

Oracle Database 12c: Advanced PL/SQL

Student Guide - Volume II

D80343GC10

Edition 1.0

April 2014

D86296

ORACLE®

Author

Sharon Sophia Stephen

**Technical Contributors
and Reviewers**

Branislav Valny

Brent Dayley

Krishnanjani Chitta

Laszlo Czinkoczki

Nancy Greenberg

Sailaja Pasupuleti

Swarnapriya Shridhar

Wayne Abbott

Editors

Aju Kumar

Raj Kumar

Graphic Designer

Divya Thallap

Publishers

Joseph Fernandez

Jayanthy Keshavamurthy

Veena Narasimhan

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Disclaimer

This document contains proprietary information and is protected by copyright and other intellectual property laws. You may copy and print this document solely for your own use in an Oracle training course. The document may not be modified or altered in any way. Except where your use constitutes "fair use" under copyright law, you may not use, share, download, upload, copy, print, display, perform, reproduce, publish, license, post, transmit, or distribute this document in whole or in part without the express authorization of Oracle.

The information contained in this document is subject to change without notice. If you find any problems in the document, please report them in writing to: Oracle University, 500 Oracle Parkway, Redwood Shores, California 94065 USA. This document is not warranted to be error-free.

Restricted Rights Notice

If this documentation is delivered to the United States Government or anyone using the documentation on behalf of the United States Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS

The U.S. Government's rights to use, modify, reproduce, release, perform, display, or disclose these training materials are restricted by the terms of the applicable Oracle license agreement and/or the applicable U.S. Government contract.

Trademark Notice

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Contents

1 Introduction

- Course Objectives 1-2
- Lesson Agenda 1-3
- Course Agenda 1-4
- Appendixes Used in This Course 1-5
- Lesson Agenda 1-6
- Development Environments: Overview 1-7
- What Is Oracle SQL Developer? 1-8
- Coding PL/SQL in SQL*Plus 1-9
- Lesson Agenda 1-10
- Tables Used in This Course 1-11
- Order Entry Schema 1-12
- Human Resources Schema 1-14
- Oracle SQL and PL/SQL Documentation 1-15
- Summary 1-16
- Practice 1 Overview: Getting Started 1-17

2 PL/SQL Programming Concepts: Review

- Objectives 2-2
- Pre-Quiz 2-3
- Lesson Agenda 2-11
- PL/SQL Block Structure 2-12
- Naming Conventions 2-13
- Procedures 2-15
- Procedure: Example 2-16
- Stored Functions 2-17
- Functions: Example 2-18
- Ways to Execute Functions 2-19
- Lesson Agenda 2-20
- Restrictions on Calling Functions from SQL Expressions 2-21
- Lesson Agenda 2-23
- PL/SQL Packages: Review 2-24
- Components of a PL/SQL Package 2-25
- Creating the Package Specification 2-26
- Creating the Package Body 2-27

Lesson Agenda	2-28
Cursor	2-29
Processing Explicit Cursors	2-31
Explicit Cursor Attributes	2-32
Cursor FOR Loops	2-33
Cursor: Example	2-34
Lesson Agenda	2-35
Handling Exceptions	2-36
Exceptions: Example	2-38
Predefined Oracle Server Errors	2-39
Predefined Oracle Server Exceptions	2-40
Trapping Non-Predefined Oracle Server Errors	2-42
Trapping User-Defined Exceptions	2-44
Lesson Agenda	2-45
RAISE_APPLICATION_ERROR Procedure	2-46
Lesson Agenda	2-48
Dependencies	2-49
Displaying Direct and Indirect Dependencies	2-51
Lesson Agenda	2-52
Using Oracle-Supplied Packages	2-53
Some of the Oracle-Supplied Packages	2-54
DBMS_OUTPUT Package	2-55
UTL_FILE Package	2-56
Summary	2-57
Practice 2: Overview	2-58

3 Designing PL/SQL Code

Objectives	3-2
Lesson Agenda	3-3
Guidelines for Cursor Design	3-4
Lesson Agenda	3-10
Cursor Variables: Overview	3-11
Working with Cursor Variables	3-12
Strong Versus Weak REF CURSOR Variables	3-13
Step 1: Defining a REF CURSOR Type	3-14
Step 1: Declaring a Cursor Variable	3-15
Step 1: Declaring a REF CURSOR Return Type	3-16
Step 2: Opening a Cursor Variable	3-17
Step 3: Fetching from a Cursor Variable	3-19
Step 4: Closing a Cursor Variable	3-20
Passing Cursor Variables as Arguments	3-21

Using the SYS_REF_CURSOR Predefined Type	3-24
Dynamic SQL for Cursor Variable	3-26
Rules for Cursor Variables	3-27
Comparing Cursor Variables with Static Cursors	3-28
Lesson Agenda	3-29
Predefined PL/SQL Data Types	3-30
Subtypes: Overview	3-31
Standard Subtypes	3-32
Benefits of Subtypes	3-34
Declaring Subtypes	3-35
Using Subtypes	3-36
Subtype Compatibility	3-37
Lesson Agenda	3-38
Declaring and Defining White Lists	3-39
Using the ACCESSIBLE BY Clause in the CREATE PROCEDURE Statement	3-40
Using the ACCESSIBLE BY Clause in the CREATE FUNCTION Statement	3-41
Using the ACCESSIBLE BY Clause in the CREATE PACKAGE Statement	3-42
Using the ACCESSIBLE BY Clause in the CREATE TYPE Statement	3-43
Using the ACCESSIBLE BY Clause	3-44
Quiz	3-45
Summary	3-50
Practice 3: Overview	3-51

4 Overview of Collections

Objectives	4-2
Lesson Agenda	4-3
Understanding Collections	4-4
Collection Types	4-5
Lesson Agenda	4-7
Using Associative Arrays	4-8
Creating the Array	4-10
Traversing the Array	4-11
Lesson Agenda	4-13
Using Nested Tables	4-14
Nested Table Storage	4-15
Creating Nested Table Types	4-16
Declaring Collections: Nested Table	4-17
Using Nested Tables	4-18
Referencing Collection Elements	4-20
Using Nested Tables in PL/SQL	4-21
Lesson Agenda	4-23

Varrays 4-24
Declaring Collections: Varray 4-25
Using Varrays 4-26
Quiz 4-28
Summary 4-31
Practice 4: Overview 4-32

5 Using Collections

Objectives 5-2
Lesson Agenda 5-3
Working with Collections in PL/SQL 5-4
Initializing Collections 5-7
Referencing Collection Elements 5-9
Using Collection Methods 5-10
Commonly Used Collection Methods 5-11
Using Collection Methods 5-12
Manipulating Individual Elements 5-14
Querying a Collection Using the TABLE Operator 5-16
Querying a Collection with Static SQL 5-17
Querying a Collection with the TABLE Operator 5-18
Lesson Agenda 5-20
Avoiding Collection Exceptions 5-21
Avoiding Collection Exceptions: Example 5-22
Lesson Agenda 5-23
Listing Characteristics for Collections 5-24
Guidelines for Using Collections Effectively 5-25
Lesson Agenda 5-26
PL/SQL Bind Types 5-27
Subprogram with a BOOLEAN Parameter 5-28
Subprogram with a Record Parameter 5-29
Subprogram with a Parameter of PL/SQL Collection Type 5-30
Quiz 5-31
Summary 5-34
Practice 5: Overview 5-35

6 Manipulating Large Objects

Objectives 6-2
Lesson Agenda 6-3
What Is a LOB? 6-4
Components of a LOB 6-6
Internal LOBs 6-7

Managing Internal LOBs	6-8
Lesson Agenda	6-9
What Are BFILEs?	6-10
Securing BFILEs	6-11
What Is a DIRECTORY?	6-12
Using the DBMS_LOB Package	6-14
DBMS_LOB.READ and DBMS_LOB.WRITE	6-15
Managing BFILEs: Role of a DBA	6-16
Managing BFILEs: Role of a Developer	6-17
Preparing to Use BFILEs	6-18
Populating BFILE Columns with SQL	6-19
Populating a BFILE Column with PL/SQL	6-20
Using DBMS_LOB Routines with BFILEs	6-21
Lesson Agenda	6-22
Initializing LOB Columns Added to a Table	6-23
Populating LOB Columns	6-25
Writing Data to a LOB	6-26
Reading LOBs from the Table	6-30
Updating LOB by Using DBMS_LOB in PL/SQL	6-31
Checking the Space Usage of a LOB Table	6-32
Selecting CLOB Values by Using SQL	6-34
Selecting CLOB Values by Using DBMS_LOB	6-35
Selecting CLOB Values in PL/SQL	6-36
Removing LOBs	6-37
Quiz	6-38
Lesson Agenda	6-41
Temporary LOBs	6-42
Creating a Temporary LOB	6-43
Lesson Agenda	6-44
SecureFile LOBs	6-45
Storage of SecureFile LOBs	6-46
Creating a SecureFile LOB	6-47
Comparing Performance	6-48
Quiz	6-49
Summary	6-50
Practice 6: Overview	6-51

7 Using Advanced Interface Methods

Objectives	7-2
Calling External Procedures from PL/SQL	7-3
Benefits of External Procedures	7-4

External C Procedure Components	7-5
How PL/SQL Calls a C External Procedure	7-6
extproc Process	7-7
Development Steps for External C Procedures	7-8
Call Specification	7-12
Publishing an External C Routine	7-15
Executing the External Procedure	7-16
Working with Java Methods Using PL/SQL: Overview	7-17
Calling a Java Class Method by Using PL/SQL	7-18
Development Steps for Java Class Methods	7-19
Loading Java Class Methods	7-20
Publishing a Java Class Method	7-21
Executing the Java Routine	7-23
Creating Packages for Java Class Methods	7-24
Quiz	7-25
Summary	7-28
Practice 7: Overview	7-29

8 Performance and Tuning

Objectives	8-2
Lesson Agenda	8-3
Native and Interpreted Compilation	8-4
Deciding on a Compilation Method	8-5
Setting the Compilation Method	8-6
Viewing the Compilation Settings	8-8
Setting Up a Database for Native Compilation	8-10
Compiling a Program Unit for Native Compilation	8-11
Lesson Agenda	8-12
Tuning PL/SQL Code	8-13
Avoiding Implicit Data Type Conversion	8-14
Understanding the NOT NULL Constraint	8-15
Working of the NOT NULL Constraint	8-16
Using the PLS_INTEGER Data Type for Integers	8-17
Using the SIMPLE_INTEGER Data Type	8-18
Modularizing Your Code	8-19
Comparing SQL with PL/SQL	8-20
Using Bulk Binding	8-24
Using SAVE EXCEPTIONS	8-30
Handling FORALL Exceptions	8-31
Rephrasing Conditional Control Statements	8-32
Passing Data Between PL/SQL Programs	8-34

Lesson Agenda	8-37
Intraunit Inlining: Introduction	8-38
Using Inlining	8-39
Inlining Concepts	8-40
Inlining: Example	8-43
Inlining: Guidelines	8-45
Quiz	8-46
Summary	8-49
Practice 8: Overview	8-50

9 Improving Performance with Caching

Objectives	9-2
Lesson Agenda	9-3
What Is Result Caching?	9-4
Increasing Result Cache Memory Size	9-5
Setting Result_Cache_Max_Size	9-6
Enabling Query Result Cache	9-7
Using the DBMS_RESULT_CACHE Package	9-8
Lesson Agenda	9-9
SQL Query Result Cache	9-10
Clearing the Shared Pool and Result Cache	9-12
Examining the Memory Cache	9-13
Examining the Execution Plan for a Query	9-14
Examining Another Execution Plan	9-15
Executing Both Queries	9-17
Viewing Cache Results Created	9-18
Re-Executing Both Queries	9-19
Viewing Cache Results Found	9-20
Lesson Agenda	9-21
PL/SQL Function Result Cache	9-22
Marking PL/SQL Function Results to Be Cached	9-23
Clearing the Shared Pool and Result Cache	9-24
Lesson Agenda	9-25
Creating a PL/SQL Function by Using the RESULT_CACHE Clause	9-26
Lesson Agenda	9-27
Calling the PL/SQL Function Inside a Query	9-28
Verifying Memory Allocation	9-29
Viewing Cache Results Created	9-30
Calling the PL/SQL Function Again	9-31
Viewing Cache Results Found	9-32
Confirming That the Cached Result Was Used	9-33

Lesson Agenda 9-34
Definer's Rights Versus Invoker's Rights 9-35
Defining Invoker's Right and Definer's Right 9-36
Specifying Invoker's Rights: Setting AUTHID to CURRENT_USER 9-37
Using CURRENT_USER and RESULT_CACHE 9-38
Ensuring That the Invoker's Rights Function is Result Cached: Requirements 9-39
Privilege Check for an Invoker's Rights Unit 9-40
Granting Roles to PL/SQL Packages and Stand-Alone Stored Subprograms 9-41
Quiz 9-42
Summary 9-47
Practice 9: Overview 9-48

10 Analyzing PL/SQL Code

Objectives 10-2
Lesson Agenda 10-3
Finding Coding Information: Using the Dictionary Views 10-4
Finding Coding Information: Using the Supplied Packages 10-5
Finding Coding Information 10-6
Using SQL Developer to Find Coding Information 10-10
Using DBMS_DESCRIBE 10-12
Using ALL_ARGUMENTS 10-15
Using SQL Developer to Report on Arguments 10-17
Using DBMS_UTILITY.FORMAT_CALL_STACK 10-19
Finding Error Information 10-21
Using DBMS_UTILITY.FORMAT_ERROR_STACK 10-22
Finding Error Information 10-23
Lesson Agenda 10-27
PL/Scope 10-28
Collecting PL/Scope Data 10-29
Using PL/Scope 10-30
USER/ALL/DBA_IDENTIFIERS Catalog View 10-31
Sample Data for PL/Scope 10-32
Collecting Identifiers 10-35
Viewing Identifier Information 10-36
Performing a Basic Identifier Search 10-38
Using USER_IDENTIFIERS to Find All Local Variables 10-39
Finding Identifier Actions 10-40
Lesson Agenda 10-42
DBMS_METADATA Package 10-43
Metadata API 10-44
Subprograms in DBMS_METADATA 10-45

FETCH_xxx Subprograms 10-46
SET_FILTER Procedure 10-47
Filters 10-48
Examples of Setting Filters 10-49
Programmatic Use: Example 1 10-50
Programmatic Use: Example 2 10-52
Browsing APIs 10-54
Browsing APIs: Examples 10-55
Lesson Agenda 10-57
Using the DBMS_UTILITY.EXPAND_SQL_TEXT Procedure 10-58
EXPAND_SQL_TEXT Exceptions 10-59
Using the UTL_CALL_STACK Package 10-60
UTL_CALL_STACK: Security Model 10-61
New Predefined Inquiry Directives in Oracle Database 12c 10-62
Quiz 10-63
Summary 10-66
Practice 10: Overview 10-67

11 Profiling and Tracing PL/SQL Code

Objectives 11-2
Lesson Agenda 11-3
Tracing PL/SQL Execution 11-4
Tracing PL/SQL: Steps 11-8
Step 1: Enable Specific Subprograms 11-9
Steps 2 and 3: Identify a Trace Level and Start Tracing 11-10
Step 4: Turn Off Tracing 11-11
Step 5: Examine the Trace Information 11-12
plsql_trace_runs and plsql_trace_events 11-13
Lesson Agenda 11-15
Hierarchical Profiling Concepts 11-16
Using the PL/SQL Profiler 11-18
Understanding Raw Profiler Data 11-24
Using the Hierarchical Profiler Tables 11-25
Using DBMS_HPROF.ANALYZE 11-26
Using DBMS_HPROF.ANALYZE to Write to Hierarchical Profiler Tables 11-27
Analyzer Output from the DBMSHP_RUNS Table 11-28
Analyzer Output from the DBMSHP_FUNCTION_INFO Table 11-29
plshprof: A Simple HTML Report Generator 11-30

Using plshprof	11-31
Using the HTML Reports	11-34
Quiz	11-38
Summary	11-41
Practice 11: Overview	11-42

12 Implementing Fine-Grained Access Control for VPD

Objectives	12-2
Lesson Agenda	12-3
Fine-Grained Access Control: Overview	12-4
Identifying Fine-Grained Access Features	12-5
How Fine-Grained Access Works	12-6
Why Use Fine-Grained Access	12-8
Lesson Agenda	12-9
Using an Application Context	12-10
Creating an Application Context	12-12
Setting a Context	12-13
Implementing a Policy	12-15
Step 1: Setting Up a Driving Context	12-16
Step 2: Creating the Package	12-17
Step 3: Defining the Policy	12-19
Step 4: Setting Up a Logon Trigger	12-22
Example Results	12-23
Data Dictionary Views	12-24
Using the ALL_CONTEXT Dictionary View	12-25
Policy Groups	12-27
More About Policies	12-28
Quiz	12-30
Summary	12-33
Practice 12: Overview	12-34

13 Safeguarding Your Code Against SQL Injection Attacks

Objectives	13-2
Lesson Agenda	13-3
Understanding SQL Injection	13-4
Identifying Types of SQL Injection Attacks	13-5
SQL Injection: Example	13-6
Assessing Vulnerability	13-7
Avoidance Strategies Against SQL Injection	13-8
Protecting Against SQL Injection: Example	13-9
Lesson Agenda	13-10

Reducing the Attack Surface	13-11
Expose the Database Only Via PL/SQL API	13-12
Using Invoker's Rights	13-13
Reducing Arbitrary Inputs	13-15
Strengthen Database Security	13-16
Lesson Agenda	13-17
Using Static SQL	13-18
Using Dynamic SQL	13-21
Lesson Agenda	13-22
Using Bind Arguments with Dynamic SQL	13-23
Using Bind Arguments with Dynamic PL/SQL	13-24
What if You Cannot Use Bind Arguments?	13-25
Lesson Agenda	13-26
Understanding DBMS_ASSERT	13-27
Formatting Oracle Identifiers	13-28
Working with Identifiers in Dynamic SQL	13-29
Choosing a Verification Route	13-30
Avoiding Injection by Using DBMS_ASSERT.ENQUOTE_LITERAL	13-31
Avoiding Injection by Using DBMS_ASSERT.SIMPLE_SQL_NAME	13-34
DBMS_ASSERT Guidelines	13-36
Quiz	13-40
Summary	13-44
Practice 13: Overview	13-45

Appendix A: Table Descriptions and Data

Appendix B: Using SQL Developer

Objectives	B-2
What Is Oracle SQL Developer?	B-3
Specifications of SQL Developer	B-4
SQL Developer 4.0 Interface	B-5
Creating a Database Connection	B-7
Browsing Database Objects	B-10
Displaying the Table Structure	B-11
Browsing Files	B-12
Finding Database Objects	B-13
Creating a Schema Object	B-15
Creating a New Table: Example	B-16
Using the SQL Worksheet	B-17
Executing SQL Statements	B-21
Saving SQL Scripts	B-22

Executing Saved Script Files: Method 1	B-23
Executing Saved Script Files: Method 2	B-24
Formatting the SQL Code	B-25
Using Snippets	B-26
Using Snippets: Example	B-27
Using Recycle Bin	B-28
Debugging Procedures and Functions	B-29
Database Reporting	B-30
Creating a User-Defined Report	B-31
Search Engines and External Tools	B-32
Setting Preferences	B-33
Resetting the SQL Developer Layout	B-34
Data Modeler in SQL Developer	B-35
Summary	B-36

Appendix C: Using SQL*Plus

Objectives	C-2
SQL and SQL*Plus Interaction	C-3
SQL Statements Versus SQL*Plus Commands	C-4
SQL Versus SQL*Plus	C-5
Using SQL*Plus	C-6
SQL Plus Commands: Categories	C-7
Logging In to SQL*Plus	C-8
Displaying the Table Structure	C-9
SQL*Plus Editing Commands	C-11
Using LIST, n, and APPEND	C-13
Using the CHANGE Command	C-14
SQL*Plus File Commands	C-15
Using the SAVE and START Commands	C-16
SERVERROUTPUT Command	C-17
Using the SQL*Plus SPOOL Command	C-18
Using the AUTOTRACE Command	C-19
Summary	C-20

Appendix D: Designing and Testing Your Code to Avoid SQL Injection Attacks

- Objectives D-2
- Using Bind Arguments D-3
- Avoiding Privilege Escalation D-4
- Beware of Filter Parameters D-5
- Trapping and Handling Exceptions D-6
- Coding Review and Testing Strategy D-7
- Reviewing Code D-8
- Running Static Code Analysis D-9
- Testing with Fuzzing Tools D-10
- Generating Test Cases D-11
- Summary D-13

THESE eKIT MATERIALS ARE FOR YOUR USE IN THIS CLASSROOM ONLY. COPYING eKIT MATERIALS FROM THIS COMPUTER IS STRICTLY PROHIBITED

Oracle University and Error : You are not a Valid Partner use only

Improving Performance with Caching

9

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Objectives

After completing this lesson, you should be able to do the following:

- Improve memory usage by caching SQL result sets
- Write queries that use the result cache hint
- Use the DBMS_RESULT_CACHE package
- Set up PL/SQL functions to use PL/SQL result caching



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In this lesson, you learn about the Oracle Database 12c caching techniques that can improve performance. You examine the improvement on the performance of queries by caching the results of a query in memory, and then using the cached results in future executions of the query or query fragments. The cached results reside in the result cache memory portion of the shared global area (SGA).

The result-caching mechanism of the PL/SQL cross-session function provides applications with a language-supported and system-managed means for storing the results of PL/SQL functions in an SGA, which is available to every session that runs the application.

Lesson Agenda

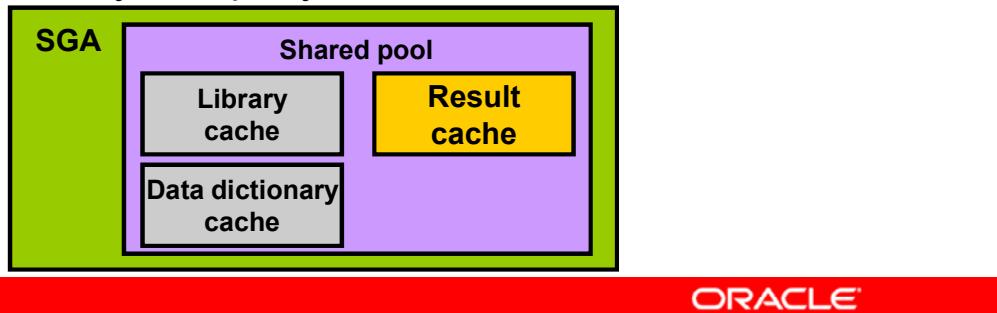
- Improving memory usage by caching SQL result sets
 - Enabling the query result cache
 - Using the DBMS_RESULT_CACHE package
- Implementing SQL query result caching
 - Writing queries that use the result cache hint
- Using PL/SQL function result caching
 - Setting up PL/SQL functions to use PL/SQL result caching
 - Implementing PL/SQL function result caching
 - Invoker's rights function result cache



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

What Is Result Caching?

- The result cache enables SQL query and PL/SQL function results to be stored in cache memory.
- Subsequent executions of the same query or function can be served directly out of the cache, improving response times.
- This technique can be especially effective for SQL queries and PL/SQL functions that are executed frequently.
- Cached query results become invalid when the database data accessed by the query is modified.

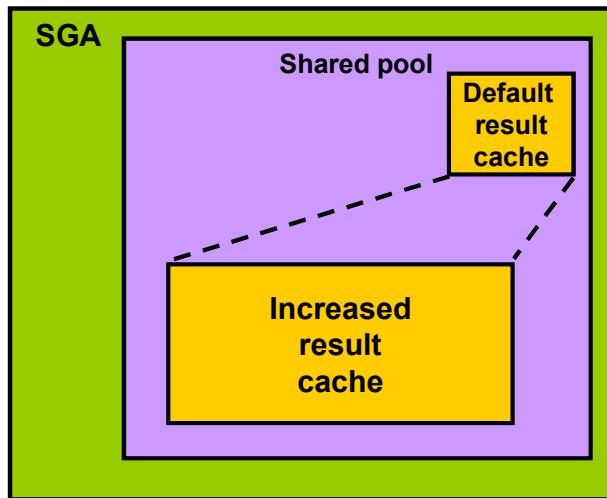


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The new SQL query result cache enables explicit caching of queries and query fragments in an area of the shared pool called “result cache memory.” When a query is executed, the result cache is built up and the result is returned. The database can then use the cached results for subsequent query executions, thereby resulting in faster response times. Cached query results become invalid when the data in the database objects being accessed by the query is modified.

Increasing Result Cache Memory Size

You can increase the small, default result cache memory size by using the `RESULT_CACHE_MAX_SIZE` initialization parameter.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

By default, on database startup, Oracle allocates memory to the result cache in the shared pool. The memory size allocated depends on the memory size of the shared pool as well as the memory management system.

- When using the `MEMORY_TARGET` initialization parameter to specify the memory allocation, Oracle allocates 0.25% of the memory target to the result cache.
- When you set the size of the shared pool by using the `SGA_TARGET` initialization parameter, Oracle allocates 0.5% of the SGA target to the result cache.
- If you specify the size of the shared pool by using the `SHARED_POOL_SIZE` initialization parameter, Oracle allocates 1% of the shared pool size to the result cache.

Note: Oracle will not allocate more than 75% of the shared pool to the result cache.

Use the `RESULT_CACHE_MAX_RESULT` initialization parameter to specify the maximum percentage of result cache memory that can be used by any single result. The default value is 5%, but you can specify any value between 1% and 100%.

