL Developer

Each of the SQL worksheets comprise multiple icons, as shown in the following figure:



The function of each icon, going from left to right, is described as follows:

- **Run Statement:** Select an SQL query text or PL/SQL block and click on this to execute it
- **Run Script:** execute a stored script
- **Explain Plan:** generate the explain plan of an SQL
- **Autotrace:** examine the autotrace parameters
- **SQL Tuning Advisor:** analyze simple or complex SQL statements and provide tuning recommendations
- **Commit:** COMMIT an active transaction
- **Rollback:** ROLLBACK a transaction
- **Open a SQL Worksheet:** open a nonshared SQL worksheet
- **To Upper/Lower/Initcap:** change the case of the query identifiers
- **Clear:** clear the content of the SQL worksheet
- **SQL History:** list the SQL statements previously executed
- **TimesTen Index Advisor:** analyze the SQL workload and recommend indexes

The resulting grid at the bottom provides icons for pinning a query result, printing output, refreshing the result, and displaying the SQL statement. At the result set level, it includes the following functionalities:

- **Save Grid as Report:** This allows the user to save an SQL query as a report.
- **Single Record View:** This provides a single record view in a dialog box.
- **Count Rows:** This returns the count of rows in the query result set.
- **Find/Highlight:** This searches for text and highlights its occurrences.
- **Publish to APEX:** This allows quick creation of an APEX application page from the SQL query. It requires details such as the workspace, application name, theme, page name, and underlying SQL.
- **Export:** This exports the query result set as insert scripts, SQL* Loader, CSV, delimited, HTML, XLS, PDF, or XML.

Core features of SQL Developer

The core functionalities of SQL Developer are those that help developers and administrators to drive their day-to-day activities with ease and enhanced productivity.

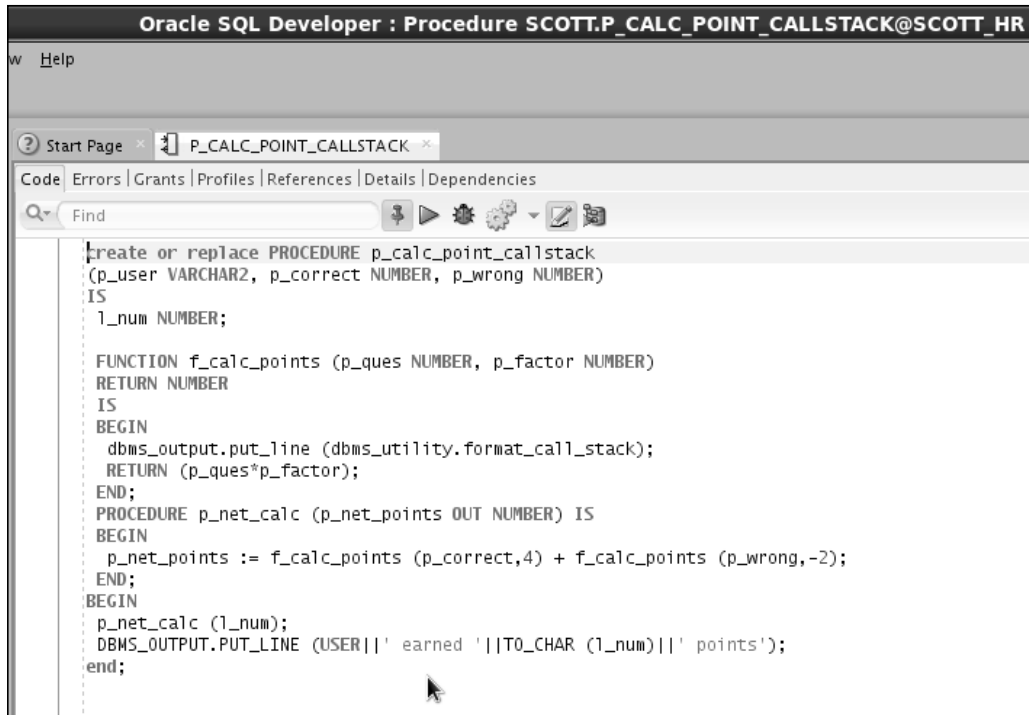
Object Browser

You can browse the objects in a user schema by expanding a saved connection (which represents a schema). You can find tables, (editioning) views, indexes, packages, procedures, functions, (cross edition) triggers, types, sequences, materialized views, materialized view logs, (public) synonyms, (public) database links, editions, directories, and other schema objects.