Setting Result_Cache_Max_Size

- Set Result_Cache_Max_Size from the command line or in an initialization file created by a DBA.
- The cache size is dynamic and can be changed either permanently or until the instance is restarted.

```
SQL> ALTER SYSTEM SET result_cache_max_size = 2M SCOPE = MEMORY;
System altered.

SQL> SELECT name, value
  2  FROM v$parameter
  3 WHERE name = 'result_cache_max_size';

NAME                                VALUE
-----                               -----
result_cache_max_size                2097152
1 row selected.
```

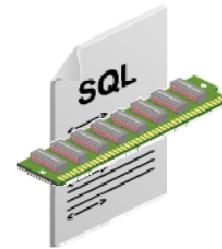


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

By default, the server-side result cache is configured to use a very small portion of the shared pool. You can manually set the result cache memory size by using the RESULT_CACHE_MAX_SIZE initialization parameter. Setting RESULT_CACHE_MAX_SIZE to 0 during database startup disables the server-side result cache. RESULT_CACHE_MAX_SIZE cannot be dynamically changed if the value is set to 0 during database startup in the SPFILE (sever parameter file) or the init.ora (initialization) file.

Enabling Query Result Cache

- Use the RESULT_CACHE_MODE initialization parameter in the database initialization parameter file.
- RESULT_CACHE_MODE can be set to:
 - MANUAL (default): You must add the RESULT_CACHE hint to your queries for the results to be cached.
 - FORCE: Results are always stored in the result cache memory, if possible.



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You can enable query result cache at the database level by using the RESULT_CACHE_MODE initialization parameter in the database initialization parameter file. The same parameter can also be used at the session level by using the ALTER SESSION command.

RESULT_CACHE_MODE can be set to:

- **MANUAL (default):** You must add the RESULT_CACHE hint to your queries for the results to be cached or to be served out of the cache. The RESULT_CACHE hint can also be added in subqueries and inline views.
- **FORCE:** Results are always stored in the result cache memory, if possible.

The use of the SQL query result cache introduces the ResultCache operator in the query execution plan.

Using the DBMS_RESULT_CACHE Package

The DBMS_RESULT_CACHE package provides an interface for a DBA to manage memory allocation for SQL query result cache and the PL/SQL function result cache.

```
execute dbms_result_cache.memory_report
```

```
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 = 1792K bytes (1792 blocks)
Maximum Result Size = 89K bytes (89 blocks)
[Memory]
Total Memory = 9440 bytes [0.004% of the Shared Pool]
... Fixed Memory = 9440 bytes [0.004% of the Shared Pool]
... Dynamic Memory = 0 bytes [0.000% of the Shared Pool]
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You can use the DBMS_RESULT_CACHE package to perform various operations, such as bypassing the cache, retrieving statistics on the cache memory usage, and flushing the cache. For example, to view the memory allocation statistics, use dbms_result_cache.memory_report. The output of this command is similar to the following:

```
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 = 1056K bytes (1056 blocks)
Maximum Result Size = 52K bytes (52 blocks)
[Memory]
Total Memory = 5140 bytes [0.003% of the Shared Pool]
... Fixed Memory = 5140 bytes [0.003% of the Shared Pool]
... Dynamic Memory = 0 bytes [0.000% of the Shared Pool]
```

Lesson Agenda

- Improving memory usage by caching SQL result sets
 - Enabling the query result cache
 - Using the `DBMS_RESULT_CACHE` package
- Implementing SQL query result caching
 - Writing queries that use the result cache hint
- Using PL/SQL function result caching
 - Setting up PL/SQL functions to use PL/SQL result caching
 - Implementing PL/SQL function result caching
 - Invoker's rights function result cache

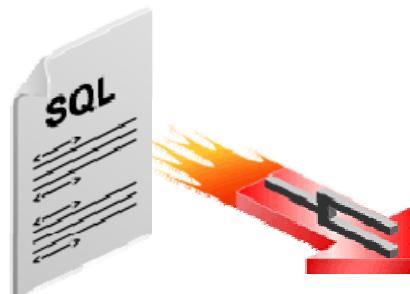


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

An example of setting the result cache hint is provided in the following few slides.

SQL Query Result Cache

- Definition:
 - Cache the results of the current query or query fragment in memory, and then use the cached results in future executions of the query or query fragments.
 - Cached results reside in the result cache memory portion of the SGA.
- Benefits:
 - Improved performance



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You can improve the performance of your queries by caching the results of a query in memory, and then using the cached results in future executions of the query or query fragments. The cached results reside in the result cache memory portion of the SGA. This feature is designed to speed up query execution on systems with large memories.

SQL Query Result Cache

- Scenario:
 - You need to find the greatest average value of credit limit grouped by state over the whole population.
 - The query returns a large number of rows being analyzed to yield a few rows or one row.
 - In your query, the data changes fairly slowly (say every hour), but the query is repeated fairly often (say every second).
- Solution:
 - Use the new optimizer hint `/*+ result_cache */` in your query:

```
SELECT /*+ result_cache */  
    AVG(cust_credit_limit), cust_state_province  
FROM sh.customers  
GROUP BY cust_state_province;
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

SQL result caching is useful when your queries need to analyze a large number of rows to return a small number of rows or a single row.

Two new optimizer hints are available to turn on and turn off SQL result caching:

```
/*+ result_cache */  
/*+ no_result_cache */
```

These hints enable you to override the settings of the `RESULT_CACHE_MODE` initialization parameter.

You can execute `DBMS_RESULT_CACHE.MEMORY_REPORT` to produce a memory usage report of the result cache.

Clearing the Shared Pool and Result Cache

```
--- flush.sql
--- Start with a clean slate. Flush the cache and shared pool.
--- Verify that memory was released.
SET ECHO ON
SET FEEDBACK 1
SET SERVEROUTPUT ON

execute dbms_result_cache.flush
alter system flush shared_pool
/
execute dbms_result_cache.memory_report
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

To understand the use of query result cache, ensure that you are using clean, new data. Connect to the database as SYS.

Clear the shared pool and the result cache by executing the code shown in the slide.

Examining the Memory Cache

```
--- flush.sql  
--- Start with a clean slate. Flush the cache and shared pool.  
--- Verify that memory was released.  
SET ECHO ON  
SET FEEDBACK 1  
SET SERVEROUTPUT ON  
  
execute dbms_result_cache.flush  
alter system flush shared_pool  
/  
execute dbms_result_cache.memory_report
```

```
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 = 1792K bytes (1792 blocks)  
Maximum Result Size = 89K bytes (89 blocks)  
[Memory]  
Total Memory = 9440 bytes [0.004% of the Shared Pool]  
... Fixed Memory = 9440 bytes [0.004% of the Shared Pool]  
... Dynamic Memory = 0 bytes [0.000% of the Shared Pool]
```

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Examine the memory cache by executing the code shown in the slide. Cache memory is 0 bytes, because nothing has yet been cached.

Examining the Execution Plan for a Query

```

--- plan_query1.sql
--- Generate the execution plan.
--- (The query name Q1 is optional)
explain plan for
select /*+ result_cache q_name(Q1) */ * from orders;

--- Display the execution plan.
select plan_table_output from
table(dbms_xplan.display('plan table',null,'serial'));

```

The optimizer hint places the query in the result cache.

```

explain plan succeeded.
PLAN_TABLE_OUTPUT
-----
Plan hash value: 1275100350

| Id | Operation          | Name           | Rows | Bytes | Cost (%CPU) | Time      |
| 0  | SELECT STATEMENT   |                | 80   | 2960  | 3 (0)       | 00:00:01  |
| 1  |  RESULT CACHE     | 979kh5kqt... | 80   | 2960  | 3 (0)       | 00:00:01  |
| 2  | TABLE ACCESS FULL | ORDERS        | 80   | 2960  |            |           |

Result Cache Information (identified by operation id):
-----
1 - column-count=8; dependencies=(OE.ORDERS); name="select /*+ result_cache q_name(Q1) */ * from orders"

```

14 rows selected

Verify that the query result is placed in the result cache.

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You examine the execution plan for two queries, and then execute both the queries. After executing both the queries, you view the memory allocation and usage statistics.

First, execute the code shown in the slide and examine the execution plan for the first query. The query uses the RESULT_CACHE optimizer hint.

Examining Another Execution Plan

```

--- plan_query2.sql
set echo on
--- Generate the execution plan.(The query name Q2 is optional)
explain plan for
select c.customer_id, o.ord_count
  from (select /*+ result_cache q_name(Q2) */
            customer_id, count(*) ord_count
           from orders
          group by customer_id) o, customers c
         where o.customer_id = c.customer_id;

--- Display the execution plan.
--- using the code in ORACLE_HOME/rdbms/admin/utlxpls

select plan_table_output from table(dbms_xplan.display('plan_table',
null,'serial'));

```

```

...
Result Cache Information (identified by operation id):
-----
3 - column-count=2; dependencies=(OE.ORDERS); name="select /*+ result_cache q_name(Q2) */
customer_id, count(*) ord_count
from orders
group by customer_id"

25 rows selected

```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Execute the code shown in the slide and examine the execution plan for the second query. This query also uses the RESULT_CACHE optimizer hint.

```

--- Generate the execution plan.(The query name Q2 is
optional)
explain plan for
select c.customer_id, o.ord_count
  from (select /*+ result_cache q_name(Q2) */
            customer_id, count(*) ord_count
           from orders
          group by customer_id) o, customers c
         where o.customer_id = c.customer_id
explain plan succeeded.
PLAN_TABLE_OUTPUT
-----
Plan hash value: 2892511806
-----
| Id  | Operation          | Name   |
Rows  | Bytes | Cost (%CPU) | Time   |
-----| 0   | SELECT STATEMENT |          |
| 37  | 1110  |      1  (0) | 00:00:01 |
-----
```

1	NESTED LOOPS		37
1110	1 (0)	00:00:01	
*	VIEW		37
962	1 (0)	00:00:01	
3	RESULT CACHE	3x0x88r47u2jga7fzz398g0x1s	
4	HASH GROUP BY		37
148	1 (0)	00:00:01	
5	INDEX FULL SCAN	ORD_CUSTOMER_IX	80
320	1 (0)	00:00:01	
*	INDEX UNIQUE SCAN	CUSTOMERS_PK	1
4	0 (0)	00:00:01	

Predicate Information (identified by operation id) :

```
2 - filter("O"."CUSTOMER_ID">>0)
6 - access("O"."CUSTOMER_ID"="C"."CUSTOMER_ID")
      filter("C"."CUSTOMER_ID">>0)
```

Result Cache Information (identified by operation id) :

```
3 - column-count=2; dependencies=(OE.ORDERS); name="select
/*+ result_cache q_name(Q2) */
      customer_id, count(*) ord_count
      from orders
      group by customer_id"
```

25 rows selected

Executing Both Queries

```
-- query3.sql
-- Cache result of both queries, then use the cached result.
Set timing on
set echo on
select /*+ result_cache q_name(Q1) */ * from orders;
select c.customer_id, o.ord_count
  from (select /*+ result_cache q_name(Q3) */
            customer_id, count(*) ord_count
           from orders
          group by customer_id) o, customers c
         where o.customer_id = c.customer_id;
```

```
...
2452    07-OCT-07 02.59.43.462632000 AM direct      149   5    12589          159
2457    01-NOV-07 05.22.16.162632000 AM direct      118   5    21586.2          159
80 rows selected

81ms elapsed

select c.customer_id, o.ord_count
  from (select /*+ result_cache q_name(Q3) */
            customer_id, count(*) ord_count
           from orders
          group by customer_id) o, customers c
         where o.customer_id = c.customer_id

CUSTOMER_ID          ORD_COUNT
-----  -----
123                  1
151                  1
...
23ms elapsed
```

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You have examined the execution plan for both the queries. Now execute both the queries by executing the code shown in the slide.

Viewing Cache Results Created

```
col name format a55
select * from v$result_cache_statistics
/
```

#	ID	NAME	VALUE
1		Block Size (Bytes)	1024
2		Block Count Maximum	1792
3		Block Count Current	32
4		Result Size Maximum (Blocks)	89
	5	Create Count Success	2
	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



Number of cache results successfully created



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The next step is to run a query against V\$RESULT_CACHE_STATISTICS to view the memory allocation and usage statistics. You can view the memory allocation by executing the code as shown in the slide.

Note that the CREATE COUNT SUCCESS statistic has a value of 2, which is the number of cache results that were successfully created (one for each query statement).

In the next steps, you re-execute the queries and view the cache results found.

Re-Executing Both Queries

```
--- query3.sql
--- Cache result of both queries, then use the cached result.
set echo on
select /*+ result_cache q_name(Q1) */ * from orders

select c.customer_id, o.ord_count
  from (select /*+ result_cache q_name(Q3) */
            customer_id, count(*) ord_count
           from orders
          group by customer_id) o, customers c
         where o.customer_id = c.customer_id
    /
```

```
...
2452 07-OCT-07 02.59.43.462632000 AM direct      149   5   12589      159
2457 01-NOV-07 05.22.16.162632000 AM direct      118   5   21586.2      159
80 rows selected
```

46ms elapsed

```
select c.customer_id, o.ord_count
  from (select /*+ result_cache q_name(Q3) */
            customer_id, count(*) ord_count
           from orders
          group by customer_id) o, customers c
         where o.customer_id = c.customer_id
```

CUSTOMER_ID	ORD_COUNT
123	1
151	1

8ms elapsed

Note that the query runs faster after caching. (Earlier timings were 81ms and 23 ms.)

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Execute both the queries shown in the slide. Note that the query runs faster after caching.

Viewing Cache Results Found

```
col name format a55
select * from v$result_cache_statistics
/
```

ID	NAME	VALUE
1	Block Size (Bytes)	1024
2	Block Count Maximum	1792
3	Block Count Current	32
4	Result Size Maximum (Blocks)	89
5	Create Count Success	2
6	Create Count Failure	0
7	Find Count	2
8	Invalidation Count	0
9	Delete Count Invalid	0
10	Delete Count Valid	0
11	Hash Chain Length	1



Number of cache results successfully found

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Query V\$RESULT_CACHE_STATISTICS again to view memory allocation and usage statistics. Do this by again executing the code shown in the slide.

Note that the FIND COUNT statistic now has a value of 2. This is the number of cache results that were successfully found (one for each query statement).

Lesson Agenda

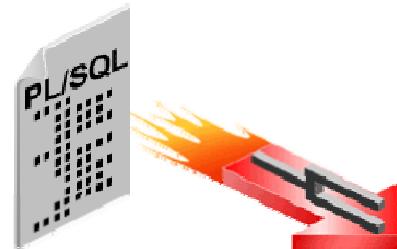
- Improving memory usage by caching SQL result sets
 - Enabling the query result cache
 - Using the DBMS_RESULT_CACHE package
- Implementing SQL query result caching
 - Writing queries that use the result cache hint
- Using PL/SQL function result caching
 - Setting up PL/SQL functions to use PL/SQL result caching
 - Implementing PL/SQL function result caching
 - Invoker's rights function result cache



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

PL/SQL Function Result Cache

- Definition:
 - Enables data that is stored in cache to be shared across sessions
 - Stores the function result cache in an SGA, making it available to any session that runs your application
- Benefits:
 - Improved performance
 - Improved scalability



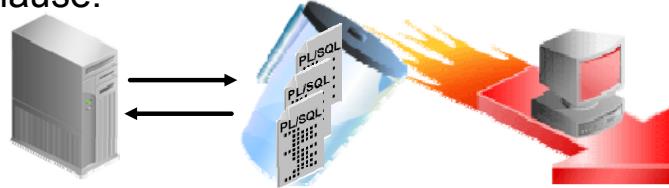
ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Starting with Oracle Database 11g, you can use the result-caching mechanism of the PL/SQL cross-session function. This caching mechanism provides you with a language-supported and system-managed means for storing the results of PL/SQL functions in an SGA, which is available to every session that runs your application. The caching mechanism is both efficient and easy to use, and it relieves you of the burden of designing and developing your own caches and cache-management policies.

Marking PL/SQL Function Results to Be Cached

- Scenario:
 - You need a PL/SQL function that derives a complex metric.
 - The data that your function calculates changes slowly, but the function is frequently called.
- Solution:
 - Use the new RESULT_CACHE clause in your function definition.
 - You can also have the cache purged when a dependent table experiences a DML operation, by using the RELIES_ON clause.



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

To enable result caching for a function, use the RESULT_CACHE clause in your PL/SQL function. Using this clause results in the following:

- If a result-cached function is called, the system checks the cache.
- If the cache contains the result from a previous call to the function with the same parameter values, the system returns the cached result to the caller and does not re-execute the function body.
- If the cache does not contain the result, the system executes the function body and adds the result (for these parameter values) to the cache before returning control to the caller.

The cache can accumulate many results—one result for every unique combination of parameter values with which each result-cached function was called. If the system needs more memory, it ages out (deletes) one or more cached results.

You can specify the database objects that are used to compute a cached result, so that if any of them is updated, the cached result becomes invalid and must be recomputed.

The best candidates for result caching are functions that are called frequently but depend on information that changes infrequently or never.

Clearing the Shared Pool and Result Cache

```
--- flush.sql
--- Start with a clean slate. Flush the cache and shared pool.
--- Verify that memory was released.
SET ECHO ON
SET FEEDBACK 1
SET SERVEROUTPUT ON

execute dbms_result_cache.flush
alter system flush shared_pool
/
execute dbms_result_cache.memory_report
```

```
Result Cache Memory Report
[Parameters]
Block Size      = 1K bytes
Maximum Cache Size = 1056K bytes (1056 blocks)
Maximum Result Size = 52K bytes (52 blocks)
[Memory]
Total Memory = 5140 bytes [0.003% of the Shared Pool]
... Fixed Memory = 5140 bytes [0.003% of the Shared Pool]
... Dynamic Memory = 0 bytes [0.000% of the Shared Pool]
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

To understand the use of PL/SQL function result caching, clear the shared pool and the result cache again by performing the steps shown in the slide.

Lesson Agenda

- Improving memory usage by caching SQL result sets
 - Enabling the query result cache
 - Using the DBMS_RESULT_CACHE package
- Implementing SQL query result caching
 - Writing queries that use the result cache hint
- Using PL/SQL function result caching
 - Setting up PL/SQL functions to use PL/SQL result caching
 - Implementing PL/SQL function result caching
 - Invoker's rights function result cache



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Creating a PL/SQL Function by Using the RESULT_CACHE Clause

- Include the RESULT_CACHE option in the function definition.
- Optionally, include the RELIES_ON clause.

```
CREATE OR REPLACE FUNCTION ORD_COUNT(cust_no number)
RETURN NUMBER
RESULT_CACHE RELIES_ON (orders)
IS
  v_count NUMBER;
BEGIN
  SELECT COUNT(*) INTO v_count
  FROM orders
  WHERE customer_id = cust_no;

  return v_count;
end;
```

Specifies that the result should be cached

Specifies the table upon which the function relies



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

When writing code for the PL/SQL result cache option, you need to:

- Include the RESULT_CACHE option in the function declaration section of a package
- Include the RESULT_CACHE option in the function definition
- Optionally, include the RELIES_ON clause to specify tables or views on which the function results depend

In the example shown in the slide, the ORD_COUNT function has result caching enabled through the RESULT_CACHE option in the function declaration. In this example, the RELIES_ON clause is used to identify the ORDERS table on which the function results depend.

You can also run DBMS_RESULT_CACHE.MEMORY_REPORT to view the result cache memory results.

For more information about result caching in Oracle Database, review the Oracle by Example tutorial, which is available at:

http://www.oracle.com/webfolder/technetwork/tutorials/obe/db/11g/r1/prod/perform/rescache/res_cache.htm

Lesson Agenda

- Improving memory usage by caching SQL result sets
 - Enabling the query result cache
 - Using the DBMS_RESULT_CACHE package
- Implementing SQL query result caching
 - Writing queries that use the result cache hint
- Using PL/SQL function result caching
 - Setting up PL/SQL functions to use PL/SQL result caching
 - **Implementing PL/SQL function result caching**
 - Invoker's rights function result cache



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Calling the PL/SQL Function Inside a Query

```
select cust_last_name, ord_count(customer_id) no_of_orders  
from customers  
where cust_last_name = 'MacGraw'
```

CUST_LAST_NAME	NO_OF_ORDERS
MacGraw	3



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Call the ORD_COUNT PL/SQL function inside a query by performing the query shown in the slide.

Verifying Memory Allocation

```
--- Establish the cache content
set serveroutput on
execute
dbms_result_cache.memory_report
```

```
anonymous block completed
Result Cache Memory Report
[Parameters]
Block Size      = 1K bytes
Maximum Cache Size = 1792K bytes (1792 blocks)
Maximum Result Size = 89K bytes (89 blocks)
[Memory]
Total Memory = 107836 bytes [0.050% of the Shared Pool]
... Fixed Memory = 9440 bytes [0.004% of the Shared Pool]
... Dynamic Memory = 98396 bytes [0.046% of the Shared Pool]
..... Overhead = 65628 bytes
..... Cache Memory = 32K bytes (32 blocks)
..... Unused Memory = 29 blocks
..... Used Memory = 3 blocks
..... Dependencies = 2 blocks (2 count)
..... Results = 1 blocks
..... PLSQL    = 1 blocks (1 count)
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

To verify that memory was allocated, execute `dbms_result_cache.memory_report` as shown in the slide to view the memory allocation statistics.

Viewing Cache Results Created

```
col name format a55
select * from v$result_cache_statistics
/
```

ID	NAME	VALUE
1	Block Size (Bytes)	1024
2	Block Count Maximum	1792
3	Block Count Current	32
4	Result Size Maximum (Blocks)	89
5	Create Count Success	1
6	Create Count Failure	0
7	Find Count	1
8	Invalidation Count	0
9	Delete Count Invalid	0
10	Delete Count Valid	0
11	Hash Chain Length	1



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The next step is to query V\$RESULT_CACHE_STATISTICS to view the memory allocation and usage statistics. Do this by performing the steps shown in the slide.

Note that the CREATE COUNT SUCCESS statistic has a value of 1. This is the number of cache results that were successfully created (one for each query statement).

Calling the PL/SQL Function Again

```
select cust_last_name, ord_count(customer_id) no_of_orders  
from customers  
where cust_last_name = 'MacGraw'
```

CUST_LAST_NAME	NO_OF_ORDERS
MacGraw	3



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Run the query again, as shown in the slide.

Viewing Cache Results Found

```
col name format a55
select * from v$result_cache_statistics
/
```

ID	NAME	VALUE
1	Block Size (Bytes)	1024
2	Block Count Maximum	1792
3	Block Count Current	32
4	Result Size Maximum (Blocks)	89
5	Create Count Success	1
6	Create Count Failure	0
7	Find Count	1
8	Invalidation Count	0
9	Delete Count Invalid	0
10	Delete Count Valid	0
11	Hash Chain Length	1



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Query V\$RESULT_CACHE_STATISTICS again to view the memory allocation and usage statistics. Do this by performing the steps shown in the slide.

Note that the FIND COUNT statistic now has a value of 1. This is the number of cache results that were successfully found (one for each query statement).

Confirming That the Cached Result Was Used

```
select type, namespace,status, scan_count,name  
from v$result_cache_objects  
/
```

Type	Namespace	Status	Scan_Count	Name
Dependency (null)		Published	0	OE.ORDERS
Dependency (null)		Published	0	OE.ORD_COUNT
Result PLSQL		Published	1	"OE"."ORD_COUNT":8."ORD_COUNT"#fac892c7867b54c6 #1



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Query V\$RESULT_CACHE_OBJECTS to confirm that the cached result was used, by performing the steps shown in the slide.

Lesson Agenda

- Improving memory usage by caching SQL result sets
 - Enabling the query result cache
 - Using the DBMS_RESULT_CACHE package
- Implementing SQL query result caching
 - Writing queries that use the result cache hint
- Using PL/SQL function result caching
 - Setting up PL/SQL functions to use PL/SQL result caching
 - Implementing PL/SQL function result caching
 - Invoker's rights function result cache



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Definer's Rights Versus Invoker's Rights

Definer's rights:

- Programs execute with the privileges of the creating user.
- User does not require privileges on underlying objects that the procedure accesses. User requires privilege only to execute a procedure.

Invoker's rights:

- Programs execute with the privileges of the calling user.
- User requires privileges on the underlying objects that the procedure accesses.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Definer's Rights Model

By default, all programs are executed with the privileges of the user who created the subprogram. This is known as the definer's rights model, which:

- Provides a caller of the program the privilege to execute the procedure, but no privileges on the underlying objects that the procedure accesses
- Requires the owner to have all the necessary object privileges for the objects that the procedure references

For example, if user Scott creates a PL/SQL subprogram `get_employees` that is subsequently invoked by Sarah, then the `get_employees` procedure runs with the privileges of the definer Scott.

Invoker's Rights Model

In the invoker's rights model, which was introduced in Oracle8i, programs are executed with the privileges of the calling user. A user of a procedure running with invoker's rights requires privileges on the underlying objects that the procedure references.

For example, if Scott's PL/SQL subprogram `get_employees` is invoked by Sarah, the `get_employees` procedure runs with the privileges of the invoker Sarah.

Defining Invoker's Right and Definer's Right

- A unit whose AUTHID value is CURRENT_USER is called an invoker's rights unit, or IR unit.
- A unit whose AUTHID value is DEFINER is called a definer's rights unit, or DR unit.
- PL/SQL units and schema objects for which you cannot specify an AUTHID value work as follows:

View Name	Column Name
Anonymous block	IR unit
BEQUEATH CURRENT_USER view	Quite similar to an IR unit
BEQUEATH DEFINER view	DR unit
Trigger	DR unit



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The AUTHID property of a stored PL/SQL unit affects the name resolution and privilege checking of SQL statements that the unit issues at run time. The AUTHID property does not affect compilation, and has no meaning for units that have no code, such as collection types. AUTHID property values are exposed in the static data dictionary view *_PROCEDURES. For units for which AUTHID has meaning, the view shows the value CURRENT_USER or DEFINER; for other units, the view shows NULL.

The table in the slide shows the behavior of PL/SQL units and schema units for which you cannot specify an AUTHID.

The PL/SQL function declaration with both AUTHID CURRENT_USER and RESULT CACHE is explained in the next few slides.

Specifying Invoker's Rights: Setting AUTHID to CURRENT_USER

```
CREATE OR REPLACE PROCEDURE add_dept(
    p_id NUMBER, p_name VARCHAR2) AUTHID CURRENT_USER IS
BEGIN
    INSERT INTO departments
    VALUES (p_id, p_name, NULL, NULL);
END;
```

When used with stand-alone functions, procedures, or packages:

- Names used in queries, DML, Native Dynamic SQL, and DBMS_SQL package are resolved in the invoker's schema
- Calls to other packages, functions, and procedures are resolved in the definer's schema



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You can set the invoker's rights for different PL/SQL subprogram constructs as follows:

```
CREATE FUNCTION name RETURN type AUTHID CURRENT_USER IS...
CREATE PROCEDURE name AUTHID CURRENT_USER IS...
CREATE PACKAGE name AUTHID CURRENT_USER IS...
CREATE TYPE name AUTHID CURRENT_USER IS OBJECT...
```

The default is AUTHID DEFINER, which specifies that the subprogram executes with the privileges of its owner. Most supplied PL/SQL packages, such as DBMS_LOB, DBMS_ROWID, and so on, are invoker's rights packages.

Name Resolution

For a definer's rights procedure, all external references are resolved in the definer's schema. For an invoker's rights procedure, the resolution of external references depends on the kind of statement in which they appear:

- Names used in queries, data manipulation language (DML) statements, dynamic SQL, and DBMS_SQL are resolved in the invoker's schema.
- All other statements, such as calls to packages, functions, and procedures, are resolved in the definer's schema.

Using CURRENT_USER and RESULT_CACHE

```
CREATE OR REPLACE FUNCTION get_hire_date (emp_id NUMBER)
  RETURN VARCHAR
RESULT_CACHE
AUTHID CURRENT_USER
IS
  date_hired DATE;
BEGIN
  SELECT hire_date INTO date_hired
  FROM HR.EMPLOYEES
  WHERE EMPLOYEE_ID = emp_id;
  RETURN TO_CHAR(date_hired);
END;
```

```
SQL> select get_hire_date(206) from dual;

GET_HIRE_DATE(206)
-----
07-JUN-02

SQL> select hire_date, employee_id from employees where employee_id = 206;

HIRE_DATE EMPLOYEE_ID
-----
07-JUN-02      206
```

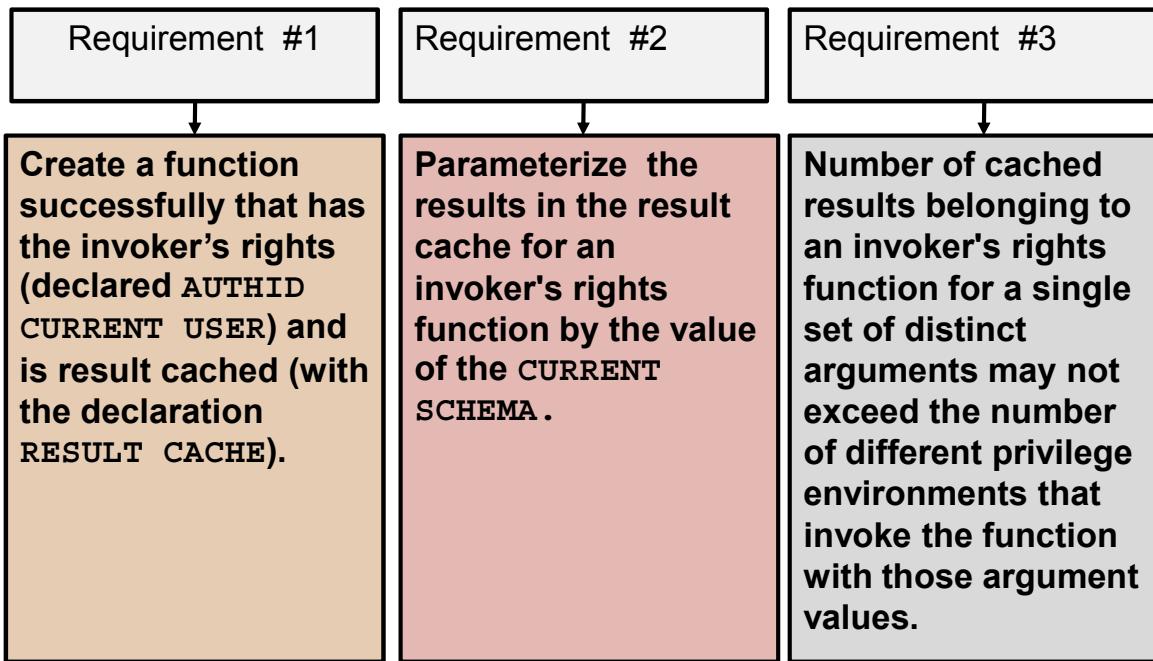
ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

With the rich new features of 12c, the invoker's rights function can be result cached. The code example in the slide depicts the creation of a `get_hire_date` function that takes the `employee_id` as a parameter and returns the hire date. Notice that the function is result cached and the `AUTHID` property is set to invoker's rights.

The `get_hire_date` function uses the `TO_CHAR` function to convert a `DATE` item to a `VARCHAR` item. The `get_hire_date` function does not specify a format mask, so the format mask defaults to the one that `NLS_DATE_FORMAT` specifies.

Ensuring That the Invoker's Rights Function is Result Cached: Requirements



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

There are a few requirements that ensure that the invoker's rights function is result cached.

You can create a function successfully that has the invoker's rights (declared AUTHID CURRENT_USER) and is result cached (with the declaration RESULT CACHE).

Results in the result cache for an invoker's rights function must be parameterized by the value of CURRENT_SCHEMA in the invoking environment if any name resolution that is sensitive to the value of this setting occurs while the result is being computed.

The number of cached results belonging to an invoker's rights function for a single set of distinct argument values may not exceed the number of different privilege environments that invoke the function with those argument values. This effectively places an upper bound on the amount of space an invoker's rights function may consume per set of distinct argument values. If any privilege-sensitive operation is performed while building a cache result for an invoker's rights function, the result must somehow be parameterized by that privilege.

Although this requirement could be met by parameterizing by various privileges directly, this project will take the somewhat more general and safer approach of parameterizing by the enabled users and roles that provide privileges used by the function.

Privilege Check for an Invoker's Rights Unit

You can use one of these statements to grant appropriate privilege:

```
GRANT INHERIT PRIVILEGES ON current_user TO PUBLIC  
GRANT INHERIT PRIVILEGES ON current_user TO unit_owner  
GRANT INHERIT ANY PRIVILEGES TO unit_owner
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

- The runtime system does a privilege check before executing an invoker's rights unit.
- The unit owner must have either the `INHERIT PRIVILEGES` privilege on the invoker, or the `INHERIT ANY PRIVILEGES` privilege.
- If the privilege check fails, the runtime system raises the `ORA-06598` error.
- Based on the unit's requirement, you can choose to grant the right privilege to execute the Invoker's Right unit.

Granting Roles to PL/SQL Packages and Stand-Alone Stored Subprograms

- The SQL GRANT command grants roles to PL/SQL packages and stand-alone subprograms:
 - Create an IR unit.
 - Grant roles using the IR unit.
- The IR unit runs with the privileges of both the invoker and the roles.



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Before Oracle Database 12c, a definer's rights (DR) unit always ran with the privileges of the definer and an invoker's rights (IR) unit always ran with the privileges of the invoker. If you wanted to create a PL/SQL unit that all users could invoke, even if their privileges were lower than yours, it had to be a DR unit. The DR unit always ran with all your privileges, regardless of which user invoked it.

As of Oracle Database 12c, you can grant roles to individual PL/SQL packages and stand-alone subprograms. Instead of a DR unit, you can create an IR unit and then grant it roles. The IR unit runs with the privileges of both the invoker and the roles, but without any additional privileges that you have.

You grant roles to an IR unit so that users with lower privileges than yours can run the unit with only the privileges needed to do so. There is no reason to grant roles to a DR unit, because its invokers run it with all your privileges.

Using the SQL GRANT command, you can grant roles to PL/SQL packages and stand-alone stored subprograms. Roles granted to a PL/SQL unit do not affect compilation. They affect the privilege checking of SQL statements that the unit issues at run time; the unit runs with the privileges of both its own roles and any other currently enabled roles.

Quiz

Which of the following statements are true?

- a. When a query is executed, the result cache is built up in the result cache memory.
- b. Subsequent executions of the same query or function can be served directly out of the cache, improving response times.
- c. This technique should not be used for SQL queries and PL/SQL functions that are executed frequently.
- d. Cached query results remains valid even after the database data accessed by the query is modified.



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Answer: a, b

Quiz

You can set the RESULT_CACHE_MODE to FORCE at the session level by using the ALTER SESSION command, so that the results of all the queries are always stored in the result cache memory.

- a. True
- b. False



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Answer: a

Quiz

You can use the DBMS_RESULT_CACHE package to:

- a. Bypass the cache
- b. Retrieve statistics on the cache memory usage
- c. Flush the cache
- d. None of the above



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Answer: a, b, c

Quiz

On querying V\$RESULT_CACHE_STATISTICS to view the memory allocation and usage statistics, the number of cache results successfully **found** is denoted by:

- a. The CREATE COUNT SUCCESS statistic
- b. The FIND COUNT statistic
- c. The INVALIDATION COUNT statistic
- d. The HASH CHAIN LENGTH statistic



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Answer: b

Quiz

You can create a function successfully that has both invoker's rights and is result cached.

- a. True
- b. False



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Answer: a

It is possible to create a unit that has both invoker's rights (declared AUTHID CURRENT USER) and is result cached (with the declaration RESULT CACHE).

Summary

In this lesson, you should have learned how to:

- Improve memory usage by caching SQL result sets
- Write queries that use the result cache hint
- Use the DBMS_RESULT_CACHE package
- Set up PL/SQL functions to use PL/SQL result caching



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In this lesson, you saw the Oracle Database 12c caching techniques that can improve performance.

Practice 9: Overview

This practice covers the following topics:

- Writing code to use SQL caching
- Writing code to use PL/SQL caching



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In this practice, you implement SQL query result caching and PL/SQL result function caching. You run scripts to measure the cache memory values, manipulate queries and functions to turn caching on and off, and then examine cache statistics. This practice uses the OE schema.

10

Analyzing PL/SQL Code

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Objectives

After completing this lesson, you should be able to do the following:

- Use the supplied packages and dictionary views to find coding information
- Determine identifier types and usages with PL/Scope
- Use the DBMS_METADATA package to obtain metadata from the data dictionary as XML or creation DDL that can be used to re-create the objects



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In this lesson, you learn how to write PL/SQL routines that analyze PL/SQL applications. You learn about the dictionary views and packages that you can use to find information within your code and to generate information about your code.

Lesson Agenda

- Running reports on source code
- Determining identifier types and usages
- Using DBMS_METADATA to retrieve object definitions
- PL/SQL Enhancements



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Finding Coding Information: Using the Dictionary Views

Dictionary View	Description
ALL_ARGUMENTS	Includes information about the parameters for the procedures and functions that you can call
ALL_SOURCE	Includes the lines of source code for all programs you modify
ALL PROCEDURES	Contains the list of procedures and functions that you can execute
ALL_DEPENDENCIES	Is one of the several views that give you information about the dependencies between database objects



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The Oracle dictionary views store information about your compiled PL/SQL code. You can write SQL statements against the views to find information about your code.

Finding Coding Information: Using the Supplied Packages

`dbms_describe`

`dbms_utility`



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You can also use the Oracle-supplied `DBMS_DESCRIBE` package to obtain information about a PL/SQL object. The package contains the `DESCRIBE PROCEDURE` procedure, which provides a brief description of a PL/SQL stored procedure. It takes the name of a stored procedure and returns information about each parameter of that procedure.

You can use the `DBMS_UTILITIY`-supplied package to follow a call stack and an exception stack.

Finding Coding Information

Find all instances of CHAR in your code:

```
SELECT NAME, line, text
FROM      user_source
WHERE     INSTR (UPPER(text), ' CHAR') > 0
          OR INSTR (UPPER(text), ' CHAR(') > 0
          OR INSTR (UPPER(text), ' CHAR ()') > 0;

NAME          LINE  TEXT
-----
CUST_ADDRESS_TYP    6      , country_id      CHAR(2)
...
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You may want to find all occurrences of the CHAR data type. The CHAR data type is fixed in length and can cause false negatives on comparisons with VARCHAR2 strings. By finding the CHAR data type, you can modify the object, if appropriate, and change it to VARCHAR2.

Finding Coding Information

Create a package with various queries that you can easily call:

```
CREATE OR REPLACE PACKAGE query_code_pkg
AUTHID CURRENT_USER
IS
  PROCEDURE find_text_in_code (str IN VARCHAR2);
  PROCEDURE encap_compliance ;
END query_code_pkg;
/
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

A better idea is to create a package to hold various queries that you can easily call. `QUERY_CODE_PKG` will hold two validation procedures:

- **FIND_TEXT_IN_CODE**: Displays all programs with a specified character string. It queries `USER_SOURCE` to find occurrences of any text string. The text string is passed as a parameter. For efficiency, the `BULK COLLECT` statement is used to retrieve all matching rows into the collection variable.
- **ENCAP_COMPLIANCE**: Identifies those programs that reference a table directly. This procedure queries the `ALL_DEPENDENCIES` view to find the PL/SQL code objects that directly reference a table or a view.

You can also include a procedure to validate a set of standards for exception handling.

QUERY_CODE_PKG Code

```

CREATE OR REPLACE PACKAGE BODY query_code_pkg IS
  PROCEDURE find_text_in_code (str IN VARCHAR2)
  IS
    TYPE info_rt IS RECORD (NAME user_source.NAME%TYPE,
                           text user_source.text%TYPE );
    TYPE info_aat IS TABLE OF info_rt INDEX BY PLS_INTEGER;
    info_aa info_aat;
  BEGIN
    SELECT NAME || '-' || line, text
    BULK COLLECT INTO info_aa FROM user_source
      WHERE UPPER (text) LIKE '%' || UPPER (str) || '%'
        AND NAME != 'VALSTD' AND NAME != 'ERRNUMS';
    DBMS_OUTPUT.PUT_LINE ('Checking for presence of ' ||
                          str || ':');
    FOR indx IN info_aa.FIRST .. info_aa.LAST LOOP
      DBMS_OUTPUT.PUT_LINE (
        info_aa (indx).NAME || ',' || info_aa (indx).text);
    END LOOP;
  END find_text_in_code;

  PROCEDURE encaps_compliance IS
    SUBTYPE qualified_name_t IS VARCHAR2 (200);
    TYPE refby_rt IS RECORD (NAME qualified_name_t,
                             referenced_by qualified_name_t );
    TYPE refby_aat IS TABLE OF refby_rt INDEX BY PLS_INTEGER;
    refby_aa refby_aat;
  BEGIN
    SELECT owner || '.' || NAME refs_table
      , referenced_owner || '.' || referenced_name
      AS table_referenced
    BULK COLLECT INTO refby_aa
      FROM all_dependencies
      WHERE owner = USER
        AND TYPE IN ('PACKAGE', 'PACKAGE BODY',
                     'PROCEDURE', 'FUNCTION')
        AND referenced_type IN ('TABLE', 'VIEW')
        AND referenced_owner NOT IN ('SYS', 'SYSTEM')
    ORDER BY owner, NAME, referenced_owner, referenced_name;
    DBMS_OUTPUT.PUT_LINE ('Programs that reference
                           tables or views');
    FOR indx IN refby_aa.FIRST .. refby_aa.LAST LOOP
      DBMS_OUTPUT.PUT_LINE (refby_aa (indx).NAME || ',' ||
                            refby_aa (indx).referenced_by);
    END LOOP;
  END encaps_compliance;
END query_code_pkg;
/

```

Finding Coding Information

```
EXECUTE query_code_pkg.encap_compliance  
Programs that reference tables or views  
OE.CREDIT_CARD_PKG, OE.CUSTOMERS  
OE.LOAD_PRODUCT_IMAGE, OE.PRODUCT_INFORMATION  
OE.SET_VIDEO, OE.CUSTOMERS  
OE.WRITE_LOB, OE.PRODUCT_DESCRIPTIONS  
...
```

1

PL/SQL procedure successfully completed.

```
EXECUTE query_code_pkg.find_text_in_code('customers')  
  
Checking for presence of customers:  
CREDIT_CARD_PKG-12,           FROM customers  
CREDIT_CARD_PKG-36,           UPDATE customers  
CREDIT_CARD_PKG-41,           UPDATE customers  
SET_VIDEO-6,                 SELECT cust_first_name FROM customers  
SET_VIDEO-13,                UPDATE customers SET video = file_ptr  
...
```

2

PL/SQL procedure successfully completed.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

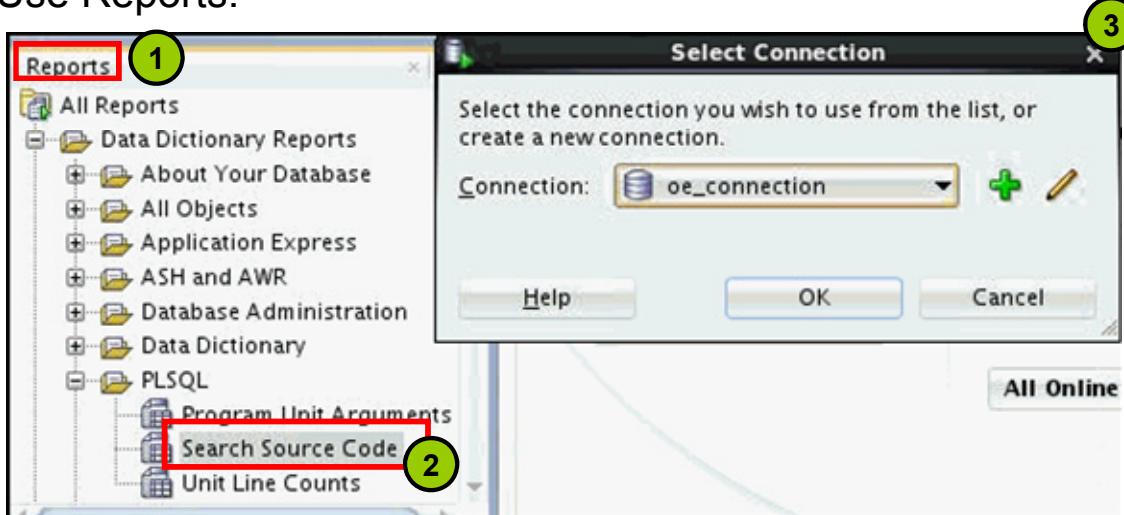
QUERY_CODE_PKG: Examples

In the first example in the slide, the ENCAP_COMPLIANCE procedure displays all PL/SQL code objects that reference a table or view directly. Both the code name and table or view name are listed in the output.

In the second example, the FIND_TEXT_IN_CODE procedure returns all PL/SQL code objects that contain the “customers” text string. The code name, line number, and line are listed in the output.

Using SQL Developer to Find Coding Information

Use Reports:



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

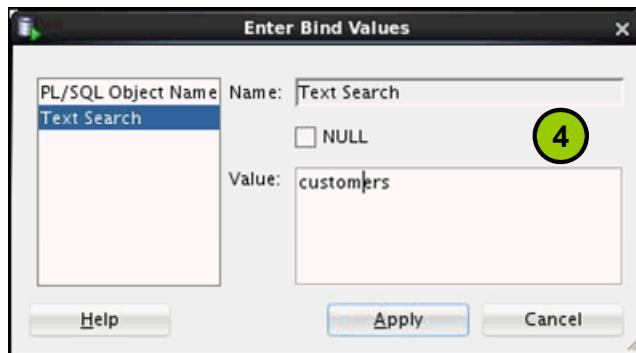
SQL Developer comes with predefined reports that you can use to find information about PL/SQL coding.

If you want to find the occurrence of a text string or an object name, use the Reports feature.

1. Select the Reports tabbed page in SQL Developer.
2. Expand the PL/SQL node and select the Search Source Code option.
3. Enter the connection information.

Using SQL Developer to Find Coding Information

Enter a text string or an object name.



Results:

Owner	PL/SQL Object Name	Type	Line	Text
OE	CHANGE_CREDIT	PROCEDURE	2	(p_in_id IN customers.customer_id%TYPE,
OE	CHANGE_CREDIT	PROCEDURE	6	UPDATE customers
OE	CREDIT_CARD_PKG	PACKAGE BODY	12	FROM customers
OE	CREDIT_CARD_PKG	PACKAGE BODY	36	UPDATE customers
OE	CREDIT_CARD_PKG	PACKAGE BODY	41	UPDATE customers
OE	CUST_DATA	PACKAGE BODY	8	FROM customers

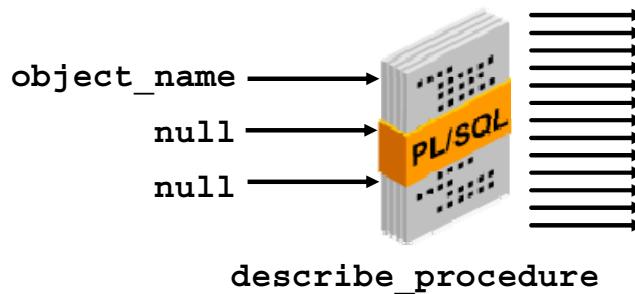
ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

4. Select either an Object Name or a Text Search for the search type. In the Value field, enter the text string that you want to search in your PL/SQL source code for the connection that you specified in step 3.
5. View the results of your search.

Using DBMS_DESCRIBE

- Can be used to retrieve information about a PL/SQL object
- Contains one procedure: DESCRIBE_PROCEDURE
- Includes:
 - Three scalar IN parameters
 - One scalar OUT parameter
 - 12 associative array OUT parameters



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You can use the DBMS_DESCRIBE package to find information about your procedures. It contains one procedure named DESCRIBE_PROCEDURE. The DESCRIBE_PROCEDURE routine accepts the name of the procedure that you are enquiring about. There are two other IN parameters. Both must be either NULL or an empty string. These two parameters are reserved.

The DBMS_DESCRIBE package returns detailed parameter information in a set of associative arrays. The details are numerically coded. You can find the following information from the results returned:

- **Overload:** If overloaded, holds a value for each version of the procedure
- **Position:** Position of the argument in the parameter list. 0 is reserved for the RETURN information of a function.
- **Level:** For composite types only; it holds the level of the data type
- **Argument name:** Name of the argument
- **Data type:** A numerically coded value representing a data type
- **Default value:** 0 for no default value; 1 if the argument has a default value
- **Parameter mode:** 0 = IN, 1 = OUT, 2 = IN OUT

Note: This is not the complete list of values returned from the DESCRIBE_PROCEDURE routine. For a complete list, see the *PL/SQL Packages and Types Reference 12c Release 1* reference manual.

Using DBMS_DESCRIBE

Create a package to call the DBMS_DESCRIBE.DESCRIBE PROCEDURE routine:

```
CREATE OR REPLACE PACKAGE use_dbms_describe
IS
  PROCEDURE get_data (p_obj_name VARCHAR2);
END use_dbms_describe;
/
```

```
EXEC use_dbms_describe.get_data('query_code_pkg.find_text_in_code')
```

anonymous block completed

Name	Mode	Position	Datatype
STR	0	1	1
Name	Mode	Position	Datatype
STR	0	1	1



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Because DESCRIBE PROCEDURE returns information about your parameters in a set of associative arrays, it is easy to define a package to call and handle the information returned from it.

In the first example in the slide, the specification for the USE_DBMS_DESCRIBE package is defined. This package holds one procedure, GET_DATA. The GET_DATA routine calls the DBMS_DESCRIBE.DESCRIBE PROCEDURE routine. The implementation of the USE_DBMS_DESCRIBE package is shown on the next page. Note that several associative array variables are defined to hold the values returned via the OUT parameters from the DESCRIBE PROCEDURE routine. Each of these arrays uses the predefined package types:

```
TYPE VARCHAR2_TABLE IS TABLE OF VARCHAR2(30)
  INDEX BY BINARY_INTEGER;
TYPE NUMBER_TABLE IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
```

In the call to the DESCRIBE PROCEDURE routine, you need to pass three parameters: the name of the procedure that you are enquiring about and two null values. These null values are reserved for future use.

In the second example in the slide, the results are displayed for the parameters of the query_code_pkg.find_text_in_code function. A data type of 1 indicates that it is a VARCHAR2 data type.