PL/SQL Editor and Debugger

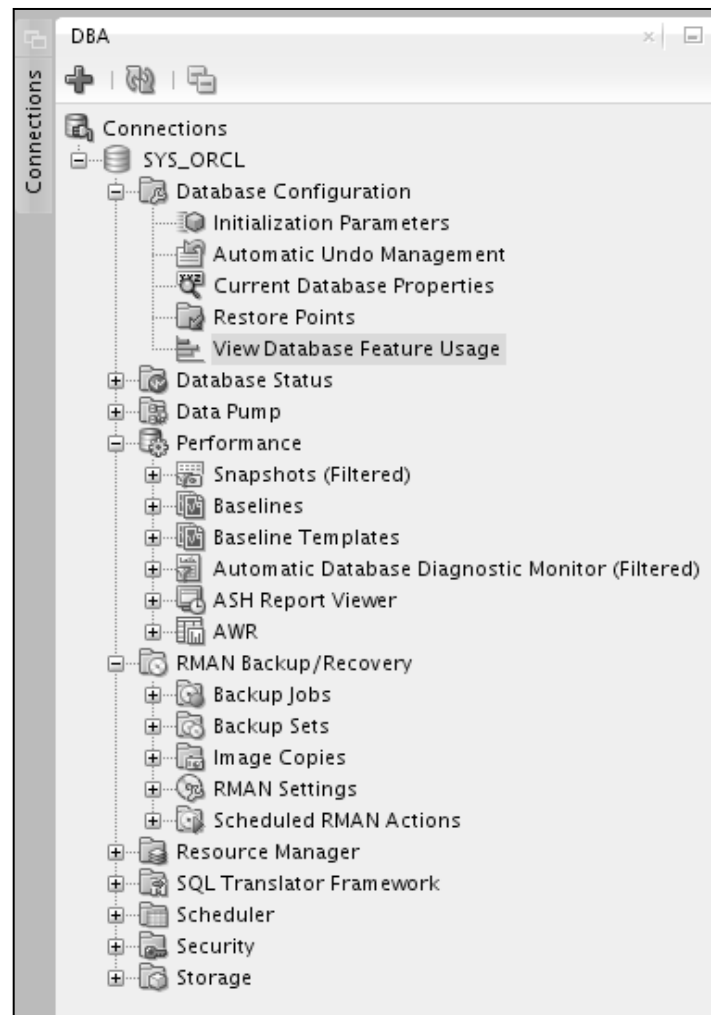
The procedure editor allows you to create or edit a PL/SQL package, stored procedure, or stored function. You can compile/compile for debug, debug, or profile the program units from the editor.


The following screenshot shows a procedure, P_CALC_POINT_CALLSTACK, in the procedure editor. Note the **Errors**, **Grants**, **Profiles**, **References**, **Details**, and **Dependencies** tabs. The toolbar icons allow the user to run, debug, compile, set read-only mode, and profile the program unit:



DBA Panel

You can either add existing connections from the connection tree or create a new connection with SYSDBA privileges. Under the DBA panel, you can perform the core functions of a database administrator like the one shown in the following figure:




 Use of **Performance** under DBA Console requires Oracle Diagnostics Pack. It is included in SQL Developer 4.0.

Database Utilities

This section briefly describes the database utilities available in SQL Developer:

- **Database Copy:** This allows the copying of data from one connection to another connection. Copy options can be object copy, schema copy or tablespace copy.

- **Database Diff:** This utility compares two connections on all or selected types of object, and generates the delta report.
- **Database Export:** This is used to generate the creation (DDL), as well as data insert scripts, for all selected object types.
- **Migration:** This is used to migrate a third-party database to Oracle. The non-Oracle databases that can be translated to Oracle can be MySQL, Microsoft SQL Server, Sybase, and IBM DB2. SQL Developer allows you to create a database connection to the non-Oracle databases. The migration process involves the creation of a migration project and a migration repository to hold the meta information of the non-Oracle objects. The translation framework is then used to translate all non-Oracle objects to Oracle-specific code and generate the DDL scripts. After the DDL scripts are executed, the last step of the process is data migration.
- **SQL Monitor:** This enables real-time SQL monitoring for a connection. The feature requires the Oracle Tuning Pack.

The Data Modeler

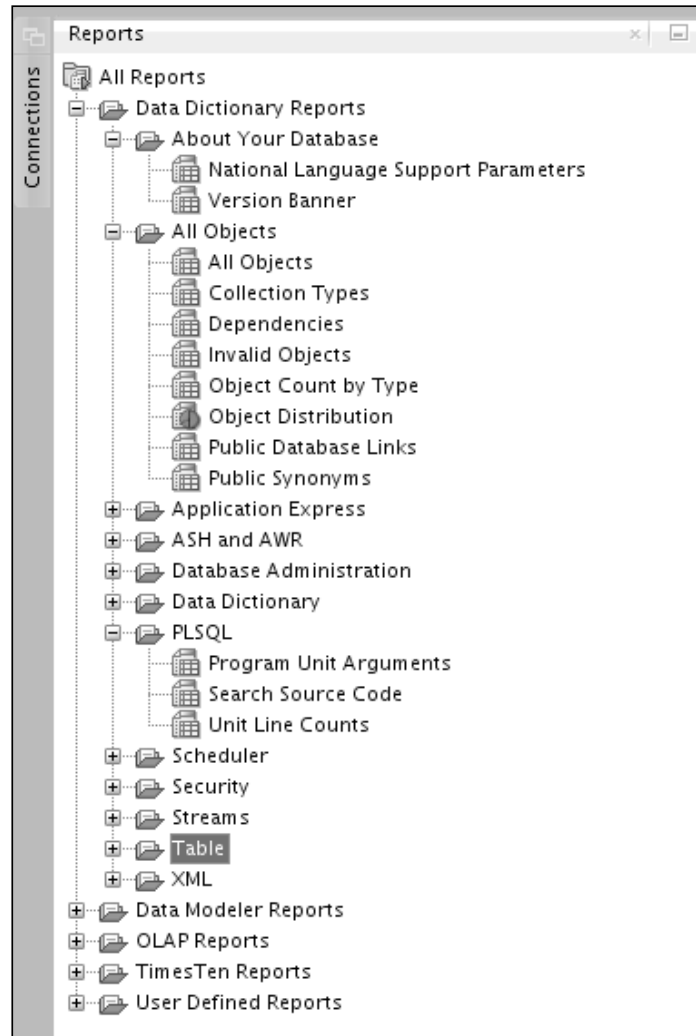
The Data Modeler is a free database design and modeling tool integrated with SQL Developer. SQL Developer provides a graphical user interface to create and manage data models. The unique capabilities of the Data Modeler simplify modeling tasks and helps designers to achieve better productivity. Although it is integrated with SQL Developer as an extension, the tool is available for standalone download as well.

This tool allows you to strategize and analyze logical data models. It also allows you to draw relational and physical data models. You can also import the ERDs and share them with your peers for review discussions, future reference, re-engineering or management approval.

SQL Developer reports

Oracle SQL Developer stores multiple predefined reports, which run based on user-provided inputs. The user must supply the values for the bind variables required for the report execution. Some of the popular reports are object reports, PL/SQL reports, and DBA reports. Object reports can be used to generate dependency tracking and invalid objects. PL/SQL reports can be used to report the argument usage in a given program unit or complete schema. Database administration reports can be used to generate reports on SQL cursors, database parameters, locks, memory consumption, session-wise information, top SQL reports, and waits and events. Other reports, such as the AWR and ASH reports, reveal active session history statistics and the last AWR generated.

Note that these reports are different from the user-defined reports. The Oracle-defined reports are non-editable, but they can be included in the user-defined reports as child reports or multilevel reports:



Version control

SQL Developer includes support for version control, which helps in maintaining the source code versions. You can implement source control by importing the files to be versioned, check-out when required, and check-in after the changes have been committed.

The SQL Translation Framework

The SQL Translation Framework is used to translate SQL statements from a non-Oracle Database script to an Oracle Database SQL script. It is installed along with the Oracle Database software. However, it must be configured with the appropriate SQL Translator to identify the non-Oracle code and convert it into Oracle-compliant code. During the translation process, a profile (known as the SQL Translation Profile) is generated for the SQL Translator to review and edit the translations. A SQL Translator may have multiple SQL Translation Profiles.

SQL Developer 4.0 and 4.1 New Features

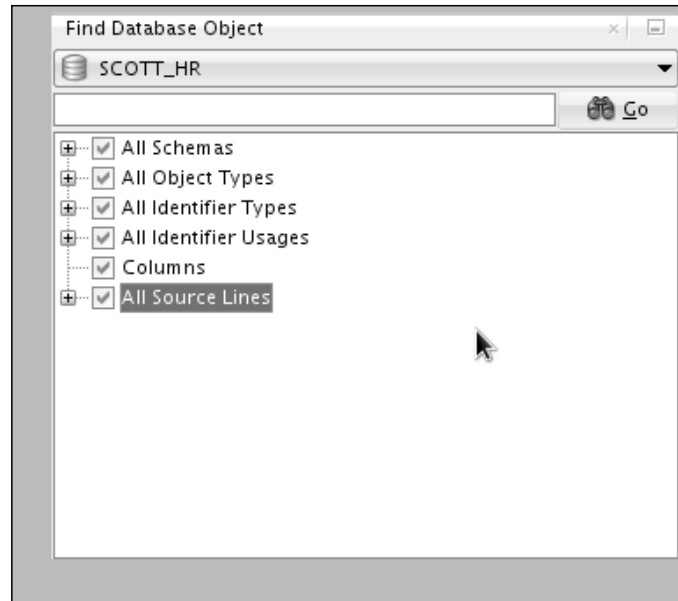
The latest version, SQL Developer 4.1, introduces numerous enhancements and features. The following is a list of new features:

- **Support for JDK 8:** SQL Developer 4.1 runs on JDK 8.
- **Database performance monitoring:** SQL Developer 4.0 included a node under the DBA connection tree for Oracle Diagnostics Pack. The performance engineers can now capture snapshots, establish baselines, run ASH and AWR, and display **Automatic Database Diagnostic Monitor (ADDM)** reports.
- **Full support for Oracle Database 12c features:** SQL Developer 4.0 fully supports the Oracle Database 12c features. In a multitenant database, a container database administrator can comfortably work with SQL Developer to manage pluggable databases. You can create, clone, drop, plug, or unplug a pluggable database. Similarly, you can create and manage data redaction policies on the tables. You can also clone a pluggable database directly to the Oracle cloud.