Calling DBMS_DESCRIBE.DESCRIBE PROCEDURE

```

CREATE OR REPLACE PACKAGE use_dbms_describe IS
  PROCEDURE get_data (p_obj_name VARCHAR2);
END use_dbms_describe;
/
CREATE OR REPLACE PACKAGE BODY use_dbms_describe IS
  PROCEDURE get_data (p_obj_name VARCHAR2)
  IS
    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
    DBMS_DESCRIBE.DESCRIBE_PROCEDURE
      (p_obj_name, null, null, -- these are the 3 in parameters
       v_overload, v_position, v_level, v_arg_name,
       v_datatype, v_def_value, v_in_out, v_length,
       v_precision, v_scale, v_radix, v_spare, null);
    IF v_in_out.FIRST IS NULL THEN
      DBMS_OUTPUT.PUT_LINE ('No arguments to report.');
    ELSE
      DBMS_OUTPUT.PUT
        ('Name                                Mode');
      DBMS_OUTPUT.PUT_LINE(' Position      Datatype ');
      FOR i IN v_arg_name.FIRST .. v_arg_name.LAST LOOP
        IF v_position(i) = 0 THEN
          DBMS_OUTPUT.PUT('This is the RETURN data for
                           the function: ');
        ELSE
          DBMS_OUTPUT.PUT (
            rpad(v_arg_name(i), LENGTH(v_arg_name(i)) +
                  42-LENGTH(v_arg_name(i)), ' '));
        END IF;
        DBMS_OUTPUT.PUT( '      ' ||
          v_in_out(i) || '      ' || v_position(i) ||
          '      ' || v_datatype(i));
        DBMS_OUTPUT.NEW_LINE;
      END LOOP;
    END IF;
  END get_data;
END use_dbms_describe;

```

Using ALL_ARGUMENTS

Query the ALL_ARGUMENTS view to find information about arguments for procedures and functions:

```
SELECT object_name, argument_name, in_out, position, data_type
  FROM all_arguments
 WHERE package_name = 'CREDIT_CARD_PKG';

OBJECT_NAME          ARGUMENT_NAME    IN_OUT      POSITION DATA_TYPE
-----              -----          -----
DISPLAY_CARD_INFO   P_CUST_ID        IN           1 NUMBER
UPDATE_CARD_INFO    O_CARD_INFO     OUT          1 OBJECT
UPDATE_CARD_INFO    P_CARD_NO       IN           3 VARCHAR2
UPDATE_CARD_INFO    P_CARD_TYPE     IN           2 VARCHAR2
UPDATE_CARD_INFO    P_CUST_ID       IN           1 NUMBER
CUST_CARD_INFO      P_CARD_INFO     IN/OUT       1 OBJECT
CUST_CARD_INFO      P_CUST_ID       IN/OUT       2 TABLE
CUST_CARD_INFO      P_CUST_ID       IN           1 NUMBER
CUST_CARD_INFO      P_CUST_ID       OUT          0 PL/SQL BOOLEAN

10 rows selected.
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You can also query the ALL_ARGUMENTS dictionary view to find information about the arguments of procedures and functions to which you have access. Like DBMS_DESCRIBE, the ALL_ARGUMENTS view returns information in textual rather than numeric form. Though there is overlap between the two, there is unique information to be found both in DBMS_DESCRIBE and ALL_ARGUMENTS.

In the example shown in the slide, the argument name, mode, position, and data type are returned for ORDERS_APP_PKG. Note the following:

- A position of 1 and a sequence and level of 0 indicates that the procedure has no arguments.
- A function that has no arguments is displayed as a single row for the RETURN clause, with a position of 0.
- The argument name for the RETURN clause is NULL.
- If the programs are overloaded, the OVERLOAD column (not shown in the slide) indicates the Nth overloading; otherwise, it is NULL.
- The DATA_LEVEL column (not shown in the slide) value of 0 identifies a parameter as it appears in the program specification.

Using ALL_ARGUMENTS

Other column information:

- Details about the data type are found in the DATA_TYPE and TYPE_ columns.
- All arguments in the parameter list are at level 0.
- For composite parameters, the individual elements of the composite are assigned levels, starting at 1.
- The POSITION-DATA_LEVEL column combination is unique only for a level 0 argument (the actual parameter, not its subtypes if it is a composite).



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The DATA_TYPE column holds the generic PL/SQL data type. To find more information about the data type, query the TYPE_ columns.

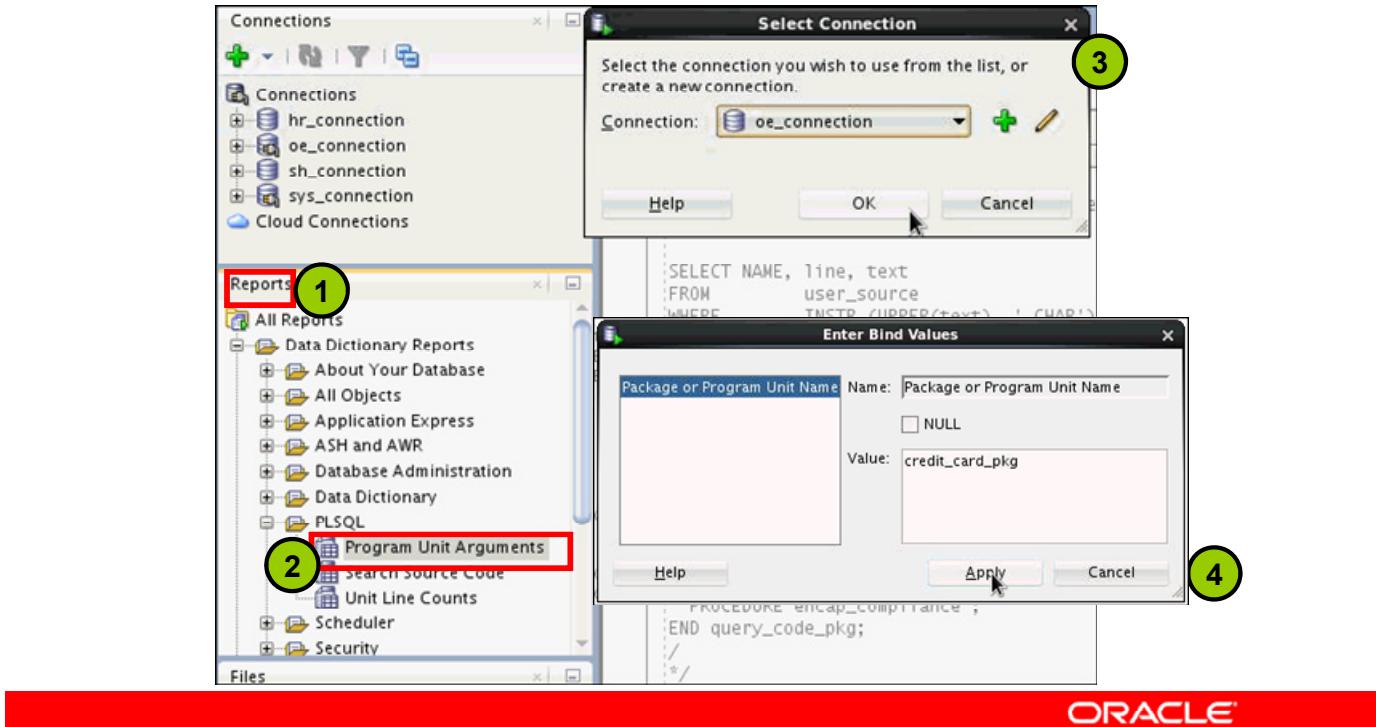
- **TYPE_NAME:** Holds the name of the type of the argument. If the type is a package local type (that is, it is declared in a package specification), this column displays the name of the package.
- **TYPE_SUBNAME:** Is relevant only for package local types. It displays the name of the type declared in the package identified in the TYPE_NAME column. For example, if the data type is a PL/SQL table, you can find out the type of the table by only looking at the TYPE_SUBNAME column.

Note: The DEFAULT_VALUE and DEFAULT_LENGTH columns are reserved for future use and do not currently contain information about a parameter's default value. However, you can use DBMS_DESCRIBE to find some default value information. In this package, the parameter DEFAULT_VALUE returns 1 if there is a default value; otherwise, it returns 0.

By combining the information from DBMS_DESCRIBE and ALL_ARGUMENTS, you can find valuable information about parameters, as well as about how your PL/SQL routines are overloaded.

Using SQL Developer to Report on Arguments

Use Reports:



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

ORACLE

SQL Developer comes with predefined reports that you can use to find PL/SQL coding information about your program unit arguments.

Use the Reports feature to find information about program unit arguments.

1. Select the Reports tabbed page in SQL Developer.
2. Expand the PL/SQL node and select Program Unit Arguments.
3. When prompted, enter the connection information.
4. Enter the name of the program unit on which you want the arguments reported.

Using SQL Developer to Report on Arguments

Results:

5

Owner	Package	Program_Unit	Position	Argument	In_Out
1 OE	CREDIT_CARD_PKG	DISPLAY_CARD_INFO	1	P_CUST_ID	In
2 OE	CREDIT_CARD_PKG	UPDATE_CARD_INFO	1	P_CUST_ID	In
3 OE	CREDIT_CARD_PKG	UPDATE_CARD_INFO	2	P_CARD_TYPE	In
4 OE	CREDIT_CARD_PKG	UPDATE_CARD_INFO	3	P_CARD_NO	In

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

5. View the results of your search.

In the example shown in the slide, the output displays the owner of the stored PL/SQL package, the name of the PL/SQL package, the names of the subroutines within the PL/SQL package, the position of the arguments within the package, the argument names, and the argument types (IN, OUT, or IN OUT).

Using DBMS_UTILITy.FORMAT_CALL_STACK

- This function returns the formatted text string of the current call stack.
- Use it to find the line of code being executed.

```
EXECUTE third_one
-----
PL/SQL Call Stack -----
object      line  object
handle     number  name
0x566ce8e0          4  procedure OE.FIRST_ONE
0x5803f7a8          5  procedure OE.SECOND_ONE
0x569c3770          6  procedure OE.THIRD_ONE
0x567ee3d0          1  anonymous block

PL/SQL procedure successfully completed.
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Another tool that is available to you is the FORMAT_CALL_STACK function in the DBMS_UTILITy-supplied package. It returns the call stack in a formatted character string. The results shown in the slide were generated based on the following routines:

```
SET SERVEROUTPUT ON
CREATE OR REPLACE PROCEDURE first_one
IS
BEGIN
    dbms_output.put_line(
        substr(dbms_utility.format_call_stack, 1, 255));
END;
/

CREATE OR REPLACE PROCEDURE second_one
IS
BEGIN
    null;
    first_one;
END;
/
```

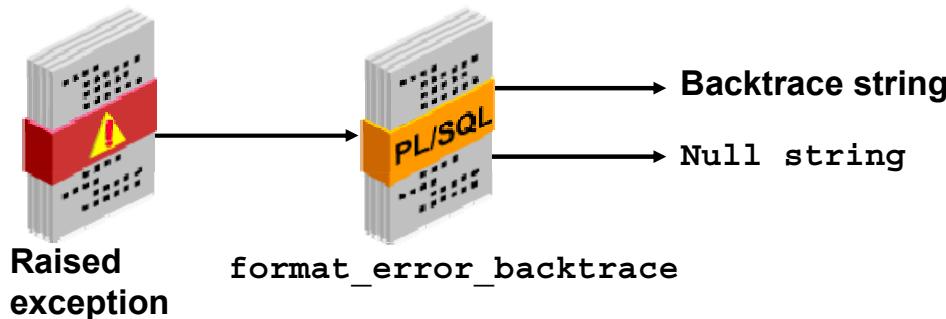
```
CREATE OR REPLACE PROCEDURE third_one
IS
BEGIN
    null;
    null;
    second_one;
END;
/
```

The output from the `FORMAT_CALL_STACK` function shows the object handle number, the line number where a routine is called from, and the routine that is called. Note that the `NULL;` statements added to the procedures are used to emphasize the line number where the routine is called from.

Finding Error Information

`DBMS_UTILITy.FORMAT_ERROR_BACKTRACE`:

- Shows you the call stack at the point where an exception is raised
- Returns:
 - The backtrace string
 - A null string if no errors are being handled



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You can use the `DBMS_UTILITy.FORMAT_ERROR_BACKTRACE` function to display the call stack at the point where an exception was raised, even if the procedure is called from an exception handler in an outer scope. The output returned is similar to the output of the `SQLERRM` function, but not subject to the same size limitation.

Using DBMS _UTILITY .FORMAT _ERROR _STACK

This function is used to format the current error stack.

Syntax:

```
DBMS _UTILITY .FORMAT _ERROR _STACK  
RETURN VARCHAR2 ;
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You can use the DBMS _UTILITY .FORMAT _ERROR _STACK function to format the current error stack. It can be used in exception handlers to view the full error stack. The function returns the error stack up to 2,000 bytes.

Finding Error Information

```
CREATE OR REPLACE PROCEDURE top_with_logging IS
  -- NOTE: SQLERRM in principle gives the same info
  -- as format_error_stack.
  -- But SQLERRM is subject to some length limits,
  -- while format_error_stack is not.
BEGIN
  P5(); -- this procedure, in turn, calls others,
        -- building a stack. P0 contains the exception
EXCEPTION
  WHEN OTHERS THEN
    log_errors ( 'Error_Stack...' || CHR(10) ||
      DBMS_UTILITY.FORMAT_ERROR_STACK() );
    log_errors ( 'Error_Backtrace...' || CHR(10) ||
      DBMS_UTILITY.FORMAT_ERROR_BACKTRACE() );
    DBMS_OUTPUT.PUT_LINE ( '-----' );
END top_with_logging;
/
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

To show you the functionality of the FORMAT_ERROR_STACK and FORMAT_ERROR_BACKTRACE functions, a TOP_WITH_LOGGING procedure is created. This procedure calls the LOG_ERRORS procedure and passes to it the results of the FORMAT_ERROR_STACK and FORMAT_ERROR_BACKTRACE functions.

The LOG_ERRORS procedure is shown on the next page.

Finding Error Information

```
CREATE OR REPLACE PROCEDURE log_errors ( i_buff IN VARCHAR2 ) IS
  g_start_pos PLS_INTEGER := 1;
  g_end_pos   PLS_INTEGER;
  FUNCTION output_one_line RETURN BOOLEAN IS
BEGIN
  g_end_pos := INSTR ( i_buff, CHR(10), g_start_pos );
  CASE g_end_pos > 0
    WHEN TRUE THEN
      DBMS_OUTPUT.PUT_LINE ( SUBSTR ( i_buff,
                                      g_start_pos, g_end_pos-g_start_pos ) );
      g_start_pos := g_end_pos+1;
      RETURN TRUE;
    WHEN FALSE THEN
      DBMS_OUTPUT.PUT_LINE ( SUBSTR ( i_buff, g_start_pos,
                                      (LENGTH(i_buff)-g_start_pos)+1 ) );
      RETURN FALSE;
    END CASE;
  END output_one_line;
BEGIN
  WHILE output_one_line() LOOP NULL;
  END LOOP;
END log_errors;
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

LOG_ERRORS: Example

This procedure takes the return results of the FORMAT_ERROR_STACK and FORMAT_ERROR_BACKTRACE functions as an IN string parameter, and reports the results back to you using DBMS_OUTPUT.PUT_LINE. The LOG_ERRORS procedure is called twice from the TOP_WITH_LOGGING procedure. The first call passes the results of FORMAT_ERROR_STACK, and the second procedure passes the results of FORMAT_ERROR_BACKTRACE.

Note: You can use UTL_FILE instead of DBMS_OUTPUT to write and format the results to a file.

Next, several procedures are created and one procedure calls another so that a stack of procedures is built. The P0 procedure raises a zero divide exception when it is invoked. The call stack is:

```
TOP_WITH_LOGGING > P5 > P4 > P3 > P2 > P1 > P0
    SET DOC OFF
    SET FEEDBACK OFF
    SET ECHO OFF

    CREATE OR REPLACE PROCEDURE P0 IS
        e_01476 EXCEPTION;
        pragma exception_init ( e_01476, -1476 );
    BEGIN
        RAISE e_01476; -- this is a zero divide error
    END P0;
    /
    CREATE OR REPLACE PROCEDURE P1 IS
    BEGIN
        P0();
    END P1;
    /
    CREATE OR REPLACE PROCEDURE P2 IS
    BEGIN
        P1();
    END P2;
    /
    CREATE OR REPLACE PROCEDURE P3 IS
    BEGIN
        P2();
    END P3;
    /
    CREATE OR REPLACE PROCEDURE P4 IS
        BEGIN P3();
    END P4;
    /
    CREATE OR REPLACE PROCEDURE P5 IS
        BEGIN P4();
    END P5;
    /
    CREATE OR REPLACE PROCEDURE top IS
    BEGIN
        P5(); -- this procedure is used to show the results
              -- without using the TOP_WITH_LOGGING routine.
    END top;
    /
    SET FEEDBACK ON
```

Finding Error Information

Results:

```
EXECUTE top_with_logging
Error_Stack...
ORA-01476: divisor is equal to zero
Error_Backtrace...
ORA-06512: at "OE.P0", line 5
ORA-06512: at "OE.P1", line 3
ORA-06512: at "OE.P2", line 3
ORA-06512: at "OE.P3", line 3
ORA-06512: at "OE.P4", line 2
ORA-06512: at "OE.P5", line 2
ORA-06512: at "OE.TOP_WITH_LOGGING", line 7
-----
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The results from executing the `TOP_WITH_LOGGING` procedure is shown in the slide. Note that the error stack displays the exception encountered. The backtrace information traces the flow of the exception to its origin.

If you execute the `TOP` procedure without using the `TOP_WITH_LOGGING` procedure, these are the results:

```
EXECUTE top
BEGIN top; END;
*
ERROR at line 1:
ORA-01476: divisor is equal to zero
ORA-06512: at "OE.P0", line 5
ORA-06512: at "OE.P1", line 3
ORA-06512: at "OE.P2", line 3
ORA-06512: at "OE.P3", line 3
ORA-06512: at "OE.P4", line 2
ORA-06512: at "OE.P5", line 2
ORA-06512: at "OE.TOP", line 3
ORA-06512: at line 1
```

Note that the line number reported is misleading.

Lesson Agenda

- Running reports on source code
- Determining identifier types and usages
- Using DBMS_METADATA to retrieve object definitions
- PL/SQL Enhancements



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

PL/Scope

Definition

- ─ Is a tool that is used for extracting, organizing, and storing user-supplied identifiers from PL/SQL source code
- ─ Works with the PL/SQL compiler
- ─ Gathers data on scoping and overloading

Benefits

- ─ Can potentially increase developer productivity
- ─ Is a valuable resource in helping developers understand source code
- ─ Can be used to build a PL/SQL IDE



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

With PL/Scope, you can produce a cross-referenced repository of PL/SQL identifiers to gather information about your PL/SQL applications.

PL/Scope has two targeted audiences:

- **PL/SQL application developers:** Targeted to use PL/Scope through IDEs, such as JDeveloper and SQL Developer
- **IDE and Tools developers:** Provides the means to build a comprehensive cross-referenced repository by extracting, categorizing, organizing, and storing all identifiers discovered in the PL/SQL source code

Collecting PL/Scope Data

Collected data includes:

- Identifier types
- Usages
 - Declaration
 - Definition
 - Reference
 - Call
 - Assignment
- Location of usage



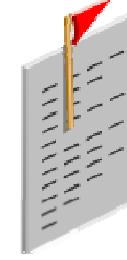
ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You can use PL/Scope to collect data about the user-defined identifiers found in the PL/SQL source code. The data is collected at compilation time and made available in the static data dictionary views.

Using PL/Scope

- Set the PL/SQL compilation parameter `PLSCOPE_SETTINGS`.
- Valid values for `IDENTIFIERS`:
 - `ALL` – Collect all PL/SQL identifier actions found in compiled source.
 - `NONE` – Do not collect any identifier actions (the default).
- Set at the session, system, or per library unit basis:
 - `ALTER SESSION SET PLSCOPE_SETTINGS = 'IDENTIFIERS: ALL'`
 - `ALTER SYSTEM SET PLSCOPE_SETTINGS = 'IDENTIFIERS: ALL'`
 - `ALTER functionname COMPILE PLSCOPE_SETTINGS = 'IDENTIFIERS: ALL'`
- The `USER/ALL/DBA_IDENTIFIER`s catalog view holds the collected identifier values.



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

To use PL/Scope, you need to set the PL/SQL compilation parameter `PLSCOPE_SETTINGS` to either '`IDENTIFIERS:ALL`' or '`IDENTIFIERS:NONE`'. You can use the `ALTER SESSION`, `ALTER SYSTEM`, or `ALTER COMPILE` statement to set this parameter. After the parameter is set, any code that you compile is analyzed for PL/Scope. The information collected is gathered in the `USER/ALL/DBA_IDENTIFIER`s dictionary view.

The identifier action describes how the identifier is used, such as in a declaration, definition, reference, assignment, or call.

Query the `USER|ALL|DBA_PLSQL_OBJECT_SETTINGS` views to view what `PLSCOPE_SETTINGS` are set to. This view contains a column called `PLSCOPE_SETTINGS`. When you query this column, by default, all objects are set to '`IDENTIFIERS:NONE`', unless you reset the PL/SQL compilation parameter and recompile the code.

USER/ALL/DBA_IDENTIFIERs Catalog View

Column	Description
OWNER	Owner of the identifier
NAME	Name of the identifier (may not be unique)
SIGNATURE	A unique signature for this identifier
TYPE	The type of the identifier, such as variable, formal, varray
OBJECT_NAME	The object name where the action occurred
OBJECT_TYPE	The type of object where the action occurred
USAGE	The action performed on the identifier, such as declaration, definition, reference, assignment, or call
USAGE_ID	The unique key for the identifier usage
LINE	The line where the identifier action occurred
COL	The column where the identifier action occurred
USAGE_CONTEXT_ID	The context USAGE_ID of the identifier action



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The USER/ALL/DBA_IDENTIFIERs dictionary view is used to collect information about your identifiers for any code that is compiled or altered when the PL/SQL compilation parameter is set to PLSCOPE_SETTINGS = 'IDENTIFIERS:ALL'.

Sample Data for PL/Scope

Sample data for scoping:

```
CREATE OR REPLACE PACKAGE BODY credit_card_pkg
IS
  PROCEDURE update_card_info
    (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no VARCHAR2)
  IS
    v_card_info typ_cr_card_nst;
    i INTEGER;
  BEGIN
    SELECT credit_cards
    ...
  END update_card_info;
  PROCEDURE display_card_info
    (p_cust_id NUMBER)
  IS
    v_card_info typ_cr_card_nst;
    i INTEGER;
  BEGIN
    SELECT credit_cards
    ...
  END display_card_info;
END credit_card_pkg; -- package body
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

CREDIT_CARD_PKG created earlier is used for PL/SQL scoping. The complete code is shown on the following page.

The code is a simple package that contains one type, two procedures, and one function. Identifier information will be collected on this package as shown on the following pages.

```
CREATE OR REPLACE PACKAGE credit_card_pkg
IS
    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no
VARCHAR2) ;

    PROCEDURE display_card_info
        (p_cust_id NUMBER);
END credit_card_pkg; -- package spec
/

CREATE OR REPLACE PACKAGE BODY credit_card_pkg
IS
    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no
VARCHAR2)
    IS
        v_card_info typ_cr_card_nst;
        i INTEGER;
    BEGIN
        SELECT credit_cards
            INTO v_card_info
            FROM customers
            WHERE customer_id = p_cust_id;
        IF v_card_info.EXISTS(1) THEN -- cards exist, add more
            i := v_card_info.LAST;
            v_card_info.EXTEND(1);
            v_card_info(i+1) := typ_cr_card(p_card_type, p_card_no);
            UPDATE customers
                SET credit_cards = v_card_info
                WHERE customer_id = p_cust_id;
        ELSE -- no cards for this customer yet, construct one
            UPDATE customers
                SET credit_cards = typ_cr_card_nst
                    (typ_cr_card(p_card_type, p_card_no))
                WHERE customer_id = p_cust_id;
        END IF;
    END update_card_info;
```

```
PROCEDURE display_card_info
    (p_cust_id NUMBER)
IS
    v_card_info typ_cr_card_nst;
    i INTEGER;
BEGIN
    SELECT credit_cards
        INTO v_card_info
        FROM customers
        WHERE customer_id = p_cust_id;
    IF v_card_info.EXISTS(1) THEN
        FOR idx IN v_card_info.FIRST..v_card_info.LAST LOOP
            DBMS_OUTPUT.PUT('Card Type: ' ||
v_card_info(idx).card_type
                           || ' ');
            DBMS_OUTPUT.PUT_LINE('/ Card No: ' ||
v_card_info(idx).card_num );
        END LOOP;
    ELSE
        DBMS_OUTPUT.PUT_LINE('Customer has no credit cards.');
    END IF;
END display_card_info;
END credit_card_pkg; -- package body
/
```

Collecting Identifiers

```
ALTER SESSION SET PLSCOPE_SETTINGS = 'IDENTIFIERS:ALL';  
ALTER PACKAGE credit_card_pkg COMPILE;
```

Identifier information is collected in the `USER_IDENTIFIER`s dictionary view, where you can:

- Perform a basic identifier search
- Use contexts to describe identifiers
- Find identifier actions
- Describe identifier actions



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In the example shown in the slide, the PL/SQL compilation parameter `PLSCOPE_SETTINGS` is enabled to collect information about all identifiers. `CREDIT_CARD_PKG` is then recompiled. After it is recompiled, the identifier information is available in the `ALL|USER|DBA_IDENTIFIER` views.

To verify that you have enabled identifier information to be collected on the `CREDIT_CARD_PKG` package, issue the following statement:

```
SELECT PLSCOPE_SETTINGS  
FROM USER_PLSQL_OBJECT_SETTINGS  
WHERE NAME='CREDIT_CARD_PKG' AND TYPE='PACKAGE BODY';  
  
PLSCOPE_SETTINGS  
-----  
IDENTIFIERS:ALL
```

Viewing Identifier Information

Create a hierarchical report of identifier information:

```
WITH v AS
  (SELECT    Line,
             Col,
             INITCAP(NAME)  Name,
             LOWER(TYPE)    Type,
             LOWER(USAGE)   Usage,
             USAGE_ID, USAGE_CONTEXT_ID
      FROM USER_IDENTIFIERS
     WHERE Object_Name = 'CREDIT_CARD_PKG'
       AND Object_Type = 'PACKAGE BODY' )
  SELECT RPAD(LPAD(' ', 2*(Level-1)) ||
              Name, 20, '.') || ' ' ||
              RPAD(Type, 20) || RPAD(Usage, 20)
              IDENTIFIER_USAGE_CONTEXTS
     FROM v
    START WITH USAGE_CONTEXT_ID = 0
   CONNECT BY PRIOR USAGE_ID = USAGE_CONTEXT_ID
  ORDER SIBLINGS BY Line, Col;
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The sample SQL statement shown in the slide retrieves and formats the identifier information that is collected for the CREDIT_CARD_PKG package body. Note that the inline view retrieves values from the Name, Type, and Usage columns in the USER_IDENTIFIERS dictionary view.