Other language enhancements in Oracle Database 12c, such as the `IDENTITY` columns, `FETCH FIRST`, `32K VARCHAR2` support, implicit cursors, and default to a sequence generator, enable efficient migration from non-Oracle to Oracle.

- **Support for Oracle Database products:** SQL Developer 4.0 supports Data Mining, Times Ten, XML DB, and Spatial and graphs.
- **Instance Viewer:** Without the need for an agent installation on the server, SQL Developer can display your database instance status and current activity. The instance activity is locally cached and graphically drawn, and gets refreshed from time to time. From the DBA Panel connection tree, select **Database Status node** and click on **DB Instance**.

- **Object search:** The enhanced object search allows an object to be searched across schemas and object types and through the identifiers and their appearances. The history of searched objects, along with the choices, is saved for future access:



- **New search and replace:** The search text field in the procedure editor is enhanced to highlight the occurrences of searched text, save the history of texts searched, and to allow replacement of existing text with replacement text. In addition, multicursor editing enables easy formatting of multiple rows in an SQL worksheet.
- **PDB cloning to Oracle Database Cloud Service:** SQL Developer 4.1 allows a PDB to be cloned directly to the Oracle Database Cloud. A cloud connection must be created with the Database Cloud Service environment details.
- **Support for Oracle NoSQL Database:** SQL Developer 4.1 supports Oracle NoSQL KVLite stores.
- **Import data from a spreadsheet:** SQL Developer 4.1 allows the data from a spreadsheet to be directly imported into a table. The spreadsheet data can be imported to an existing table or a new table.

Summary

SQL Developer has matured immensely over the years and has emerged as the primary IDE for Oracle Database. Full support for Oracle Database options in traditional as well as cloud deployments and the most recent database features are the key differentiators of the tool. This tool is not only meant for application developers, but also provides wide support for database administrators and architects. SQLcl is gaining a lot of traction in the developer community because of its advanced formatting and editing features.

Index

Symbols

32K VARCHAR2 46, 47

A

Active Session History (ASH) 72

American National Standard Institute
(ANSI) 43

archive logs 38

associative arrays
about 101-103
features 103

Automatic Database Diagnostic
Monitor (ADDM) 388

Automatic Segment Space
Management (ASSM) 201

Automatic Workload Repository (AWR) 72

B

BasicFiles

migrating, to SecureFiles 220-222

BFILE 195

binaries, SQL Developer

URL, for downloading 379

Binary Large Object (BLOB) 194

Buffer Cache

versus Result Cache 280

bulk binding 262

bulk processing, in PL/SQL

about 257

BULK COLLECT 257-262

FORALL 262-267

FORALL and exception handling 267-270

C

Character Large Object (CLOB) 195

collection

about 99

associative array 100

nested table 101

non-persistent collection 100

overview 100

persistent collection 100

types 100

types, comparing 122, 123

types, selecting 124

varray 101

collection methods, PL/SQL

about 127

COUNT function 128

DELETE function 133-135

EXISTS function 127, 128

EXTEND function 131, 132

FIRST function 129

Last function 129

LIMIT function 129

NEXT function 130

PRIOR function 130

TRIM function 132, 133

common users 42

compilation mode

selecting 240

setting 240

settings, querying 241

COUNT function 128

cursor attributes

%FOUND 6, 78

%ISOPEN 6, 78

%NOTFOUND 6, 78

- `%ROWCOUNT` 6, 78
- about 78
- cursor execution cycle, stages**
 - BIND 76
 - CLOSE 77
 - EXECUTE 77
 - FETCH 77
 - OPEN 76
 - PARSE 76
- cursors**
 - about 5
 - cursor attributes 6
 - Cursor FOR loop 8
 - execution cycle 5, 6
- cursor structures**
 - about 76
 - cursor execution cycle 76
 - explicit cursors 76-83
 - implicit cursors 76-80
- cursor variables**
 - about 83-85
 - as arguments 88
 - design, considerations 89, 90
 - design, guidelines 90, 91
 - restrictions 89
 - strong ref cursor, types 85, 86
 - SYS_REFCURSOR 87
 - weak ref cursor, types 85, 86
 - working with 86, 87

D

- database connection**
 - creating 380, 381
- database consolidation** 36-38
- database dependency**
 - direct 26
 - direct dependency, displaying 26, 27
 - enhancement 27, 28
 - indirect 26
 - indirect dependency, displaying 26, 27
 - issues 27, 28
 - managing 26
 - metadata 27
- Database In-Memory**
 - versus Result Cache 280
- database resource manager (DBRM)** 41