Viewing Identifier Information

Results:

```
IDENTIFIER_USAGE_CONTEXTS
-----
Credit_Card_Pkg..... package           definition
Cust_Card_Info.... function            definition
P_Cust_Id..... formal in             declaration
P_Card_Info..... formal in out       declaration
V_Card_Info_Exis variable            declaration
P_Card_Info..... formal in out       assignment
P_Cust_Id..... formal in             reference
Plitblm..... synonym                call
P_Card_Info... formal in out         reference
V_Card_Info_Ex variable              assignment
V_Card_Info_Exis variable            assignment
...
...
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The results are formatted and display the name of the identifier, type, and how the identifier is used.

Note: PLITBLM is an Oracle-supplied package that contains subprograms to help implement the basic language features. PLITBLM handles index-table operations.

Performing a Basic Identifier Search

Display the unique identifiers in your schema by querying for all 'DECLARATION' identifier actions:

```
SELECT NAME, SIGNATURE, TYPE
FROM USER_IDENTIFIERS
WHERE USAGE='DECLARATION'
ORDER BY OBJECT_TYPE, USAGE_ID;
```

#	NAME	SIGNATURE	TYPE
1	TAX_AMT	CB57A22DF0C8ED7946088BD2E91CD833	FUNCTION
2	GET_CREDIT	72EBEDAE9DD0C3A46F11DA088EDE076D	FUNCTION
3	ORD_COUNT	017E03E4C553FC8A091125490DBE5208	FUNCTION
4	GET_FILESIZE	B76901C0A7047545D96589337D670EC2	FUNCTION
5	X	A4E010E85FF197741D14C266748AF704	FORMAL IN
6	V_ID	26B4544E2F2FB67F677463BC4E8451C6	FORMAL IN
7	CUST_NO	63E4BF5F621EDC75E7A338D885156D86	FORMAL IN
8	P_FILE_PTR	25B954A3FA37084EE4A7C1196CC8FC6E	FORMAL IN OUT
9	V_CREDIT	9C6689C39B24408734C6CC28309F4CE7	VARIABLE
10	V_COUNT	BF3EC8187C9F044D6C8086125C28E4CA	VARIABLE

...



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You can display the unique identifiers in your schema by querying for all 'DECLARATION' identifier actions. Valid actions are:

- **DECLARATION:** All identifiers have one and only one declaration. Each declaration may also have an associated type.
- **TYPE:** Has the value of either packages, function or procedures, object types, triggers, or exceptions
- **SIGNATURE:** A unique value that distinguishes the identifier from other identifiers with the same name, whether they are defined in the same program unit or different program units

Using USER_IDENTIFIERs to Find All Local Variables

Find all local variables:

```
SELECT a.NAME variable_name, b.NAME context_name, a.SIGNATURE
FROM USER_IDENTIFIERs a, USER_IDENTIFIERs b
WHERE a.USAGE_CONTEXT_ID = b.USAGE_ID
  AND a.TYPE = 'VARIABLE'
  AND a.USAGE = 'DECLARATION'
  AND a.OBJECT_NAME = 'CREDIT_CARD_PKG'
  AND a.OBJECT_NAME = b.OBJECT_NAME
  AND a.OBJECT_TYPE = b.OBJECT_TYPE
  AND (b.TYPE = 'FUNCTION' or b.TYPE = 'PROCEDURE')
ORDER BY a.OBJECT_TYPE, a.USAGE_ID;
```

VARIABLE_NAME	CONTEXT_NAME	SIGNATURE
V_CARD_INFO	UPDATE_CARD_INFO	5FE3409B23709E61A12314F2667949CA
I	UPDATE_CARD_INFO	F500BE5A79542F08A3ABFF4AFC986C1E
V_CARD_INFO	DISPLAY_CARD_INFO	5ACFED9813606BB5A8BCBC9063F974E4
I	DISPLAY_CARD_INFO	E5E48B87FD19043F4240884ECA77E916



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In the example shown in the slide, all local variables belonging to procedures or functions are found for the CREDIT_CARD_PKG package.

Finding Identifier Actions

Find all usages performed on the local variable:

```
SELECT USAGE, USAGE_ID, OBJECT_NAME, OBJECT_TYPE
FROM USER_IDENTIFIERS
WHERE SIGNATURE='5FE3409B23709E61A12314F2667949CA'
ORDER BY OBJECT_TYPE, USAGE_ID;
```

USAGE	USAGE_ID	OBJECT_NAME	OBJECT_TYPE
DECLARATION	9	CREDIT_CARD_PKG	PACKAGE BODY
ASSIGNMENT	14	CREDIT_CARD_PKG	PACKAGE BODY
REFERENCE	16	CREDIT_CARD_PKG	PACKAGE BODY
REFERENCE	19	CREDIT_CARD_PKG	PACKAGE BODY
REFERENCE	21	CREDIT_CARD_PKG	PACKAGE BODY
ASSIGNMENT	22	CREDIT_CARD_PKG	PACKAGE BODY
REFERENCE	28	CREDIT_CARD_PKG	PACKAGE BODY



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You can find all actions performed on an identifier. In the example in the slide, all actions performed on a variable are found by searching on the signature of the variable. The variable is called `V_CARD_INFO`, and it is used in the `DISPLAY_CARD_INFO` procedure. Variable `V_CARD_INFO`'s signature is found in the previous query. It is available to you in the `SIGNATURE` column of the `USER_IDENTIFIERS` dictionary view.

The different types of identifier usage:

- DECLARATION
- DEFINITION
- CALL
- REFERENCE
- ASSIGNMENT

Finding Identifier Actions

Find out where the assignment to the local identifier `v_card_info` occurred:

```
SELECT LINE, COL, OBJECT_NAME, OBJECT_TYPE
FROM USER_IDENTIFIERS
WHERE SIGNATURE='5ACFED9813606BB5A8BCBC9063F974E4'
AND USAGE='ASSIGNMENT';
```

LINE	COL	OBJECT_NAME	OBJECT_TYPE
35	12	CREDIT_CARD_PKG	PACKAGE BODY



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The local `v_card_info` identifier is found on line 35 of the `CREDIT_CARD_PKG` body.

Lesson Agenda

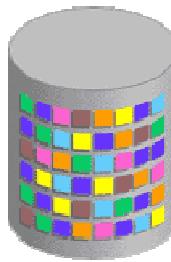
- Running reports on source code
- Determining identifier types and usages
- **Using DBMS_METADATA to retrieve object definitions**
- PL/SQL Enhancements



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

DBMS_METADATA Package

The DBMS_METADATA package provides a centralized facility for the extraction, manipulation, and resubmission of dictionary metadata.



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You can invoke DBMS_METADATA to retrieve metadata from the database dictionary as XML or creation DDL, and submit the XML to re-create the object.

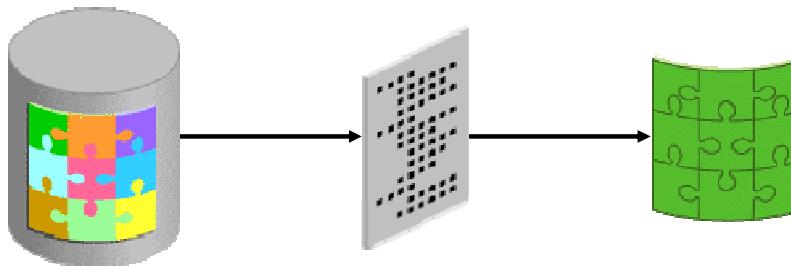
You can use DBMS_METADATA for extracting metadata from the dictionary, manipulating the metadata (adding columns, changing column data types, and so on), and then converting the metadata to data definition language (DDL) so that the object can be re-created on the same or another database. Earlier, you had to do this programmatically, which resulted in problems in each new release.

The DBMS_METADATA functionality is used for the Oracle 12c Export/Import replacement, commonly called “the Data Pump.”

Metadata API

Processing involves the following steps:

1. Fetch an object's metadata as XML.
2. Transform the XML in a variety of ways (including transforming it into SQL DDL).
3. Submit the XML to re-create the object.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Every entity in the database is modeled as an object that belongs to an object type. For example, the ORDERS table is an object; its object type is TABLE. When you fetch an object's metadata, you must specify the object type.

Every object type is implemented by using three entities:

- A user-defined type (UDT) whose attributes comprise all metadata for objects of the type. An object's XML representation is a translation of a type instance into XML with the XML tag names derived from the type attribute names. (In the case of tables, several UDTs are needed to represent the different varieties of the object type.)
- An object view of the UDT that populates the instances of the object type
- An Extensible Style Sheet Language (XSL) script that converts the XML representation of an object into SQL DDL

Subprograms in DBMS_METADATA

Name	Description
OPEN	Specifies the type of object to be retrieved, the version of its metadata, and the object model. The return value is an opaque context handle for the set of objects.
SET_FILTER	Specifies restrictions on the objects to be retrieved, such as the object name or schema
SET_COUNT	Specifies the maximum number of objects to be retrieved in a single <code>FETCH_xxx</code> call
GET_QUERY	Returns the text of the queries that will be used by <code>FETCH_xxx</code>
SET_PARSE_ITEM	Enables output parsing and specifies an object attribute to be parsed and returned
ADD_TRANSFORM	Specifies a transform that <code>FETCH_xxx</code> applies to the XML representation of the retrieved objects
SET_TRANSFORM_PARAM, SET_REMAP_PARAM	Specifies parameters to the XSLT stylesheet identified by <code>transform_handle</code>
FETCH_xxx	Returns metadata for objects that meet the criteria established by <code>OPEN</code> , <code>SET_FILTER</code>
CLOSE	Invalidates the handle returned by <code>OPEN</code> and cleans up the associated state



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The table in the slide provides an overview of the procedures and functions that are available in the `DBMS_METADATA` package. To retrieve metadata, you can specify the:

- Kind of object to be retrieved—either an object type (a table, an index, or a procedure) or a heterogeneous collection of object types that form a logical unit (such as database export and schema export)
- Selection criteria (owner, name, and so on)
- “parse items” attributes of objects to be parsed and returned separately
- Transformations on the output, implemented by XSLT scripts

The package provides two types of retrieval interfaces for two types of usage:

- **For programmatic use:** `OPEN`, `SET_FILTER`, `SET_COUNT`, `GET_QUERY`, `SET_PARSE_ITEM`, `ADD_TRANSFORM`, `SET_TRANSFORM_PARAM`, `SET_REMAP_PARAM`, `FETCH_xxx`, and `CLOSE`. These enable a flexible selection criteria and the extraction of a stream of objects.
- **For use in SQL queries and for ad hoc browsing:** The `GET_xxx` interfaces (`GET_XML` and `GET_DDL`) return metadata for a single named object. The `GET_DEPENDENT_xxx` and `GET_GRANTED_xxx` interfaces return metadata for one or more dependent or granted objects. None of these APIs supports heterogeneous object types.

FETCH_xxx Subprograms

Name	Description
FETCH_XML	This function returns the XML metadata for an object as an XMLType.
FETCH_DDL	This function returns the DDL (either to create or to drop the object) into a predefined nested table.
FETCH_CLOB	This function returns the objects (transformed or not) as a CLOB.
FETCH_XML_CLOB	This procedure returns the XML metadata for the objects as a CLOB in an IN OUT NOCOPY parameter to avoid expensive LOB copies.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The functions and procedure listed in the slide return metadata for objects meeting the criteria established by the call to the OPEN function that returned the handle, and the subsequent calls to SET_FILTER, SET_COUNT, ADD_TRANSFORM, and so on. Each call to `FETCH_xxx` returns the number of objects specified by `SET_COUNT` (or a smaller number, if fewer objects remain in the current cursor) until all objects are returned.

SET_FILTER Procedure

- Syntax:

```
PROCEDURE set_filter
( handle IN NUMBER,
  name   IN VARCHAR2,
  value  IN VARCHAR2 | BOOLEAN | NUMBER,
  object_type_path VARCHAR2
);
```

- Example:

```
...
DBMS_METADATA.SET_FILTER (handle, 'NAME', 'OE');
...
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You use the SET_FILTER procedure to identify restrictions on the objects that are to be retrieved. For example, you can specify restrictions on an object or schema that is being retrieved. This procedure is overloaded with parameters that have the following meanings:

- handle is the handle returned from the OPEN function.
- name is the name of the filter. For each filter, the object type applies to its name, data type (text or Boolean), and meaning or effect (including its default value, if there is one).
- value is the value of the filter. It can be text, Boolean, or a numeric value.
- object_type_path is a path name designating the object types to which the filter applies. By default, the filter applies to the object type of the OPEN handle.

If you use an expression filter, it is placed to the right of a SQL comparison, and the value is compared with it. The value must contain parentheses and quotation marks where appropriate. A filter value is combined with a particular object attribute to produce a WHERE condition in the query that fetches the objects.

Filters

There are over 70 filters that are organized into object type categories such as:

- Named objects
- Tables
- Objects dependent on tables
- Index
- Dependent objects
- Granted objects
- Table data
- Index statistics
- Constraints
- All object types
- Database export



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

There are over 70 filters that you can specify when using the `SET_FILTER` procedure. These filters are organized into object type categories. Some of the object type categories are listed in the slide.

When using the `SET_FILTER` procedure, you specify the name of the filter and its respective value.

For example, you can use the `SCHEMA` filter with a value to identify the schema whose objects are selected. Then use a second call to the `SET_FILTER` procedure and use a filter named `INCLUDE_USER` that has a Boolean data type for its value. If it is set to `TRUE`, objects that contain privileged information about the user are retrieved.

```
DBMS_METADATA.SET_FILTER(handle, SCHEMA, 'OE') ;
DBMS_METADATA. SET_FILTER(handle, INCLUDE_USER, TRUE) ;
```

Each call to `SET_FILTER` causes a `WHERE` condition to be added to the underlying query that fetches the set of objects. The `WHERE` conditions are combined by using an `AND` operator, so that you can use multiple `SET_FILTER` calls to refine the set of objects to be returned.

Examples of Setting Filters

Set the filter to fetch the OE schema objects excluding the object types of functions, procedures, and packages, as well as any views that contain PAYROLL at the start of the view name:

```
DBMS_METADATA.SET_FILTER(handle, 'SCHEMA_EXPR',
    'IN (''PAYROLL'', ''OE''))';
DBMS_METADATA.SET_FILTER(handle, 'EXCLUDE_PATH_EXPR',
    '='''FUNCTION'''');
DBMS_METADATA.SET_FILTER(handle, 'EXCLUDE_PATH_EXPR',
    '='''PROCEDURE'''');
DBMS_METADATA.SET_FILTER(handle, 'EXCLUDE_PATH_EXPR',
    '='''PACKAGE'''');
DBMS_METADATA.SET_FILTER(handle, 'EXCLUDE_NAME_EXPR',
    'LIKE ''PAYROLL%'''', 'VIEW');
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The example shown in the slide calls the SET_FILTER procedure several times to create a WHERE condition that identifies which object types are to be fetched. First, the objects in the PAYROLL and OE schemas are identified as object types to be fetched. Subsequently, the SET_FILTER procedure identifies certain object types (functions, procedures, and packages) and view object names that are to be excluded.

Programmatic Use: Example 1

```

CREATE PROCEDURE example_one IS
    v_hdl      NUMBER; v_th1  NUMBER; v_th2  NUMBER;
    v_doc      sys.ku$ddls; ← 1
BEGIN
    v_hdl := DBMS_METADATA.OPEN('SCHEMA_EXPORT'); ← 2
    DBMS_METADATA.SET_FILTER(v_hdl, 'SCHEMA', 'OE'); ← 3
    v_th1 := DBMS_METADATA.ADD_TRANSFORM(v_hdl, ← 4
        'MODIFY', NULL, 'TABLE');
    DBMS_METADATA.SET_REMAP_PARAM(v_th1, ← 5
        'REMAP_TABLESPACE', 'SYSTEM', 'TBS1');
    v_th2 := DBMS_METADATA.ADD_TRANSFORM(v_hdl, 'DDL'); ← 6
    DBMS_METADATA.SET_TRANSFORM_PARAM(v_th2, ←
        'SQLTERMINATOR', TRUE);
    DBMS_METADATA.SET_TRANSFORM_PARAM(v_th2, ←
        'REF_CONSTRAINTS', FALSE, 'TABLE');
    LOOP
        v_doc := DBMS_METADATA.FETCH_DDL(v_hdl); ← 7
        EXIT WHEN v_doc IS NULL;
    END LOOP;
    DBMS_METADATA CLOSE(v_hdl); ← 8
END;

```

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In this example, all objects are retrieved from the HR schema as creation DDL. The MODIFY transform is used to change the tablespaces for the tables.

1. The DBMS_METADATA package has several predefined types that are owned by SYS. The sys.ku\$ddls stand-alone object type is used in the DBMS_METADATA package. It is a table type that holds the CLOB type of data.
2. You use the OPEN function to specify the type of object to be retrieved, the version of its metadata, and the object model. It returns a context handle for the set of objects. In this example, 'SCHEMA_EXPORT' is the object type, which indicates all metadata objects in a schema. There are 85 predefined types of objects for the model that you can specify for this parameter. Both the version of metadata and the object model parameters are not identified in this example. The version of the metadata parameter defaults to 'COMPATIBLE'. You can also specify 'LATEST' or a specific database version.
3. The SET_FILTER procedure identifies restrictions on the objects that are to be retrieved.

4. The ADD_TRANSFORM function specifies a transform that FETCH_XXX applies to the XML representation of the retrieved objects. You can have multiple transforms. In the example, two transforms occur, one for each of the th1 and th2 program variables. The ADD_TRANSFORM function accepts four parameters and returns a number representing the opaque handle to the transform. The parameters are the handle returned from the OPEN statement, the name of the transform (DDL, DROP, or MODIFY), the encoding name (which is the name of the national language support [NLS] character set in which the style sheet pointed to by the name is encoded), and the object type. If the object type is omitted, the transform applies to all objects; otherwise, it applies only to the object type specified. The first transform shown in the program code is the handle returned from the OPEN function. The second transform shown in the code has two parameter values specified. The first parameter is the handle identified from the OPEN function. The second parameter value is the DDL, which means the document is transformed to the DDL that creates the object. The output of this transform is not an XML document. The third and fourth parameters are not specified. Both take the default values for the encoding and object type parameters.
5. The SET_REMAP_PARAM procedure identifies the parameters to the XSLT style sheet identified by the transform handle, which is the first parameter passed to the procedure. In the example, the second parameter value 'REMAP_TABLESPACE' means that the objects have their tablespaces renamed from an old value to a new value. In the ADD_TRANSFORM function, the choices are DDL, DROP, or MODIFY. For each of these values, the SET_REMAP_PARAM identifies the name of the parameter. REMAP_TABLESPACE means the objects in the document will have their tablespaces renamed from an old value to a new value. The third and fourth parameters identify the old value and the new value. In this example, the old tablespace name is SYSTEM, and the new tablespace name is TBS1.
6. SET_TRANSFORM_PARAM works similarly to SET_REMAP_PARAM. In the code shown, the first call to SET_TRANSFORM_PARAM identifies the parameters for the th2 variable. The SQLTERMINATOR and TRUE parameter values cause the SQL terminator (; or /) to be appended to each DDL statement. The second call to SET_TRANSFORM_PARAM identifies more characteristics for the th2 variable. REF_CONSTRAINTS, FALSE, and TABLE means that the referential constraints on the tables are not copied to the document.
7. The FETCH_DDL function returns the metadata for objects that meet the criteria established by the OPEN, SET_FILTER, ADD_TRANSFORM, SET_REMAP_PARAM, and SET_TRANSFORM_PARAM subroutines.
8. The CLOSE function invalidates the handle returned by the OPEN function and cleans up the associated state. Use this function to terminate the stream of objects established by the OPEN function.

Programmatic Use: Example 2

```
CREATE FUNCTION get_table_md RETURN CLOB IS
  v_hdl  NUMBER; -- returned by 'OPEN'
  v_th   NUMBER; -- returned by 'ADD_TRANSFORM'
  v_doc  CLOB;
BEGIN
  -- specify the OBJECT TYPE
  v_hdl := DBMS_METADATA.OPEN('TABLE');
  -- use FILTERS to specify the objects desired
  DBMS_METADATA.SET_FILTER(v_hdl , 'SCHEMA','OE');
  DBMS_METADATA.SET_FILTER
    (v_hdl , 'NAME','ORDERS');
  -- request to be TRANSFORMED into creation DDL
  v_th := DBMS_METADATA.ADD_TRANSFORM(v_hdl,'DDL');
  -- FETCH the object
  v_doc := DBMS_METADATA.FETCH_CLOB(v_hdl);
  -- release resources
  DBMS_METADATA CLOSE(v_hdl);
  RETURN v_doc;
END;
/
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The example in this slide returns the metadata for the ORDERS table. The result is:

```
set pagesize 0
set long 1000000
SELECT get_table_md FROM dual;

CREATE TABLE "OE"."ORDERS"
  (  "ORDER_ID" NUMBER(12,0),
     "ORDER_DATE" TIMESTAMP (6) WITH LOCAL TIME ZONE
  CONSTRAINT "ORDER_DATE_NN" NOT
  NULL ENABLE,
     "ORDER_MODE" VARCHAR2(8),
     "CUSTOMER_ID" NUMBER(6,0) CONSTRAINT
  "ORDER_CUSTOMER_ID_NN" NOT NULL ENABLE,
     "ORDER_STATUS" NUMBER(2,0),
     "ORDER_TOTAL" NUMBER(8,2),
     "SALES_REP_ID" NUMBER(6,0),
     "PROMOTION_ID" NUMBER(6,0),
     CONSTRAINT "ORDER_MODE_LOV" CHECK (order_mode in
  ...

```

```
...
CONSTRAINT "ORDER_TOTAL_MIN" CHECK (order_total >= 0) ENABLE,
CONSTRAINT "ORDER_PK" PRIMARY KEY ("ORDER_ID")
    USING INDEX PCTFREE 10 INITTRANS 2 MAXTRANS 255 NOLOGGING
    COMPUTE STATISTICS
STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS
       2147483645
PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL
DEFAULT)
TABLESPACE "EXAMPLE"   ENABLE,
CONSTRAINT "ORDERS_CUSTOMER_ID_FK" FOREIGN KEY ("CUSTOMER_ID")
    REFERENCES "OE"."CUSTOMERS" ("CUSTOMER_ID")
    ON DELETE SET NULL ENABLE,
CONSTRAINT "ORDERS_SALES_REP_FK" FOREIGN KEY
    ("SALES REP_ID") REFERENCES "HR"."EMPLOYEES"
    ("EMPLOYEE_ID") ON DELETE SET NULL ENABLE
)
PCTFREE 10 PCTUSED 40 INITTRANS 1 MAXTRANS 255
    NOCOMPRESS NOLOGGING
STORAGE(INITIAL 65536 NEXT 1048576
       MINEXTENTS 1 MAXEXTENTS 2147483645
PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1
    BUFFER_POOL DEFAULT)
TABLESPACE "EXAMPLE"

You can accomplish the same effect with the browsing interface:
SELECT dbms_metadata.get_ddl
    ('TABLE', 'ORDERS', 'OE')
FROM dual;
```

Browsing APIs

Name	Description
GET_XXX	The GET_XML and GET_DDL functions return metadata for a single named object.
GET_DEPENDENT_XXX	This function returns metadata for a dependent object.
GET_GRANTED_XXX	This function returns metadata for a granted object.
Where xxx is:	DDL or XML



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The browsing APIs are designed for use in SQL queries and ad hoc browsing. These functions enable you to fetch metadata for objects with a single call. They encapsulate calls to OPEN, SET_FILTER, and so on. The function that you use depends on the characteristics of the object type and whether you want XML or DDL.

For some object types, you can use multiple functions. You can use GET_XXX to fetch an index by name, or GET_DEPENDENT_XXX to fetch the same index by specifying the table on which it is defined.

GET_XXX returns a single object name.

For GET_DEPENDENT_XXX and GET_GRANTED_XXX, an arbitrary number of granted or dependent objects may match the input criteria. However, you can specify an object count when fetching these objects.

If you invoke these functions from SQL*Plus, you should use the SET LONG and SET PAGESIZE commands to retrieve the complete, uninterrupted output.

```
SET LONG 2000000
SET PAGESIZE 300
```

Browsing APIs: Examples

- Get the XML representation of OE.ORDERS:

```
SELECT DBMS_METADATA.GET_XML
      ('TABLE', 'ORDERS', 'OE')
FROM   dual;
```

- Fetch the DDL for all object grants on OE.ORDERS:

```
SELECT DBMS_METADATA.GET_DEPENDENT_DDL
      ('OBJECT_GRANT', 'ORDERS', 'OE')
FROM   dual;
```

- Fetch the DDL for all system grants granted to OE:

```
SELECT DBMS_METADATA.GET_GRANTED_DDL
      ('SYSTEM_GRANT', 'OE')
FROM   dual;
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

- Results for fetching the XML representation of OE.ORDERS:

```
DBMS_METADATA.GET_XML('TABLE', 'ORDERS', 'OE')
-----
<?xml version="1.0"?>
<ROWSET>
  <ROW>
    <TABLE_T>
      <VERS_MAJOR>1</VERS_MAJOR>
```

- Results for fetching the DDL for all object grants on OE.ORDERS:

```
DBMS_METADATA.GET_DEPENDENT_DDL
  ('OBJECT_GRANT', 'ORDERS', 'OE')
-----
GRANT SELECT ON "OE"."ORDERS" TO "PM"
GRANT SELECT ON "OE"."ORDERS" TO "BI"
```

- Results for fetching the DDL for all system grants granted to OE:

```
DBMS_METADATA.GET_GRANTED_DDL('SYSTEM_GRANT', 'OE')
-----
GRANT QUERY REWRITE TO "OE"
...

```

Browsing APIs: Examples

```

BEGIN
  DBMS_METADATA.SET_TRANSFORM_PARAM(
    DBMS_METADATA.SESSION_TRANSFORM,
    'STORAGE', false);
END;
/
SELECT DBMS_METADATA.GET_DDL('TABLE', u.table_name)
FROM user_all_tables u
WHERE u.nested = 'NO'
AND (u.iot_type IS NULL OR u.iot_type = 'IOT');

BEGIN
  DBMS_METADATA.SET_TRANSFORM_PARAM(
    DBMS_METADATA.SESSION_TRANSFORM, 'DEFAULT');
END;
/

```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The example in the slide shows how to fetch creation DDL for all “complete” tables in the current schema, filtering out nested tables and overflow segments. The steps shown in the slide are as follows:

1. The `SET_TRANSFORM_PARAM` function specifies that the storage clauses are not to be returned in the SQL DDL. The `SESSION_TRANSFORM` function is interpreted to mean “for the current session.”
2. Use the `GET_DDL` function to retrieve DDL on all nonnested and non-IOT (index-organized table) tables.