- database tuning** 229
- Database Utilities option**
 - Database Copy 385
 - Database Diff 386
 - Database Export 386
 - Migration 386
 - SQL Monitor 386
- Data Modeler** 386
- DBMS_ASSERT subprogram**
 - limitations 364
 - literal 363
 - quoted identifier 363
 - unquoted identifier 361, 362
 - used, for sanitizing inputs 360, 361
- DBMS_HPROF package**
 - about 343
 - Analyzer 344
 - and DBMS_PROFILER, differences 343
 - Data Collector 344
 - subprograms 344
- DBMS_LOB package**
 - about 197
 - DBMS_LOB constants 197, 198
 - DBMS_LOB data types 198
 - DBMS_LOB subprograms 199, 200
- DBMS_METADATA package**
 - about 328
 - data types 328, 329
 - demonstration 333-336
 - filters 331-333
 - subprograms 330, 331
 - transformation parameters 331
 - URL 329
- DBMS_RESULT_CACHE package**
 - about 295, 296
 - Oracle Database 12c enhancements,
 - to PL/SQL function cache 298
 - result cache memory report,
 - displaying 297, 298
- DBMS_TRACE package**
 - installing 337
 - subprograms 337, 338
- DBMS_TRACE subprograms**
 - URL 337
- dead code** 234
- DEFAULT ON NULL clause** 45, 46
- dense collection** 104

dynamic linked library (DLL) 140, 238

E

Early Adopter binaries, SQLcl

URL 377

environment setup, external procedures

EXTPROC.ora 144, 145

TNSNAMES.ora 144

exception handling

about 9

exception propagation 16, 17

system-defined exceptions 9, 10

user-defined exceptions 11-13

EXISTS function 127, 128

explicit cursors 5, 76, 80-83

EXTEND function 131, 132

external C programs

executing, from PL/SQL 145-149

external LOB 194

external procedures

about 139, 140

callback 142

callout 142

call specification 142

environment setup 143

executing 143

execution flow, components 140

extproc agent 141

library object 141

securing, with Oracle database 12c 149, 150

extproc agent, external procedures 141

F

FETCH FIRST clause 47

Fine-grained access control (FGAC)

about 163

working 163, 164

Fine Grained Dependency (FGD) 28

FIRST function 129

FORALL

about 262-267

and exception handling 267-270

functions

about 20

calling from SQL expressions,
restrictions 22, 23

execution methods 21, 22

features 20, 21

G

graphical user interface (GUI) 373

I

icons, SQL worksheet

Run Statement 382

identifier types

determining 323

PLSCOPE_SETTINGS parameter 325-327

PL/Scope tool 324

USER_IDENTIFIERS view 323

implicit cursors 5 76-80

implicit data type conversion

avoiding 252

In-Database Archiving 53-55

index-organized table (IOT) 105

injection 354

In-Memory Database Cache (IMDB)

versus Result Cache 280

interpreted compilation

program unit, comparing 241, 242

IPC (Internet Procedure Calls) 144

J

Java Development Kit (JDK) 379

Java programs, executing from PL/SQL

about 150

Java class, executing from Oracle PL/SQL
unit 151-154

Java class, loading into database 150, 151

Java Runtime Environment (JRE) 378

L

LAST function 129

library object, external procedures 141

Libunits 150

LIMIT function 129

Large Objects (LOBs)

about 192

data, populating 212-214

datatypes, classifying 193

- DBMS_LOB package 197
- external LOB 194
- instance initialization 196, 197
- internal LOB 193
- locator 195
- LONG, migrating to 218
- metadata 207, 208
- permanent LOB 193
- restrictions 194
- SecureFile advanced features,
 - securing 209-211
- security 192
- temporary LOB 193
- temporary LOB operations 215, 216
- usage notes 201
- working with 205, 206
- Large Objects (LOBs), data types**
 - about 194
 - BFILE 195
 - Binary Large Object (BLOB) 194
 - Character Large Object (CLOB) 195
 - NCLOB 195
- local user 42**
- LONG, migrating to LOBs**
 - ALTER TABLE command, using 219
 - Online Table Redefinition 220
 - TO_LOB function, using 219
- LRU (Least Recently Used) 278**

M

- machine code (M-code) 230**

N

- native compilation**
 - about 238
 - program unit, comparing 241-243
- nested table**
 - about 101-104
 - as schema object 106, 107
 - collection, initialization 111, 112
 - collection type, in PL/SQL 110
 - column, querying 110
 - design considerations 105
 - in index organized table 105
 - instance, creating 108, 109
 - locators 106

- metadata, querying 112
- object type, dropping 105
- object type, modifying 105
- storage 105
- type column, operations 107
- NEXT function 130, 131**
- NOT NULL constraint 253, 254**

O

- Object Browser 383**
- OCI Client results cache**
 - about 295
 - parameters 295
- OCI (Oracle Call Interface) 295**
- Oracle 8i 99**
- Oracle 11g 39**
- Oracle 12c container database (CDB) 36**
- Oracle 12c SQL and PL/SQL, features**
 - 32K VARCHAR2, support for 46, 47
 - about 43
 - column value, to sequence in Oracle 12c 45
 - DEFAULT ON NULL clause 45
 - IDENTITY columns 43, 44
 - In-Database Archiving 53-55
 - invisible columns 49-51
 - Row limiting, FETCH FIRST used 47, 48
 - temporal databases 51-53
- Oracle Advanced Security 160**
- Oracle Audit Vault and Database Firewall (AVDF) 162**
- Oracle Database 11g real native compilation 239**
- Oracle Database 11g result cache**
 - about 276
 - Result Cache, versus Buffer Cache 280
 - Result Cache, versus Database In-Memory 280
 - Result Cache, versus In-Memory Database Cache (IMDB) 280
 - Server Result Cache 277
 - Server Result Cache, configuring 278-280
- Oracle Database 12c**
 - about 35
 - enhancements, to collections 124-126
 - enhancements, to PL/SQL subprograms 25
 - external procedures, securing with 149, 150

implicit statement results 91, 92
security, enhancements 180-182

**Oracle Database 12c (12.1.0.2),
In-Memory option**

about 67
and problem statement 68
architecture 69
challenge 68
features 68, 69
In-Memory Advisor 72
INMEMORY clause 71
Oracle Database In-Memory benefits 73
performance optimizations 72
store, controlling 70

Oracle Database 12c Data Redaction

about 181
demonstration 183-186
features 182
function types 182, 183
metadata 186

**Oracle Database 12c multitenant
architecture, features**

about 38
CDB resource management 41
common users 42
manage many as one 40
multitenant for consolidation 39
plug/unplug 39
rapid provisioning 40

Oracle Database Security 160

Oracle Database Vault 162

Oracle Key Vault (OKV) 161

Oracle SecureFiles

and BasicFiles 204
compression 201, 202
db_securefile parameter 204
deduplication 201
file system logging 203
free space management 203, 204
Write Gather Cache (WGC) 203

Oracle SQL Developer

about 30
Exchange program, URL 375
for application architects 30
for DBA 30
for developers 30

SQL Developer 4.0 31

Oracle-supplied packages

categorizing 29
DBMS_ALERT 28
DBMS_DATAPUMP 29
DBMS_HTTP 28
DBMS_LOCK 28
DBMS_OUTPUT 28
DBMS_PARALLEL_EXECUTE 28
DBMS_PDB 29
DBMS_PRIVILEGE_CAPTURE 29
DBMS_REDACT 29
DBMS_REDEFINITION 29
DBMS_RESOURCE_MANAGER 29
DBMS_SCHEDULER 28
DBMS_SESSION 28
DBMS_SQL 29
DBMS_UTILITY 29
reviewing 28, 29
UTL_FILE 28
UTL_MAIL 28

P

**Payment Card Industry Data Security
Standard (PCI-DSS) 160**

persistent LOB 193

PGA (Process Global Area) 291

PL/Scope tool

about 324
PLSCOPE_SETTINGS parameter 325-327

plshprof utility

about 348-350
reports, exploring 350

PL/SQL

about 2
accomplishments 2
advantages 2
block 4
bulk processing 257
collection methods 127
database, recompiling for interpreted
compilation 243-247
database, recompiling for native
compilation 243-247
Editor and Debugger 383
enhancements 67