```

CREATE TABLE "HR"."COUNTRIES"
( "COUNTRY_ID" CHAR(2)
  CONSTRAINT "COUNTRY_ID_NN" NOT NULL ENABLE,
  "COUNTRY_NAME" VARCHAR2(40),
  "REGION_ID" NUMBER,
  CONSTRAINT "COUNTRY_C_ID_PK"
  PRIMARY KEY ("COUNTRY_ID") ENABLE,
  CONSTRAINT "COUNTR_REG_FK" FOREIGN KEY
  ...

```

3. Reset the session-level parameters to their defaults.

Lesson Agenda

- Running reports on source code
- Determining identifier types and usages
- Using DBMS_METADATA to retrieve object definitions
- PL/SQL Enhancements



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Using the DBMS_UTILITY.EXPAND_SQL_TEXT Procedure

This procedure recursively replaces any view references in the input SQL query with the corresponding view subquery.

Syntax:

```
DBMS_UTILITY.EXPAND_SQL_TEXT (
    input_sql_text      IN          CLOB,
    output_sql_text     OUT NOCOPY  CLOB);
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

- The `input_sql_text` parameter is the input SQL query text and the `output_sql_text` parameter is the View-expanded query text.
- The expanded and merged SQL statement text is copied to `output_sql_text` on successful completion.
- If there are invoker rights functions called from any of the views, they may be called as a different user in the resulting query text if the view owner is different from the user who will eventually compile or run the expanded SQL text.
- If there are references to remote objects, results are undetermined.

EXPAND_SQL_TEXT Exceptions

Exception	Description
ORA-00942	The current user does not have select privileges on all the views and tables recursively referenced in the <code>input_sql_text</code> .
ORA-24251	<code>input_sql_text</code> is not a SELECT statement.
ORA-00900	Input is not valid.
ORA-29477	The input LOB size exceeds the maximum size of 4GB -1.
ORA-00942	The current user does not have select privileges on all the views and tables recursively referenced in the <code>input_sql_text</code> .



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

This slide lists the exceptions pertaining to the EXPAND_SQL_TEXT procedure.

Using the UTL_CALL_STACK Package

The UTL_CALL_STACK package is introduced newly in Oracle Database 12c. This package:

- Is similar to DBMS_UTILITY.FORMAT_CALL_STACK
- Defines a VARRAY type UNIT_QUALIFIED_NAME
- Provides subprograms to return the current call stack for a PL/SQL program
- Displays the call stack in a structured format
- Is well suited for programmatic analysis



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The UTL_CALL_STACK package provides an interface for PL/SQL programmers to get information about currently executing programs including the subprogram name from dynamic and lexical stacks and the depths of those stacks.

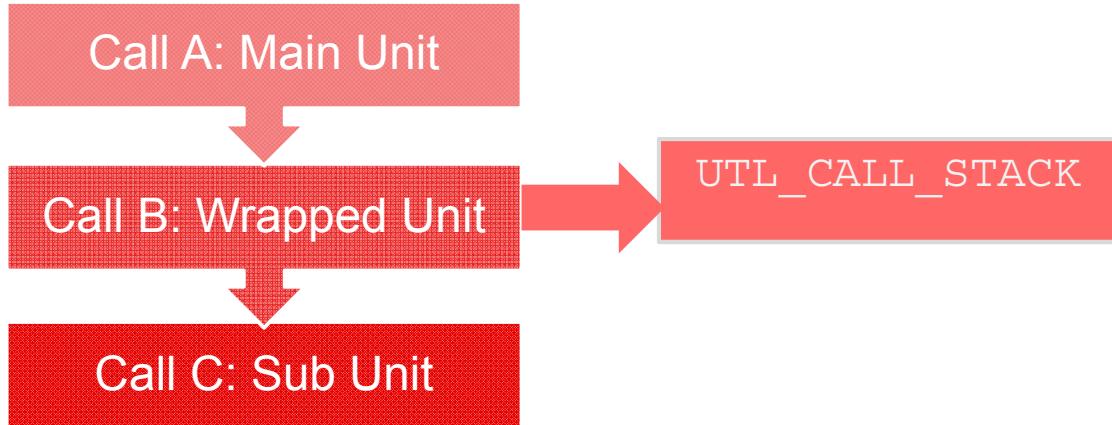
Individual functions return subprogram names, unit names, owner names, edition names, and line numbers for given dynamic depths.

More functions return error stack information.

Using these information, you can create more revealing error logs and application execution traces.

To know the complete set of UTL_CALL_STACK subprograms, refer to ‘Oracle Database PL/SQL Packages and Types Reference 12c Release 1’ documentation.

UTL_CALL_STACK: Security Model



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Security Model:

- The EXECUTE privilege on UTL_CALL_STACK should be granted to PUBLIC to use the package.
- The UTL_CALL_STACK package does not show wrapped program units.
- For example, consider a call stack in which program unit A calls B, which calls C, and in turn calls UTL_CALL_STACK to determine the subprogram list. If program unit B is wrapped, then the subprogram list shows only program unit C.

Exception:

- The exception 64610: BAD_DEPTH_INDICATOR is raised when a provided depth is out of bounds. Dynamic and lexical depths are positive integer values. Error and backtrace depths are non-negative integer values and are zero only in the absence of an exception.

New Predefined Inquiry Directives in Oracle Database 12c

Inquiry Directive	Description
<code>\$\$PLSQL_UNIT_OWNER</code>	A VARCHAR2 literal that contains the name of the owner of the current PL/SQL unit. If the current PL/SQL unit is an anonymous block, then <code>\$\$PLSQL_UNIT_OWNER</code> contains a NULL value.
<code>\$\$PL/SQL_UNIT_TYPE</code>	A VARCHAR2 literal that contains the type of the current PL/SQL unit



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

An **inquiry directive** provides information about the compilation environment.

The `$$PLSQL_OWNER` and `$$PLSQL_TYPE` predefined PL/SQL inquiry directives are now supported in Oracle Database 12c.

Syntax:

`$$name`

Usage:

```
DBMS_OUTPUT.PUT_LINE('$$PLSQL_UNIT_OWNER = ' || $$PLSQL_UNIT_OWNER);
DBMS_OUTPUT.PUT_LINE('$$PLSQL_UNIT_TYPE = ' || $$PLSQL_UNIT_TYPE);
```

To understand the complete set of predefined inquiry directives, refer to the *Oracle Database PL/SQL Language Reference 12c Release 1 documentation*.

Quiz

Which of the following SQL Developer predefined reports would you use to find the occurrence of a text string or an object name within a PL/SQL coding?

- a. Search Source Code
- b. Program Unit Arguments
- c. Unit Line Counts



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Answer: a

Quiz

Which of the following would you use to find information about arguments for procedures and functions?

- a. The ALL_ARGUMENTS view
- b. The DBMS_DESCRIBE.DESCRIBE_PROCEDURE routine
- c. SQL Developer predefined report, Program Unit Arguments
- d. None of the above



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Answer: b, c

Quiz

You can invoke DBMS_METADATA to retrieve metadata from the database dictionary as XML or creation DDL, and submit the XML to re-create the object.

- a. True
- b. False



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Answer: a

Summary

In this lesson, you should have learned how to:

- Use the supplied packages and the dictionary views to find coding information
- Determine identifier types and usages with PL/Scope
- Use the DBMS_METADATA package to obtain metadata from the data dictionary as XML or creation DDL that can be used to re-create the objects



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

This lesson showed you how to use the dictionary views and supplied PL/SQL packages to analyze your PL/SQL applications.

Practice 10: Overview

This practice covers the following topics:

- Analyzing PL/SQL code
- Using PL/Scope
- Using DBMS_METADATA



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Using the OE application that you have created, write code to analyze your application. You will perform the following:

- Find coding information
- Use PL/Scope
- Use DBMS_METADATA

THESE eKIT MATERIALS ARE FOR YOUR USE IN THIS CLASSROOM ONLY. COPYING eKIT MATERIALS FROM THIS COMPUTER IS STRICTLY PROHIBITED

Oracle University and Error : You are not a Valid Partner use only

11

Profiling and Tracing PL/SQL Code

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Objectives

After completing this lesson, you should be able to do the following:

- Trace PL/SQL program execution
- Profile PL/SQL applications



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In this lesson, you learn how to write PL/SQL routines that analyze the PL/SQL applications. You are also introduced to tracing and profiling PL/SQL code.

Lesson Agenda

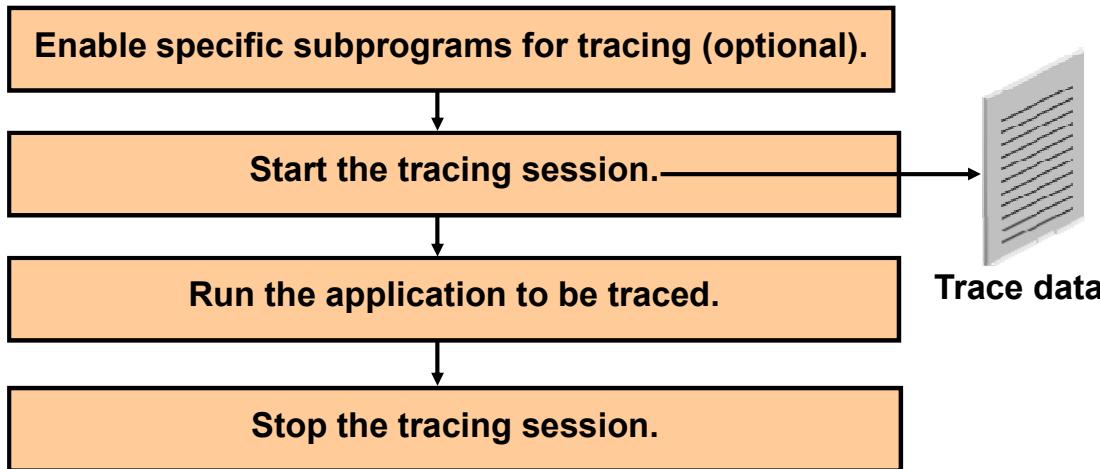
- Tracing PL/SQL program execution
- Profiling PL/SQL applications



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Tracing PL/SQL Execution

Tracing PL/SQL execution provides you with a better understanding of the program execution path, and is possible by using the `dbms_trace` package.



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In large and complex PL/SQL applications, it can sometimes become difficult to keep track of subprogram calls when a number of subprograms call each other. By tracing your PL/SQL code, you can get a clearer idea of the paths and order in which your programs execute.

Although a facility to trace your SQL code has been around for a while, Oracle now provides an API for tracing the execution of PL/SQL programs on the server. You can use the Trace API, implemented on the server as the `dbms_trace` package, to trace PL/SQL subprogram code.

Note: You cannot use PL/SQL tracing with the multithreaded server (MTS).

Tracing PL/SQL Execution

The `dbms_trace` package contains:

- `set_plsql_trace (trace_level INTEGER)`
- `clear_plsql_trace`
- `plsql_trace_version`

Function	Description
<code>set_plsql_trace</code>	Starts to trace data dumping in a session (You provide the trace level at which you want your PL/SQL code traced as an <code>IN</code> parameter.)
<code>clear_plsql_trace</code>	Starts to trace data dumping in a session
<code>plsql_trace_version</code>	Returns the version number of the trace package as an <code>OUT</code> parameter



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The `dbms_trace` Programs

`dbms_trace` provides subprograms to start and stop PL/SQL tracing in a session. The trace data is collected as the program executes, and it is written out to data dictionary tables.

A typical trace session involves:

- Enabling specific subprograms for trace data collection (optional)
- Starting the PL/SQL tracing session (`dbms_trace.set_plsql_trace`)
- Running the application that is to be traced
- Stopping the PL/SQL tracing session (`dbms_trace.clear_plsql_trace`)

Tracing PL/SQL Execution

- Using `set_plsql_trace`, select a trace level to identify how to trace calls, exceptions, SQL, and lines of code.
- Trace-level constants:
 - `trace_all_calls`
 - `trace_enabled_calls`
 - `trace_all_sql`
 - `trace_enabled_sql`
 - `trace_all_exceptions`
 - `trace_enabled_exceptions`
 - `trace_enabled_lines`
 - `trace_all_lines`
 - `trace_stop`
 - `trace_pause`
 - `trace_resume`



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Specifying a Trace Level

During the trace session, there are two levels that you can specify to trace calls, exceptions, SQL, and lines of code.

Trace Calls

- **Level 1:** Trace all calls. This corresponds to the constant `trace_all_calls`.
- **Level 2:** Trace calls only to enabled program units. This corresponds to the constant `trace_enabled_calls`.

Trace Exceptions

- **Level 1:** Trace all exceptions. This corresponds to `trace_all_exceptions`.
- **Level 2:** Trace exceptions raised only in enabled program units. This corresponds to `trace_enabled_exceptions`.

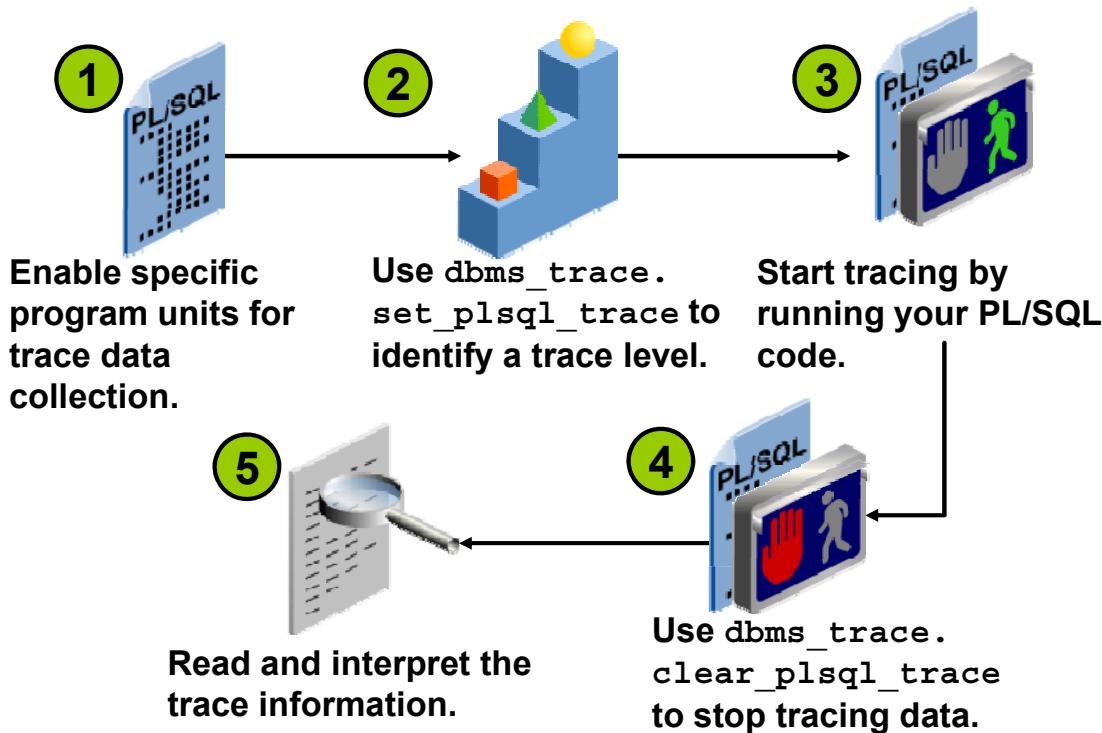
Trace SQL

- **Level 1:** Trace all SQL. This corresponds to the constant `trace_all_sql`.
- **Level 2:** Trace SQL only in enabled program units. This corresponds to the constant `trace_enabled_sql`.

Trace Lines

- **Level 1:** Trace all lines. This corresponds to the constant `trace_all_lines`.
- **Level 2:** Trace lines only in enabled program units. This corresponds to the constant `trace_enabled_lines`.

Tracing PL/SQL: Steps



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

To trace PL/SQL code by using the `dbms_trace` package, perform the following steps:

1. Enable specific program units for trace data collection.
2. Use `dbms_trace.set_plsql_trace` to identify a trace level.
3. Run your PL/SQL code.
4. Use `dbms_trace.clear_plsql_trace` to stop tracing data.
5. Read and interpret the trace information.

The following few slides demonstrate the steps to accomplish PL/SQL tracing.

Step 1: Enable Specific Subprograms

Enable specific subprograms with one of the two methods:

- Enable a subprogram by compiling it with the debug option:

```
ALTER SESSION SET PLSQL_DEBUG=true;
```

```
CREATE OR REPLACE ....
```

- Recompile a specific subprogram with the debug option:

```
ALTER [PROCEDURE | FUNCTION | PACKAGE]
      <subprogram-name> COMPILE DEBUG [BODY] ;
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Profiling large applications may produce a huge volume of data that can be difficult to manage. Before turning on the trace facility, you have the option to control the volume of data collected by enabling a specific subprogram for trace data collection. You can enable a subprogram by compiling it with the debug option. You can do this in one of two ways:

- Enable a subprogram by compiling it with the ALTER SESSION debug option, and then compile the program unit by using the CREATE OR REPLACE syntax:

```
ALTER SESSION SET PLSQL_DEBUG = true;
CREATE OR REPLACE ...
```

- Recompile a specific subprogram with the debug option:

```
ALTER [PROCEDURE | FUNCTION | PACKAGE]
      <subprogram-name> COMPILE DEBUG [BODY] ;
ALTER PROCEDURE P5 COMPILE DEBUG;
```

Note: The second method cannot be used for anonymous blocks.

Enabling specific subprograms enables you to:

- Limit and control the amount of trace data, especially in large applications
- Obtain additional trace information that is otherwise not available. For example, during the tracing session, if a subprogram calls another subprogram, the name of the called subprogram is included in the trace data if the calling subprogram was enabled by compiling it in debug mode.

Steps 2 and 3: Identify a Trace Level and Start Tracing

- Specify the trace level by using dbms_trace.set_plsql_trace:

```
EXECUTE DBMS_TRACE.SET_PLSQL_TRACE -  
(tracelevel1 + tracelevel2 ...)
```

- Execute the code that is to be traced:

```
EXECUTE my_program
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

To trace PL/SQL code execution by using dbms_trace, perform the following steps:

- Start the trace session by using the syntax shown in the slide. For example:

```
EXECUTE -  
DBMS_TRACE.SET_PLSQL_TRACE(DBMS_TRACE.trace_all_calls)
```
- Execute the PL/SQL code. The trace data is written to the data dictionary views.

Note: To specify additional trace levels in the argument, use the “+” symbol between each trace level value.

Step 4: Turn Off Tracing

Remember to turn tracing off by using the dbms_trace.clear_plsql_trace procedure:

```
EXECUTE DBMS_TRACE.CLEAR_PLSQL_TRACE
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

After tracing the PL/SQL program unit, turn tracing off by executing dbms_trace.clear_plsql_trace. This stops any further writing to the trace tables.

To avoid the overhead of writing the trace information, it is recommended that you turn tracing off when you are not using it.

Step 5: Examine the Trace Information

Examine the trace information:

- Call tracing writes out the program unit type, name, and stack depth.
- Exception tracing writes out the line number.



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

- Lower trace levels supersede higher levels when tracing is activated for multiple tracing levels.
- If tracing is requested only for enabled subprograms and if the current subprogram is not enabled, no trace data is written.
- If the current subprogram is enabled, call tracing writes out the subprogram type, name, and stack depth.
- If the current subprogram is not enabled, call tracing writes out the subprogram type, line number, and stack depth.
- Exception tracing writes out the line number. Raising the exception shows information about whether the exception is user-defined or predefined and, in the case of predefined exceptions, the exception number.

Note: An enabled subprogram is compiled with the debug option.

plsql_trace_runs and plsql_trace_events

- Trace information is written to the following dictionary views:
 - plsql_trace_runs
 - plsql_trace_events
- Run the `tracetab.sql` script to create the dictionary views.
- You need privileges to view the trace information in the dictionary views.



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

All trace information is written to the dictionary views `plsql_trace_runs` and `plsql_trace_events`. These views are created (typically by a DBA) by running the `tracetab.sql` script. The script is located in the `/u01/app/oracle/product/12.1.0/dbhome_1/rdbms/admin` folder. Run the script as SYS. After the script is run, you need the `SELECT` privilege to view information from these dictionary views.

```
--Execute as sys
@/u01/app/oracle/product/12.1.0/dbhome_1/rdbms/admin/tracetab
GRANT SELECT ON plsql_trace_runs TO OE;
GRANT SELECT ON plsql_trace_events TO OE;
```

plsql_trace_runs and plsql_trace_events

```
ALTER SESSION SET PLSQL_DEBUG=TRUE;
ALTER PROCEDURE P5 COMPILE DEBUG;

EXECUTE DBMS_TRACE.SET_PLSQL_TRACE(DBMS_TRACE.trace_all_calls)
EXECUTE p5
EXECUTE DBMS_TRACE.CLEAR_PLSQL_TRACE

SELECT proc_name, proc_line,
       event_proc_name, event_comment
FROM sys.plsql_trace_events
WHERE event_proc_name = 'P5'
OR PROC_NAME = 'P5';
```

PROC_NAME	PROC_LINE	EVENT_PROC_NAME	EVENT_COMMENT
P5	1		Procedure Call
P4	1 P5		Procedure Call

2 rows selected.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Query the plsql_trace_runs and plsql_trace_events Views

Use the `plsql_trace_runs` and `plsql_trace_events` dictionary views to view the trace information generated by using the `dbms_trace` facility. `plsql_trace_runs` holds generic information about traced programs, such as the date, time, owner, and name of the traced stored program. `dbms_trace_events` holds more specific information about the traced subprograms.

Lesson Agenda

- Tracing PL/SQL program execution
- Profiling PL/SQL applications



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Hierarchical Profiling Concepts

Definition

- ❑ Used to identify hotspots and performance tuning opportunities in PL/SQL applications
- ❑ Reports the dynamic execution profile of a PL/SQL program organized by function calls
- ❑ Reports SQL and PL/SQL execution times separately
- ❑ Provides function-level summaries

Benefits

- ❑ Provides more information than a flat profiler
- ❑ Can be used to understand the structure and control flow of complex programs



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The hierarchical profiler is available to help you identify the hotspots and performance tuning opportunities in your PL/SQL applications. This feature enables you to view reports of how a PL/SQL program is executed, organized by function call, as well as SQL and PL/SQL execution times.

You can view function-level summaries that include the:

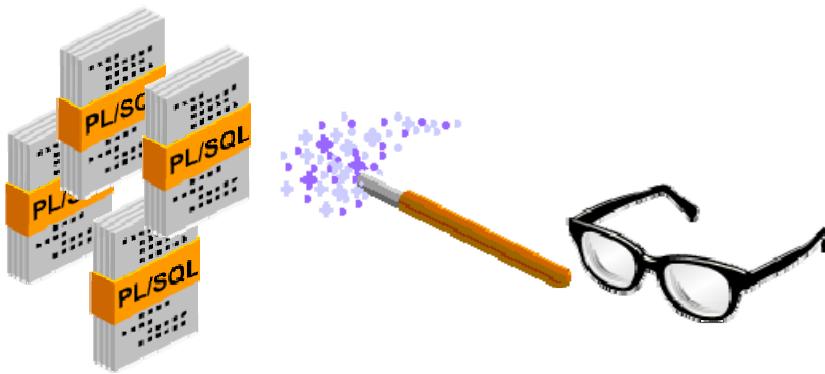
- Number of calls to a function
- Time spent in the function itself
- Time spent in the entire subtree under the function
- Detailed parent-children information for each function

You can use this information to tune your PL/SQL applications, and understand the structure, flow, and control of complex programs (especially those written by someone else).

Hierarchical Profiling Concepts

The PL/SQL hierarchical profiler consists of the:

- Data collection component
- Analyzer component



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The PL/SQL hierarchical profiler consists of two subcomponents:

- The data collection component is an intrinsic part of the PL/SQL Virtual Machine. The new `DBMS_HPROF` package provides APIs to turn hierarchical profiling on and off. The raw profiler output is written to a file.
- The analyzer component, which is also exposed by the `DBMS_HPROF` package, is used to process the raw profiler output and upload the results of the profiling into database tables that can then be queried. You can use the `plshprof` command-line utility to generate simple HTML reports directly from the raw profiler data.

Using the PL/SQL Profiler

Using the PL/SQL profiler, you can find:

- The number of calls to a function
- The function time, not including descendants
- The subtree time, including descendants
- Parent-children information for each function
 - Who were the callers of a given function?
 - What functions were called from a particular function?
 - How much time was spent in function X when called from function Y?
 - How many calls to function X came from function Y?
 - How many times did X call Y?



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Using the hierarchical PL/SQL profiler, you can find both function-level execution summary information, as well as detailed parent-children information for each function.

When profiling is turned on, function entry and exit operations are logged to a file along with time information. Fine-grained elapsed information is collected.

You do not need to recompile PL/SQL modules to use the hierarchical profiler. You can analyze both interpreted and natively compiled PL/SQL modules.

Using the PL/SQL Profiler

Sample data for profiling:

```
CREATE OR REPLACE PACKAGE BODY credit_card_pkg
IS
  PROCEDURE update_card_info
    (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no VARCHAR2)
  IS
    v_card_info typ_cr_card_nst;
    i INTEGER;
  BEGIN
    SELECT credit_cards
    ...
  END update_card_info;
  PROCEDURE display_card_info
    (p_cust_id NUMBER)
  IS
    v_card_info typ_cr_card_nst;
    i INTEGER;
  BEGIN
    SELECT credit_cards
    ...
  END display_card_info;
END credit_card_pkg; -- package body
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

CREDIT_CARD_PKG that was created earlier is used to demonstrate hierarchical profiling. The full code is shown on the following pages.

```
CREATE OR REPLACE PACKAGE credit_card_pkg
IS
    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no
VARCHAR2) ;

    PROCEDURE display_card_info
        (p_cust_id NUMBER) ;
END credit_card_pkg; -- package spec
/

CREATE OR REPLACE PACKAGE BODY credit_card_pkg
IS
    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no
VARCHAR2)
    IS
        v_card_info typ_cr_card_nst;
        i INTEGER;
    BEGIN
        SELECT credit_cards
            INTO v_card_info
            FROM customers
            WHERE customer_id = p_cust_id;
        IF v_card_info.EXISTS(1) THEN -- cards exist, add more
            i := v_card_info.LAST;
            v_card_info.EXTEND(1);
            v_card_info(i+1) := typ_cr_card(p_card_type, p_card_no);
            UPDATE customers
                SET credit_cards = v_card_info
                WHERE customer_id = p_cust_id;
        ELSE -- no cards for this customer yet, construct one
            UPDATE customers
                SET credit_cards = typ_cr_card_nst
                    (typ_cr_card(p_card_type, p_card_no))
                WHERE customer_id = p_cust_id;
        END IF;
    END update_card_info;
```

```
PROCEDURE display_card_info
    (p_cust_id NUMBER)
IS
    v_card_info typ_cr_card_nst;
    i INTEGER;
BEGIN
    SELECT credit_cards
        INTO v_card_info
        FROM customers
        WHERE customer_id = p_cust_id;
    IF v_card_info.EXISTS(1) THEN
        FOR idx IN v_card_info.FIRST..v_card_info.LAST LOOP
            DBMS_OUTPUT.PUT('Card Type: ' || 
v_card_info(idx).card_type
                                || ' ');
            DBMS_OUTPUT.PUT_LINE('/ Card No: ' ||
v_card_info(idx).card_num );
        END LOOP;
    ELSE
        DBMS_OUTPUT.PUT_LINE('Customer has no credit cards.');
    END IF;
END display_card_info;
END credit_card_pkg; -- package body
/
```

Using the PL/SQL Profiler

```
BEGIN
-- start profiling
DBMS_HPROF.START_PROFILING('PROFILE_DATA', 'pd_cc_pkg.txt');
END;
```

1

```
DECLARE
  v_card_info typ_cr_card_nst;
BEGIN
-- run application
  credit_card_pkg.update_card_info
    (154, 'Discover', '123456789');
END;
```

2

```
BEGIN
  DBMS_HPROF.STOP_PROFILING;
END;
```

3

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You use the new DBMS_HPROF package to hierarchically profile PL/SQL code. You must be granted the privilege to execute the routines in the DBMS_HPROF package:

```
GRANT EXECUTE ON sys.dbms_hprof TO OE;
```

You also need to identify the location of the profiler files. Create a DIRECTORY object to identify this information:

```
CREATE OR REPLACE DIRECTORY profile_data AS
  '/home/oracle/labs/labs';
```

1. The first step is to turn profiling on. Use the DBMS_HPROF.START_PROFILING procedure. When calling this procedure, pass the following two parameters:
 - **Directory object:** An alias for a file system path name. You need to have WRITE privileges to this location.
 - **File name:** The name of the file to which you want the output written
 - You can optionally pass a third parameter, max_depth. When max_depth value is NULL (the default), profile information is gathered for all functions irrespective of their call depth. When a non-NULL value is specified, the profiler collects data only for functions up to a call depth level of max_depth.
2. The second step is to run the code that you want profiled.

3. The third step is to turn off profiling. Use the DBMS_HPROF.STOP_PROFILING procedure to stop the profiling:

```
EXECUTE DBMS_HPROF.STOP_PROFILING
```

Understanding Raw Profiler Data

```
P#! PL/SQL Timer Started
P#C PLSQL.""".__plsql_vm"
P#X 3
P#C PLSQL.""".__anonymous_block"
P#X 1634
P#C PLSQL."OE"."CREDIT_CARD_PKG":11."UPDATE_CARD_INFO"#71749359b90ac246
#24
P#X 7
P#C PLSQL."OE"."CREDIT_CARD_PKG":11."CUST_CARD_INFO"#c2ad85321cb9b0ae
#4
P#X 11
P#C SQL."OE"."CREDIT_CARD_PKG":11.__static_sql_exec_line10" #10
P#X 1502
P#R
...
P#C PLSQL.""".__plsql_vm"
P#X 3
P#C PLSQL.""".__anonymous_block"
P#X 15
P#C PLSQL."SYS"."DBMS_HPROF":11."STOP_PROFILING"#980980e97e42f8ec #53
P#R
P#R
P#! PL/SQL Timer Stopped
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You can examine the contents of the generated raw profiler text file. However, it is easier to understand the data after using the analyzer component of hierarchical profiling. The text shown in the slide displays the contents of the `pd_cc_pkg.txt` file. Each line starts with a file indicator. These are the meanings:

- P#V – PLSHPROF banner with version number
- P#C – Call to a subprogram (call event)
- P#R – Return from a subprogram (call event)
- P#X – Elapsed time between preceding and following events
- P#! – Comment

As you will see in the section “Using DBMS_HPROF.ANALYZE,” the DBMS_HPROF.ANALYZE function generates easier-to-decipher data and saves the data in tables.

Using the Hierarchical Profiler Tables

- Upload the raw profiler data into the database tables.
- Run the `dbmshptab.sql` script that is located in the `ORACLE_HOME/rdbms/admin` folder to set up the profiler tables.

```
-- run this only once per schema
-- under the schema where you want the profiler tables located
@u01/app/oracle/product/12.1.0/dbhome_1/rdbms/admin/dbmshptab.sql
```

- Creates these tables:

Table	Description
DBMSHP_RUNS	Contains top-level information for each run command
DBMSHP_FUNCTION_INFO	Contains information on each function profiled
DBMSHP_PARENT_CHILD_INFO	Contains parent-child profiler information



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Before uploading the profiler data into the database tables, you must create the hierarchical profiler database tables. Run the `dbmshptab.sql` script under the schema where you want the profiling tables to create the hierarchical profiling tables. This script is located in your `/home/rdbms/admin/` folder.

The script creates three tables and other data structures that are required for persistently storing the profiler data.

Note: Running the script a second time drops any previously created hierarchical profiler tables.

Using DBMS_HPROF.ANALYZE

DBMS_HPROF.ANALYZE:

- Analyzes the raw profiler data
- Generates hierarchical profiler information in the profiler database tables
- Definition:

```
DBMS_HPROF.ANALYZE(
    location      IN VARCHAR2,
    filename      IN VARCHAR2,
    summary_mode  IN BOOLEAN      DEFAULT FALSE,
    trace         IN VARCHAR2    DEFAULT NULL,
    skip          IN PLS_INTEGER DEFAULT 0,
    collect        IN PLS_INTEGER DEFAULT NULL,
    run_comment   IN VARCHAR2    DEFAULT NULL)
RETURN NUMBER;
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Use the DBMS_HPROF.ANALYZE function to analyze the raw profiler output and produce the hierarchical profiler information in the database.

This function accepts the following parameters:

- **Location:** The name of the directory object that identifies the location from which to read
- **Filename:** The file name of the raw profiler data to be analyzed
- **summary_mode:** A Boolean that identifies whether to generate only top-level summary information into the database tables (TRUE) or to provide detailed analysis (FALSE). The default is false.
- **Trace:** Identifies whether to analyze only the subtrees rooted at the specified trace entry or perform the analysis for the entire run. This parameter is specified in a special, qualified format within quotation marks. It includes the schema name, module name, and function name (for example "SCOTT"."PKG"."FOO").
- **Skip:** Analyzes only the subtrees rooted at the specified trace, but ignores the first "skip" invocations to trace. The default value for "skip" is 0. Used only when trace is specified.
- **Collect:** Analyzes "collect" member of invocations of traces (starting from "skip" + 1). By default, only one invocation is collected. It is used only when trace is specified.
- **run_comment:** A comment for your run

Using DBMS_HPROF.ANALYZE to Write to Hierarchical Profiler Tables

- Use the DBMS_HPROF.ANALYZE function to upload the raw profiler results into the database tables:

```
DECLARE
  v_runid NUMBER;
BEGIN
  v_runid := DBMS_HPROF.ANALYZE (LOCATION => 'PROFILE_DATA',
                                    FILENAME => 'pd_cc_pkg.txt');
  DBMS_OUTPUT.PUT_LINE('Run ID: ' || v_runid);
END;
```

- This function returns a unique run identifier for the run. You can use this identifier to look up results corresponding to this run from the hierarchical profiler tables.

RUN_ID

1



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In the example shown in the slide, the DBMS_HPROF.ANALYZE function is used to take the raw data from the text file named pd_cc_pkg.txt and place the resulting analyzed data in the profiler tables. The function returns a unique identifier for the run.

Analyzer Output from the DBMSHP_RUNS Table

Query the DBMSHP_RUNS table to find top-level information for each run:

```
SELECT runid, run_timestamp, total_elapsed_time  
FROM dbmshp_runs  
WHERE runid = 2;
```

RUNID	RUN_TIMESTAMP	TOTAL_ELAPSED_TIME
2	10-DEC-09 02.10.47.604825000 AM	120758



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The DBMS_HPROF.ANALYZE PL/SQL API is used to analyze the raw profiler output and upload the results into the database tables. This enables you to query the data for custom report generation using tools, such as SQL Developer or other third-party tools.

In the example shown in the slide, the DBMSHP_RUNS table is queried after the DBMS_HPROF.ANALYZE is run. This table contains top-level information for each run of the DBMS_HPROF.ANALYZE command. From the output, you can see the run ID, the timestamp of the run, and the total elapsed time in milliseconds. If you provided a comment for the run, you can retrieve that information from the RUN_COMMENT column (not shown).

Analyzer Output from the DBMSHP_FUNCTION_INFO Table

Query the DBMSHP_FUNCTION_INFO table to find information about each function profiled:

```
SELECT owner, module, type, function_line#, namespace,
       calls, function_elapsed_time
  FROM dbmshp_function_info
 WHERE runid = 2;
```

OWNER	MODULE	TYPE	LINE#	NAMESPACE	CALLS	FUNCTION_ELAPSED
(null)	(null)	(null)	_anonymous_block	PLSQL	3	
(null)	(null)	(null)	_plsql_vm	PLSQL	3	
OE	CREDIT_CARD_PKG	PACKAGE BODY	UPDATE_CARD_INFO	PLSQL	1	
SYS	DBMS_HPROF	PACKAGE BODY	STOP_PROFILING	PLSQL	1	
(null)	(null)	(null)	_dyn_sql_exec_line5	SQL	1	
OE	CREDIT_CARD_PKG	PACKAGE BODY	_static_sql_exec_line21	SQL	1	
OE	CREDIT_CARD_PKG	PACKAGE BODY	_static_sql_exec_line9	SQL	1	



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You can query the DBMSHP_FUNCTION_INFO table to find summary information for each function profiled in a run of the analyzer.

In the query shown in the slide, the following information is retrieved:

- OWNER: Module owner of the function
- MODULE: The module name
- TYPE: The module type (such as package, package body, or procedure)
- LINE#: The line number in the module at which the function is defined. This line number helps to identify the source location of the function in the module and can be used by the integrated development environment (IDE) tools to navigate to the appropriate location in the source where the function is defined. The line number can also be used to distinguish between overloaded routines.
- NAMESPACE: The language information. At this time, SQL and PL/SQL are supported.
- CALLS: The number of calls to the function
- FUNCTION_ELAPSED_TIME: The time in microseconds, not including the time spent in descendant functions

plshprof: A Simple HTML Report Generator

- plshprof is a command-line utility.
- You can use plshprof to generate simple HTML reports directly from the raw profiler data.
- The HTML reports can be browsed in any browser.
- The navigational capabilities combined with the links provide a means for you to analyze the performance of large applications.



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

plshprof is a command-line utility that you can use to generate HTML reports based on the raw profiler data generated by the data collection component after running the DBMS_HPROF.ANALYZE PL/SQL API.

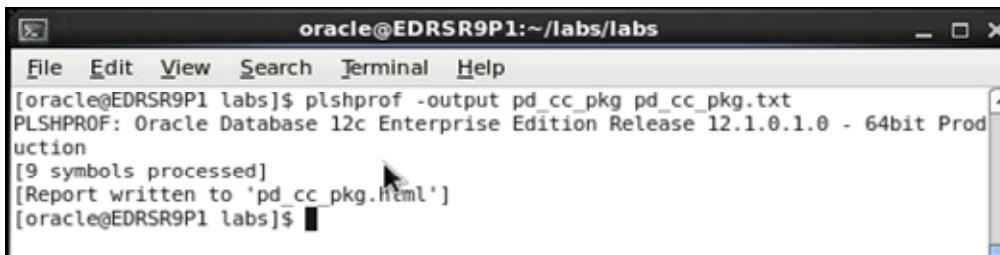
Using plshprof

After generating the raw profiler output file:

1. Change to the directory where you want the HTML output placed
2. Run plshprof
 - Syntax:

```
plshprof [option...] output_filename_1 output_filename_2
```

- Example:



A screenshot of a terminal window titled "oracle@EDRSR9P1:~/labs/labs". The window shows the command "plshprof -output pd_cc_pkg pd_cc_pkg.txt" being run. The output indicates that 9 symbols were processed and a report was written to "pd_cc_pkg.html". The terminal prompt "[oracle@EDRSR9P1 labs]\$" is visible at the bottom.

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

After you have the raw profiler output file, you can generate HTML reports for that run.

The plshprof utility has the following options:

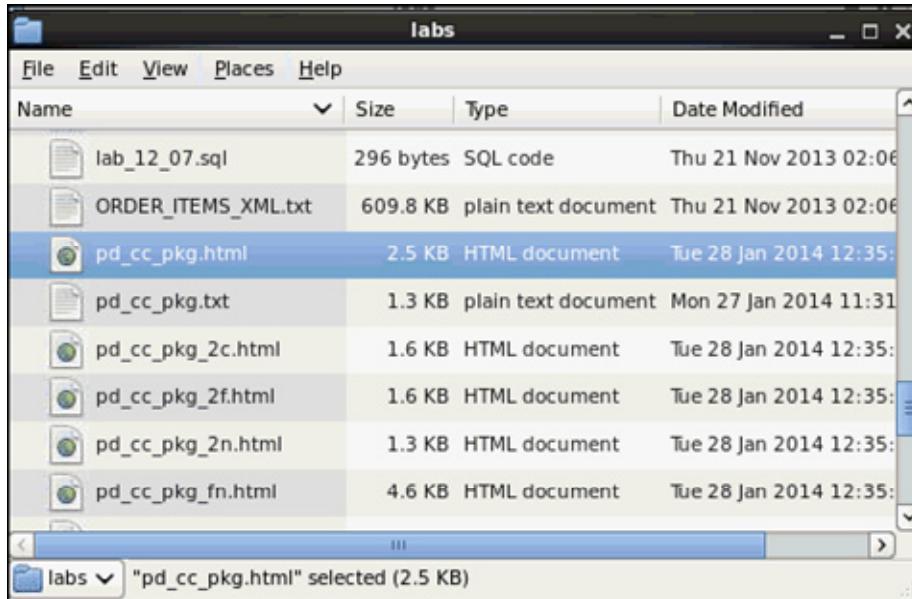
- ***-skip count***: Skips the first count calls. Use only with the ***-trace symbol***. The default is 0.
- ***-collect count***: Collects information for count calls. Use only with the ***-trace symbol***. The default is 1.
- ***-output filename***: Specifies the name of the output file ***symbol.html*** or ***tracefile1.html***.
- ***-summary***: Prints only elapsed time; no default
- ***-trace symbol***: Specifies the function name of the tree root

To generate the HTML reports, perform the following steps:

1. Change to the directory of the raw profiler output file. In the example shown in the slide, the raw profiler file ***pd_dd_pkg.txt*** is in the ***/home/oracle/labs/labs*** folder.
2. Run the ***plshprof*** utility. This utility accepts line arguments. The example in the slide identifies that the output file should be named ***pd_cc_pkg.html*** and the original raw profiler file is named ***pd_dd_pkg.txt***.

Using plshprof

Generated files:



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

ORACLE

Running the `plshprof` utility generates a set of HTML files. `filename.html` is the root page where you start browsing. The other files with the same file name prefix are hyperlinks from the base `filename.html` file. In the example in the slide, the root page is named `pd_cc_pkg.html`.

Using plshprof

3. Open filename.html in a browser:

The screenshot shows a web browser window with the URL `file:///home/oracle/labs/labs/pd_cc_pkg.html` highlighted in red at the top. The page title is **PL/SQL Elapsed Time (microsecs) Analysis**. Below the title, it says **120758 microsecs (elapsed time) & 11 function calls**. A descriptive text block states: **The PL/SQL Hierarchical Profiler produces a collection of reports that present information derived from the profiler's output log in a variety of formats. The following reports have been found to be the most generally useful as starting points for browsing:** followed by a list of links. Another section titled **In addition, the following reports are also available:** lists more links.

- [Function Elapsed Time \(microsecs\) Data sorted by Total Subtree Elapsed Time \(microsecs\)](#)
- [Function Elapsed Time \(microsecs\) Data sorted by Total Function Elapsed Time \(microsecs\)](#)

In addition, the following reports are also available:

- [Function Elapsed Time \(microsecs\) Data sorted by Function Name](#)
- [Function Elapsed Time \(microsecs\) Data sorted by Total Descendants Elapsed Time \(microsecs\)](#)
- [Function Elapsed Time \(microsecs\) Data sorted by Total Function Call Count](#)
- [Function Elapsed Time \(microsecs\) Data sorted by Mean Subtree Elapsed Time \(microsecs\)](#)
- [Function Elapsed Time \(microsecs\) Data sorted by Mean Function Elapsed Time \(microsecs\)](#)
- [Function Elapsed Time \(microsecs\) Data sorted by Mean Descendants Elapsed Time \(microsecs\)](#)
- [Module Elapsed Time \(microsecs\) Data sorted by Total Function Elapsed Time \(microsecs\)](#)

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

3. You can open the start page file in any browser. In this example, the `pd_cc_pkg.html` file is opened. This is an example of a single-run report. It contains some overall summary information and hyperlinks to additional information.

Using the HTML Reports

useful as starting points for browsing:

- [Function Elapsed Time \(microsecs\) Data sorted by Total Subtree Elapsed Time \(microsecs\)](#)
- [Function Elapsed Time \(microsecs\) Data sorted by Total Function Elapsed Time \(microsecs\)](#)

Function Elapsed Time (microsecs) Data sorted by Total Subtree Elapsed Time (microsecs)

120758 microsecs (elapsed time) & 11 function calls

Subtree	Ind%	Function	Ind%	Descendants	Ind%	Calls	Ind%	Function Name
120758	100%		17	0.0%	120741	100%	3	27.3% plsql_vm
120741	100%		1323	1.1%	119418	98.9%	3	27.3% anonymous_block
119292	98.8%		170	0.1%	119122	98.6%	1	9.1% OE.CREDIT_CARD_PKG.UPDATE_CARD_INFO (Line 3)
92954	77.0%		92954	77.0%		0	0.0%	1 9.1% OE.CREDIT_CARD_PKG._static_sql_exec_line9 (Line 9)
26168	21.7%		26168	21.7%		0	0.0%	1 9.1% OE.CREDIT_CARD_PKG._static_sql_exec_line21 (Line 21)
126	0.1%		126	0.1%		0	0.0%	1 9.1% dyn_sql_exec_line5 (Line 5)
0	0.0%		0	0.0%	0	0.0%	1	9.1% SYS.DBMS_HPROF.STOP_PROFILING (Line 59)



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The function-level summary reports provide you with a flat view of the profile information. The reports list the total number of calls, self time, subtree time, and descendants for each function. Each report is sorted on a particular attribute.

Note that in the example in the slide, each function name is hyperlinked to its corresponding parents and children report. The column that is in bold identifies how the report is sorted.

If a subprogram is nested, the profiler reports display the fully qualified name, such as OE.P.A.B; that is, procedure B is nested within procedure A of package OE.P.

Using the HTML Reports

useful as starting points for browsing:

- [Function Elapsed Time \(microsecs\) Data sorted by Total Subtree Elapsed Time \(microsecs\)](#)
- [Function Elapsed Time \(microsecs\) Data sorted by Total Function Elapsed Time \(microsecs\)](#)

Function Elapsed Time (microsecs) Data sorted by Total Function Elapsed Time (microsecs)

120758 microsecs (elapsed time) & 11 function calls

Subtree	Ind%	Function	Ind%	Cum%	Descendants	Ind%	Calls	Ind%	Function Name
92954	77.0%	92954	77.0%	77.0%	0	0.0%	1	9.1%	OE.CREDIT_CARD_PKG._static_sql_exec_line9 (Line 9)
26168	21.7%	26168	21.7%	98.6%	0	0.0%	1	9.1%	OE.CREDIT_CARD_PKG._static_sql_exec_line21 (Line 21)
120741	100%	1323	1.1%	99.7%	119418	98.9%	3	27.3%	anonymous_block
119292	98.8%	170	0.1%	99.9%	119122	98.6%	1	9.1%	OE.CREDIT_CARD_PKG.UPDATE_CARD_INFO (Line 3)
126	0.1%	126	0.1%	100%	0	0.0%	1	9.1%	dyn_sql_exec_line5 (Line 5)
120758	100%	17	0.0%	100%	120741	100%	3	27.3%	plsql_vm
0	0.0%	0	0.0%	100%	0	0.0%	1	9.1%	SYS.DBMS_HPROF.STOP_PROFILING (Line 59)

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You can use the module-level summary report to find information about each PL/SQL module, including the total time spent in the module and the total number of calls to functions in the module. The total time spent in a module is calculated by adding the self times for all functions in that module.

Using the HTML Reports

- [Namespace Elapsed Time \(microsecs\) Data sorted by Namespace](#)
- [Namespace Elapsed Time \(microsecs\) Data sorted by Total Function Call Count](#)
- [Parents and Children Elapsed Time \(microsecs\) Data](#)

Namespace Elapsed Time (microsecs) Data sorted by Namespace

120758 microsecs (elapsed time) & 11 function calls

Function	Ind%	Calls	Ind%	Namespace
1510	1.3%	8	72.7%	PLSQL
119248	98.7%	3	27.3%	SQL

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Each function tracked by the profiler is associated with a namespace. PL/SQL functions, procedures, triggers, and packages fall under the category “PLSQL” namespace. Operations corresponding to SQL statement execution from PL/SQL, such as UPDATE, SELECT, FETCH, and EXECUTE IMMEDIATE, fall under the “SQL” namespace. The namespace-level summary report provides information about the total time spent in that namespace and the total number of calls to functions in that namespace.

Using the HTML Reports

ORACLE

- [Namespace Elapsed Time \(microsecs\) Data sorted by Namespace](#)
- [Namespace Elapsed Time \(microsecs\) Data sorted by Total Function Call Count](#)
- [Parents and Children Elapsed Time \(microsecs\) Data](#)

_anonymous_block

Subtree	Ind%	Function	Ind%	Descendants	Ind%	Calls	Ind%	Function Name
120741	100%	1323	1.1%	119418	98.9%	3	27.3%	anonymous_block
Parents:								
120741	100%	1323	100%	119418	100%	3	100%	plsql_vm
Children:								
119292	99.9%	170	100%	119122	100%	1	100%	OE.CREDIT_CARD_PKG.UPDATE_CARD_INFO (Line 3)
126	0.1%	126	100%	0	N/A	1	100%	dyn_sql_exec_line5 (Line 5)
0	0.0%	0	N/A	0	N/A	1	100%	SYS.DBMS_HPROF.STOP_PROFILING (Line 59)

_plsql_vm

Subtree	Ind%	Function	Ind%	Descendants	Ind%	Calls	Ind%	Function Name
120758	100%	17	0.0%	120741	100%	3	27.3%	plsql_vm
Parents:								
120758	100%	17	100%	120741	100%	3	100%	ORACLE.root
Children:								
120741	100%	1323	100%	119418	100%	3	100%	anonymous_block

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

For each function tracked by the profiler, the parents and children report provides information about what functions call it (parents) and what functions it calls (children). It gives you the function's execution profile on a per-parent basis. It also provides the execution profile for each of the function's children when called from this function.

Quiz

Select the correct order of the five steps to trace PL/SQL code using the `dbms_trace` package:

- A. Enable specific program units for trace data collection.
 - B. Use `dbms_trace.clear_plsql_trace` to stop tracing data.
 - C. Run your PL/SQL code.
 - D. Read and interpret the trace information.
 - E. Use `dbms_trace.set_plsql_trace` to identify a trace level.
- a.** A, E, C, B, D
b. A, B, C, D, E
c. A, C, E, B, D
d. A, E, C, D, B



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Answer: a

Quiz

You can view the hierarchical profiler function level summaries that include:

- a. Number of calls to a function
- b. Time spent in the function itself
- c. Time spent in the entire subtree under the function
- d. Detailed parent-children information for each function



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Answer: a, b, c, d

Quiz

Use the _____ and the _____ to find information about each function profiled by the PL/SQL profiler.