- exception handling 9
- external C programs, executing 145-148
- Java programs, executing 150
- package 23, 24
- program, fundamentals 3
- unit white listing 61, 62
- PL/SQL block**
 - anonymous PL/SQL block 4
 - named 4
 - nested 4
- PL/SQL code**
 - implicit data type conversion, avoiding 252
 - NOT NULL constraint 253, 254
 - numeric data type, selecting 254, 256
 - parameters, calling by reference 251
 - secure applications building, bind
 - variables used 247, 251
 - tuning 247
- PL/SQL code, designing**
 - cursor structures 76
 - cursor variables 83-85
 - subtypes 93
- PL/SQL code, profiling**
 - about 343
 - DBMS_HPROF package 343
 - plshprof utility 348
 - profiler data, analyzing 346
 - raw profile data, collecting 344, 345
- PL/SQL coding information, tracking**
 - about 306, 307
 - DBMS_DESCRIBE package 312, 313
 - program execution subprogram call stack,
 - tracking 315-319
 - propagating exceptions in PL/SQL code,
 - tracking 319-323
 - USER_ARGUMENTS view 307
 - USER_DEPENDENCIES view 312
 - USER_OBJECT_SIZE view 308
 - USER_OBJECTS view 307, 308
 - USER_PLSQL_OBJECT_SETTINGS
 - view 310, 311
 - USER_PROCEDURES view 310
 - USER_SOURCE view 308, 309
 - USER_STORED_SETTINGS view 310, 311
- PL/SQL compiler**
 - about 230
 - subprogram inlining 230, 231
- PL/SQL function result cache**
 - about 290
 - and other caching techniques,
 - differences 291
 - illustration 291, 292
 - invalidation 294
 - limitations 294
 - monitoring 293
- PLSQL_OPTIMIZE_LEVEL**
 - about 232
 - cases 233-237
- PL/SQL programs**
 - compiling, for debugging 339
 - DBMS_TRACE package, installing 337
 - DBMS_TRACE subprograms 337-339
 - execution, tracing 341, 342
 - PL/SQL trace information, viewing 340
 - profiling 342
 - sample 305, 306
 - tracing, DBMS_TRACE used 336, 337
- PL/SQL program units**
 - roles, granting 62, 63
 - test setup 63-66
- PL/SQL subprogram**
 - comparative analysis 60
 - defining, in PRAGMA UDF 55, 56
 - defining, in SELECT query 55, 56
 - test setup 57-60
- PL/SQL virtual machine (PVM) 238**
- pluggable database (PDB) 36, 40**
- PRAGMA INLINE 231**
- PRIOR function 130, 131**
- Procedural Language-Structured Query Language. See PL/SQL**
- procedures. See stored procedures**
- profiler data**
 - analyzing 346
 - profiler output, analyzing 347
 - profiler tables, creating 346
 - profiler tables, querying 347, 348

R

- RAC (Real Application Clusters)**
 - about 238
 - features 300
 - result cache 299

RAISE_APPLICATION_ERROR
 procedure 13-15

raw profile data
 collecting 344, 345
 interpreting 345

Real Application Security (RAS) 161

redo logs 38

Remote-Ops (RO) 144

Result Cache
 OCI result cache 277
 PL/SQL function result cache 277
 SQL query result cache 277
 versus Buffer Cache 280
 versus Database In-Memory 280
 versus In-Memory Database
 Cache (IMDB) 280

S

SCN (system change number) 289

SDDM (SQL Developer Data Modeler) 377

SDK (software development kit) 374

Server Result Cache

 about 275-277
 configuring 278-280

Shared Global Area (SGA) 37

Single Instruction Multiple Data (SIMD) 72

SQLcl

 about 377
 commands 378
 key features 378
 URL, for YouTube video 378

SQLcl, key features and commands

 CD 378
 CTAS 378
 database connection 378
 DDL 378
 INFORMATION 378
 multiline edits 378
 Object-name/Command completion 378
 SQLFORMAT 378
 SQL history 378

SQL Developer

 background 375
 database connection, creating 380, 381
 Extensions Exchange, URL 374
 features 383

 for database administrators 376
 for developers 376
 history 375
 overview 374
 reports 386
 releases 375, 376
 version 4.1, enhancements 388, 389
 version control 387
 working with 379, 380

SQL Developer, features

 about 383
 Data Modeler 386
 Object Browser 383
 PL/SQL Editor and Debugger 383
 SQL Developer reports 386
 SQL Translation Framework 388
 version control 387

SQL Developer Data Modeler (SDDM) 31

SQL Developer tool

 key factors, for differentiating from
 other tools 374, 375

SQL injection

 about 354
 PL/SQL code, exploiting 355-359
 targets 354

SQL injection, attacks

 DBMS_ASSERT, limitations 364
 inputs sanitizing, DBMS_ASSERT
 used 360, 361
 preventing 359, 360
 preventing, best practices 366, 367
 preventing, bind variables used 365, 366
 right subprogram, selecting for right
 identifier 361

SQL injection, code

 fuzz tools 368
 review 367
 static code analysis 368
 test cases, generating 369
 testing strategy 367

SQL query result cache

 about 281-284
 invalidation 288
 limitations 289
 monitoring 284-287
 read consistency 289

SQL Translation Framework 388

SQL worksheet

- functionalities 383
- icons 382
- using 381-383

SQL worksheet, functionalities

- Count Rows 383
- Export 383
- Find/Highlight 383
- Publish to APEX 383
- Save Grid as Report 383
- Single Record View 383

stored procedures

- creating 17, 18
- executing 19

Structured Query Language (SQL) 1

subprogram inlining

- PLSQL_OPTIMIZE_LEVEL 232
- PRAGMA INLINE 231

subtype

- about 93
- classifying 93, 94
- type compatibility 95

SYSAUX tablespace 38

system-defined exceptions 9, 10

system global area (SGA) 76

SYSTEM tablespace 38

T

TEMP tablespace 38

Total Recall feature 52

Transparent Data Encryption (TDE) 161, 202

TRIM function 132, 133

U

UNDO tablespace 38

user-defined exceptions

- RAISE_APPLICATION_ERROR
- procedure 13-16

User Global Area (UGA) 5, 76, 166

V

varray

- about 101, 116, 117
- as schema object 117, 118
- in PL/SQL 122

varray type columns, operations

- about 119
- varray collection type instance, inserting 119
- varray column, querying 120, 121
- varray instance, updating 121

Virtual Private Database (VPD)

- about 161, 164
- best practices 177
- column-level 164, 165
- components 165
- demonstration 170-176
- features 177
- metadata 178
- policy utility activities 179
- with Oracle Database 12c multitenant 165
- working 164

Virtual Private Database (VPD),

components

- application context 165-167
- DBMS_RLS package 169, 170
- policy function 168
- policy type 169

W

Write Gather Cache (WGC) 203



Thank you for buying
Advanced Oracle PL/SQL Developer's Guide
Second Edition

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at www.packtpub.com.

About Packt Enterprise

In 2010, Packt launched two new brands, Packt Enterprise and Packt Open Source, in order to continue its focus on specialization. This book is part of the Packt Enterprise brand, home to books published on enterprise software – software created by major vendors, including (but not limited to) IBM, Microsoft, and Oracle, often for use in other corporations. Its titles will offer information relevant to a range of users of this software, including administrators, developers, architects, and end users.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



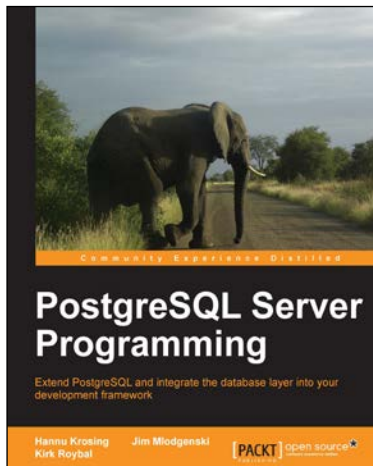
Oracle Advanced PL/SQL Developer Professional Guide

ISBN: 978-1-84968-722-5

Paperback: 440 pages

Master advanced PL/SQL concepts along with plenty of example questions for 1Z0-146 examination

1. Blitz the 1Z0-146 exam.
2. Master the advanced features of PL/SQL to design and optimize code using real-time demonstrations.
3. Efficiently design PL/SQL code with cursor design and subtypes.



PostgreSQL Server Programming

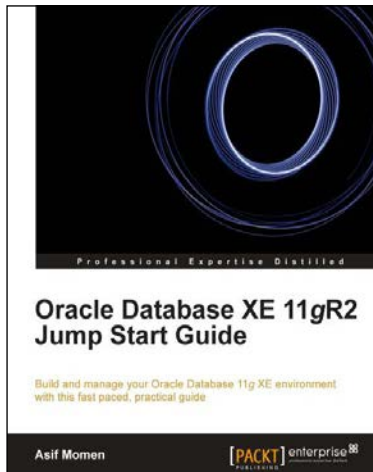
ISBN: 978-1-84951-698-3

Paperback: 264 pages

Extend PostgreSQL and integrate the database layer into your development framework

1. Understand the extension framework of PostgreSQL, and leverage it in ways that you haven't even invented yet.
2. Write functions, create your own data types, all in your favourite programming language.
3. Step-by-step tutorial with plenty of tips and tricks to kick-start server programming.

Please check www.PacktPub.com for information on our titles



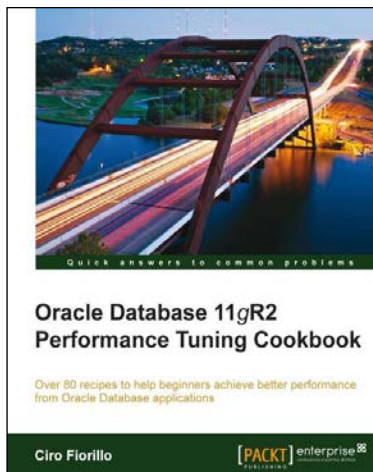
Oracle Database XE 11gR2 Jump Start Guide

ISBN: 978-1-84968-674-7

Paperback: 146 pages

Build and manage your Oracle Database 11g XE environment with this fast paced, practical guide

1. Install and configure Oracle Database XE on Windows and Linux.
2. Develop database applications using Oracle Application Express.
3. Back up, restore, and tune your database.
4. Includes clear step-by-step instructions and examples.



Oracle Database 11gR2 Performance Tuning Cookbook

ISBN: 978-1-84968-260-2

Paperback: 542 pages

Over 80 recipes to help beginners achieve better performance from Oracle Database applications

1. Learn the right techniques to achieve best performance from the Oracle Database.
2. Avoid common myths and pitfalls that slow down the database.
3. Diagnose problems when they arise and employ tricks to prevent them.
4. Explore various aspects that affect performance, from application design to system tuning.

Please check www.PacktPub.com for information on our titles