- a. DBMS_HPROF.ANALYZE table
- b. DBMSHP_RUNS table
- c. DBMSHP_FUNCTION_INFO table
- d. plshprof command-line utility



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Answer: a, d

Summary

In this lesson, you should have learned how to:

- Trace PL/SQL program execution
- Profile PL/SQL applications



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

This lesson showed you how to trace PL/SQL program execution by using the `DBMS_TRACE` package, how to profile your PL/SQL application, and, with profiling, how to set up the profiler tables, generate raw data, and use `DBMS_HPROF` to analyze raw data in the profiler tables.

Practice 11: Overview

This practice covers hierarchical profiling of PL/SQL code.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Using the OE application that you have created, write code to profile components in your application.

Use the OE schema for this practice.

12

Implementing Fine-Grained Access Control for VPD

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Objectives

After completing this lesson, you should be able to do the following:

- Describe the process of fine-grained access control
- Implement and test fine-grained access control



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In this lesson, you learn about the security features in the Oracle Database from an application developer's standpoint.

For more information about these features, refer to *Oracle Supplied PL/SQL Packages and Types Reference*, *Oracle Label Security Administrator's Guide*, *Oracle Single Sign-On Application Developer's Guide*, and *Oracle Security Overview*.

Lesson Agenda

- Describing the process of fine-grained access control
- Implementing and testing fine-grained access control

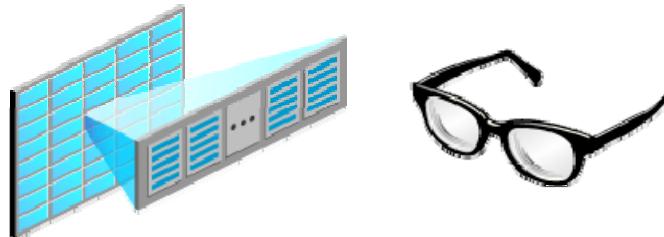


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Fine-Grained Access Control: Overview

Fine-grained access control:

- Enables you to enforce security through a low level of granularity
- Restricts users to viewing only “their” information
- Is implemented through a security policy attached to tables
- Is implemented by highly privileged system DBAs, perhaps in coordination with developers
- Dynamically modifies user statements to fit the policy



ORACLE®

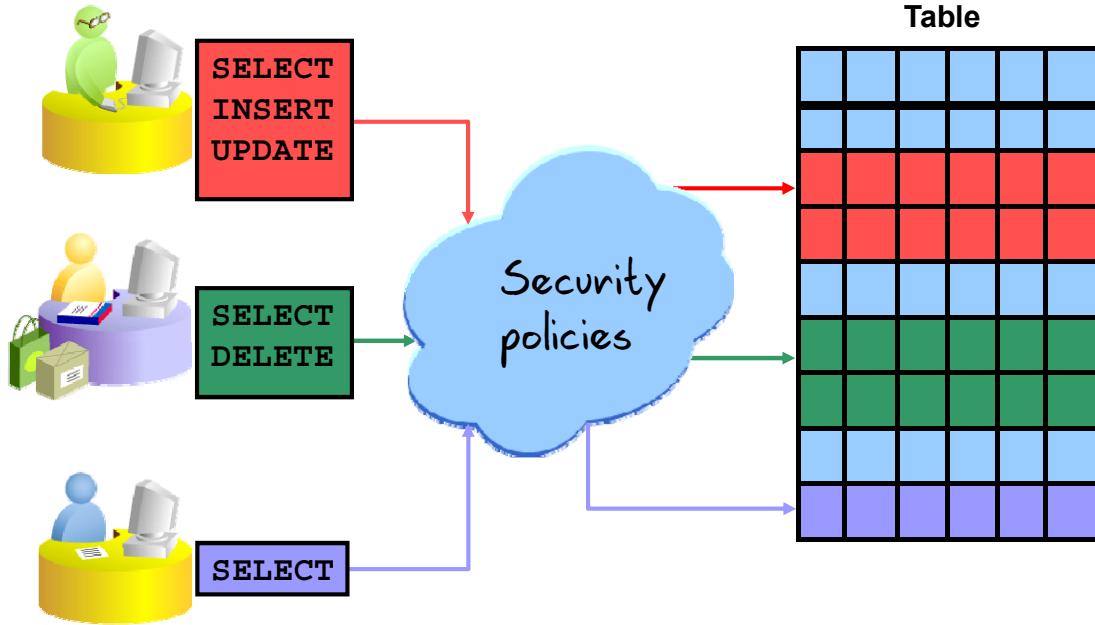
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Fine-grained access control enables you to build applications that enforce security rules (or policies) at a low level of granularity. For example, you can use it to restrict customers who access the Oracle server to see only their own account, physicians to see only the records of their own patients, or managers to see only the records of employees who work for them.

When you use fine-grained access control, you create security policy functions attached to the table or view on which you base your application. When a user enters a data manipulation language (DML) statement on that object, the Oracle server dynamically modifies the user's statement—transparently to the user—so that the statement implements the correct access control.

Fine-grained access is also known as a virtual private database (VPD), because it implements row-level security, essentially giving users access to their own private database. Fine-grained means at the individual row level.

Identifying Fine-Grained Access Features



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You can use fine-grained access control to implement security rules, called “policies,” with functions, and then associate those security policies with tables or views. The database server automatically enforces those security policies, no matter how the data is accessed.

A security policy is a collection of rules needed to enforce the appropriate privacy and security rules in the database itself, making it transparent to users of the data structure.

Attaching security policies to tables or views, rather than to applications, provides greater security, simplicity, and flexibility.

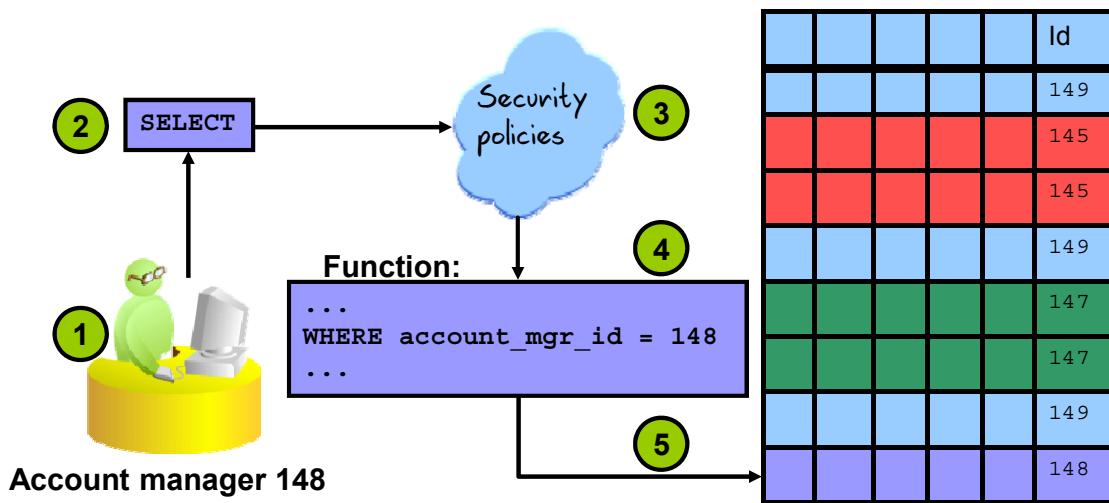
You can:

- Use different policies for SELECT, INSERT, UPDATE, and DELETE statements
- Use security policies only where you need them
- Use multiple policies for each table, including building on top of base policies in packaged applications
- Distinguish policies between different applications by using policy groups

How Fine-Grained Access Works

Implement the policy on the CUSTOMERS table:

“Account managers can see only their own customers.”



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

ORACLE

To implement a virtual private database so that account managers can see only their own customers, you must do the following:

1. Create a function to add a WHERE clause that identifies a selection criterion to a user's SQL statement.
 2. Have the user (the account manager) enter a SQL statement.
 3. Implement the security policy through the function that you created. The Oracle server calls the function automatically.
 4. Dynamically modify the user's statement through the function.
 5. Execute the dynamically modified statement.

How Fine-Grained Access Works

- You write a function to return the account manager ID (predicate value):

```
account_mgr_id := (SELECT account_mgr_id  
                      FROM   customers  
                     WHERE account_mgr_id =  
                           SYS_CONTEXT ('userenv','session_user'));
```

- The account manager user enters a query:

```
SELECT customer_id, cust_last_name, cust_email  
FROM   customers;
```

- The query is modified with the function results:

```
SELECT customer_id, cust_last_name, cust_email  
FROM   orders  
WHERE  account_mgr_id = (SELECT account_mgr_id  
                           FROM   customers  
                           WHERE account_mgr_id =  
                                 SYS_CONTEXT ('userenv','session_user'));
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Fine-grained access control is based on a dynamically modified statement. In the example in the slide, the user enters a broad query against the CUSTOMERS table that retrieves customer names and email names for a specific account manager. The Oracle server calls the function to implement the security policy. This modification is transparent to the user. It results in successfully restricting access to other customers' information, displaying only the information relevant to the account manager.

Note: The SYS_CONTEXT function returns a value for an attribute, in this case, connection attributes. This is explained in detail in the following slides.

Why Use Fine-Grained Access

To implement the business rule “Account managers can see only their own customers,” you have three options:

Option	Comment
Modify all existing application code to include a predicate (a WHERE clause) for all SQL statements.	Does not ensure privacy enforcement outside the application. Also, all application code may need to be modified in the future as business rules change.
Create views with the necessary Predicates, and then create synonyms with the same name as the table names for these views.	This can be difficult to administer, especially if there are a large number of views to track and manage.
Create a VPD for each of the account managers by creating policy functions to generate dynamic predicates. These predicates can then be applied across all objects.	This option offers the best security without major administrative overheads and it also ensures complete privacy of information. 



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

There are other methods (listed in the slide) by which you can implement the business rule “Account managers can see only their own customers.” However, by using fine-grained access, you implement security without major overheads.

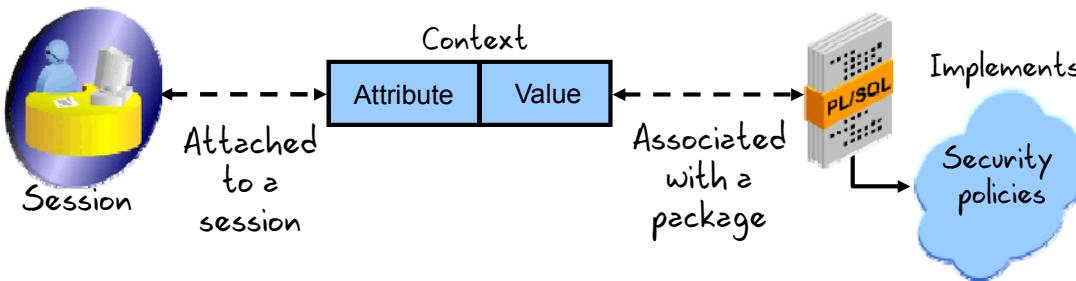
Lesson Agenda

- Describing the process of fine-grained access control
- Implementing and testing fine-grained access control



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Using an Application Context



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

An application context:

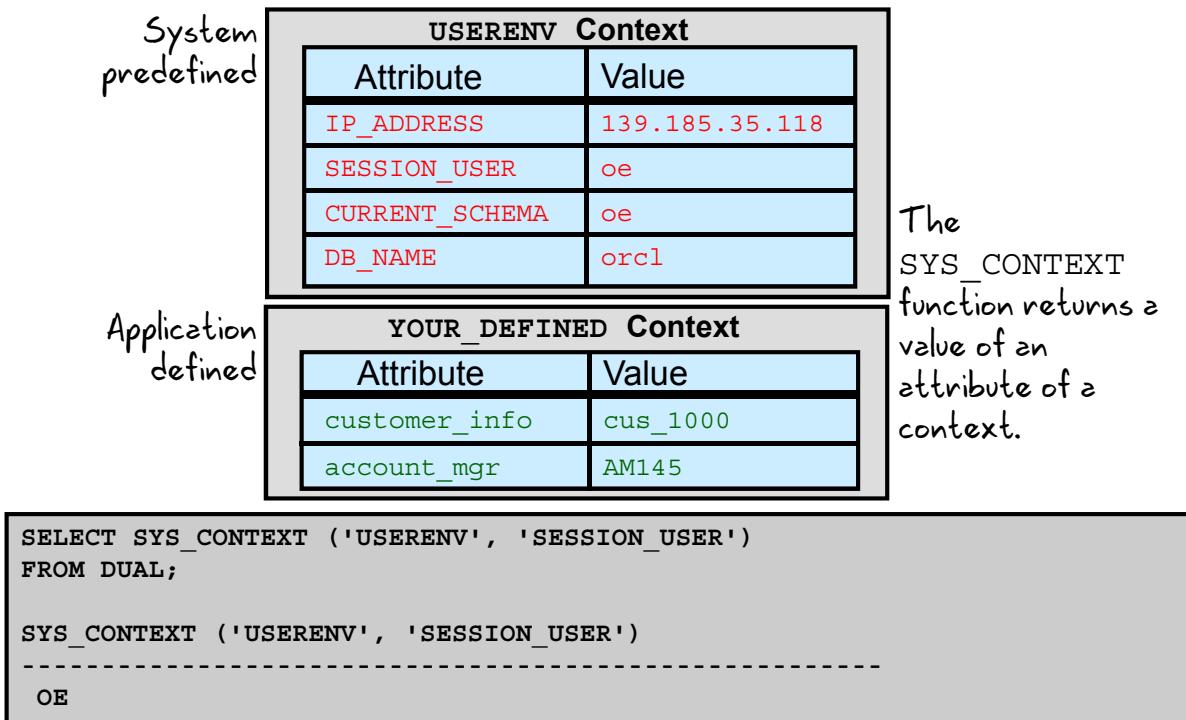
- Is used to facilitate the implementation of fine-grained access control
- Is a named set of attribute/value pairs associated with a PL/SQL package
- Is attached to a session
- Enables you to implement security policies with functions, and then associate them with applications

A context is a named set of attribute/value pairs that are global to your session. You can define an application context, name it, and associate a value with that context with a PL/SQL package. An application context enables you to write applications that draw upon certain aspects of a user's session information. It provides a way to define, set, and access attributes that an application can use to enforce access control—specifically, fine-grained access control.

Most applications contain information about the basis on which access is to be limited. In an order entry application, for example, you limit the customers' access to their own orders (`ORDER_ID`) and customer number (`CUSTOMER_ID`). Or, you may limit account managers (`ACCOUNT_MGR_ID`) to view only their own customers. These values can be used as security attributes. Your application can use a context to set values that are accessed within your code and used to generate `WHERE` clause predicates for fine-grained access control.

An application context is owned by `SYS`. Users cannot change their application's context.

Using an Application Context



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

A predefined application context named `USERENV` has a predefined list of attributes. Predefined attributes can be very useful for access control. You find the values of the attributes in a context by using the `SYS_CONTEXT` function. Although the predefined attributes in the `USERENV` application context are accessed with the `SYS_CONTEXT` function, you cannot change them.

With the `SYS_CONTEXT` function, you pass the context name and the attribute name. The attribute value is returned.

The following statement returns the name of the database that is being accessed:

```

SELECT SYS_CONTEXT ('USERENV', 'DB_NAME')
FROM DUAL;
  
```

```

SYS_CONTEXT ('USERENV', 'DB_NAME')
-----
ORCL
  
```

Creating an Application Context

```
CREATE [OR REPLACE] CONTEXT namespace
USING [schema.]plsql_package
```

- Requires the CREATE ANY CONTEXT system privilege
- Parameters:
 - *namespace* is the name of the context.
 - *schema* is the name of the schema owning the PL/SQL package.
 - *plsql_package* is the name of the package used to set or modify the attributes of the context. (It does not need to exist at the time of context creation.)

```
CREATE OR REPLACE CONTEXT order_ctx USING
oe.orders_app_pkg;
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

For fine-grained access where you want account managers to view only their customers, customers to view only their information, and sales representatives to view only their orders, you can create a context called ORDER_CTX and define for it the ACCOUNT_MGR, CUST_ID and SALE REP attributes, respectively.

Because a context is associated with a PL/SQL package, you need to name the package that you are associating with the context. This package does not need to exist at the time of context creation.

Setting a Context

- Use the supplied package procedure DBMS_SESSION.SET_CONTEXT to set a value for an attribute within a context.

```
DBMS_SESSION.SET_CONTEXT('context_name',
                          'attribute_name',
                          'attribute_value')
```

- Set the attribute value in the package that is associated with the context.

```
CREATE OR REPLACE PACKAGE orders_app_pkg
...
BEGIN
    DBMS_SESSION.SET_CONTEXT('ORDER_CTX',
                            'ACCOUNT_MGR',
                            v_user)
...

```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

When a context is defined, you can use the DBMS_SESSION.SET_CONTEXT procedure to set a value for an attribute within a context. The attribute is set in the package that is associated with the context.

```
CREATE OR REPLACE PACKAGE orders_app_pkg
IS
    PROCEDURE set_app_context;
END;
/
CREATE OR REPLACE PACKAGE BODY orders_app_pkg
IS
    c_context CONSTANT VARCHAR2(30) := 'ORDER_CTX';
    PROCEDURE set_app_context
    IS
        v_user VARCHAR2(30);
    BEGIN
        SELECT user INTO v_user FROM dual;
        DBMS_SESSION.SET_CONTEXT
            (c_context, 'ACCOUNT_MGR', v_user);
    END;
END;
/
```

In the example on the previous page, the ORDER_CTX context has the ACCOUNT_MGR attribute set to the current user logged (determined by the USER function).

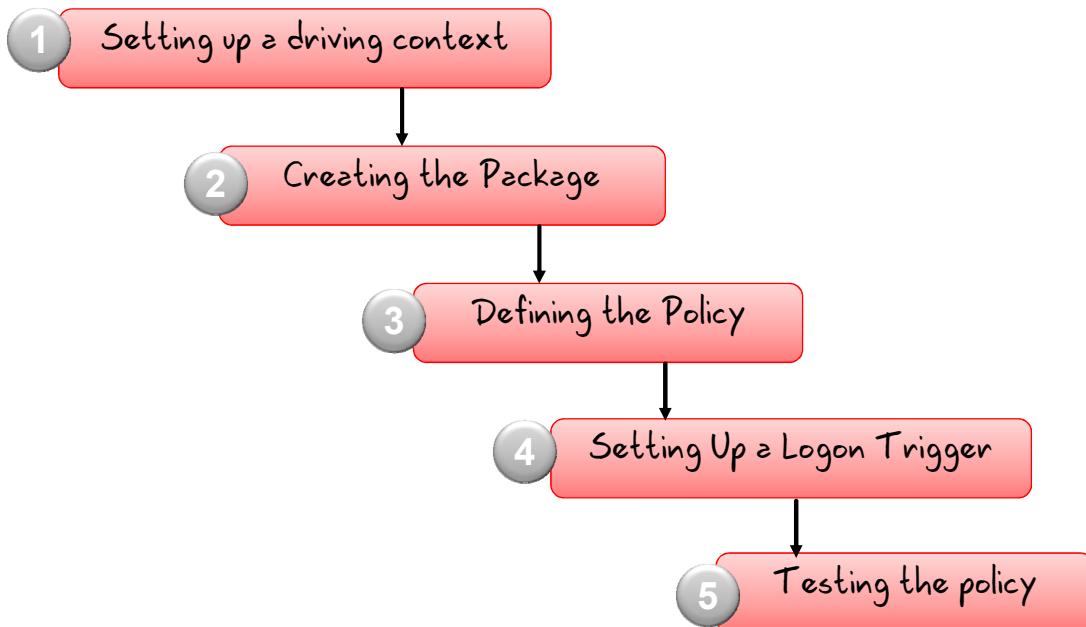
For this example, assume that users AM145, AM147, AM148, and AM149 exist. As each user logs on and the DBMS_SESSION.SET_CONTEXT is invoked, the attribute value for that ACCOUNT_MGR is set to the user ID.

```
GRANT EXECUTE ON oe.orders_app_pkg  
TO AM145, AM147, AM148, AM149;  
  
CONNECT AM145/oracle  
Connected.  
  
EXECUTE oe.orders_app_pkg.set_app_context  
  
SELECT SYS_CONTEXT('ORDER_CTX', 'ACCOUNT_MGR') FROM dual;  
  
SYS_CONTEXT('ORDER_CTX', 'ACCOUNT_MGR')  
-----  
AM145
```

If you switch the user ID, the attribute value is also changed to reflect the current user.

```
CONNECT AM147/oracle  
Connected.  
  
EXECUTE oe.orders_app_pkg.set_app_context  
  
SELECT SYS_CONTEXT('ORDER_CTX', 'ACCOUNT_MGR') FROM dual;  
  
SYS_CONTEXT('ORDER_CTX', 'ACCOUNT_MGR')  
-----  
AM147
```

Implementing a Policy



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The slide graphically describes the steps involved in implementing a policy. In the following slides, each of the steps is discussed in detail.

Step 1: Setting Up a Driving Context

Perform the following steps:

1. Set up a driving context.

```
CREATE OR REPLACE CONTEXT order_ctx  
  USING orders_app_pkg;
```

2. Create the package associated with the context that you defined in step 1. In the package:
 - a. Set the context
 - b. Define the predicate
3. Define the policy.
4. Set up a logon trigger to call the package at logon time and set the context.
5. Test the policy.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In this example, assume that the users AM145, AM147, AM148, and AM149 exist. Next, create a context and a package associated with the context. The package will be owned by OE.

Step 1: Set Up a Driving Context

Use the CREATE CONTEXT syntax to create a context:

```
CREATE CONTEXT order_ctx USING oe.orders_app_pkg;
```

Step 2: Creating the Package

```
CREATE OR REPLACE PACKAGE orders_app_pkg
IS
    PROCEDURE show_app_context;
    PROCEDURE set_app_context;
    FUNCTION the_predicate
        (p_schema VARCHAR2, p_name VARCHAR2)
        RETURN VARCHAR2;
END orders_app_pkg;      -- package spec
/
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Step 2: Create a Package

In the OE schema, the ORDERS_APP_PKG is created. This package contains three routines:

- **show_app_context**: For learning and testing purposes, this procedure displays a context attribute and value.
- **set_app_context**: This procedure sets a context attribute to a specific value.
- **the_predicate**: This function builds the predicate (the WHERE clause) that controls the rows that are visible in the CUSTOMERS table to a user. (Note that this function requires two input parameters. An error occurs when the policy is implemented if you exclude these two parameters.)

```
CREATE OR REPLACE PACKAGE BODY orders_app_pkg
IS
    c_context CONSTANT VARCHAR2(30) := 'ORDER_CTX';
    c_attrib  CONSTANT VARCHAR2(30) := 'ACCOUNT_MGR';

    PROCEDURE show_app_context
    IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Type: ' || c_attrib ||
        ' - ' || SYS_CONTEXT(c_context, c_attrib));
    END show_app_context;

    PROCEDURE set_app_context
    IS
        v_user VARCHAR2(30);
    BEGIN
        SELECT user INTO v_user FROM dual;
        DBMS_SESSION.SET_CONTEXT
            (c_context, c_attrib, v_user);
    END set_app_context;

    FUNCTION the_predicate
    (p_schema VARCHAR2, p_name VARCHAR2)
    RETURN VARCHAR2
    IS
        v_context_value VARCHAR2(100) :=
            SYS_CONTEXT(c_context, c_attrib);
        v_restriction VARCHAR2(2000);
    BEGIN
        IF v_context_value LIKE 'AM%' THEN
            v_restriction :=
                'ACCOUNT_MGR_ID =
                SUBSTR('' '|| v_context_value || '''', 3, 3)';
        ELSE
            v_restriction := null;
        END IF;
        RETURN v_restriction;
    END the_predicate;

    END orders_app_pkg; -- package body
    /
```

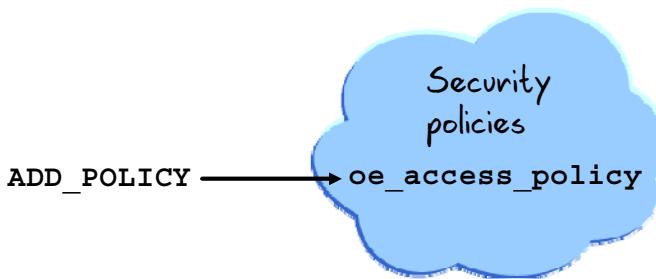
Note that the THE_PREDICATE function builds the WHERE clause and stores it in the V_RESTRICTION variable. If the SYS_CONTEXT function returns an attribute value that starts with AM, the WHERE clause is built with ACCOUNT_MGR_ID = *the last three characters of the attribute value*. If the user is AM145, the WHERE clause will be:

```
WHERE account_mgr_id = 145
```

Step 3: Defining the Policy

Use the DBMS_RLS package:

- It contains the fine-grained access administrative interface.
- It adds a fine-grained access control policy to a table or view.
- You use the ADD_POLICY and DROP_POLICY procedure to add and remove a fine-grained access control policy to a table or view.



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The DBMS_RLS package contains the fine-grained access control administrative interface. The package holds several procedures. But the package by itself does nothing until you add a policy. To add a policy, you use the ADD_POLICY procedure within the DBMS_RLS package.

Note: DBMS_RLS is available only with Oracle Database Enterprise Edition.

Step 3: Define the Policy

The DBMS_RLS.ADD_POLICY procedure adds a fine-grained access control policy to a table or view. The procedure causes the current transaction, if any, to commit before the operation is carried out. However, this does not cause a commit first if it is inside a DDL event trigger. The following are the parameters for the ADD_POLICY procedure:

```
DBMS_RLS.ADD_POLICY (
    object_schema    IN VARCHAR2 := NULL,
    object_name      IN VARCHAR2,
    policy_name      IN VARCHAR2,
    function_schema  IN VARCHAR2 := NULL,
    policy_function  IN VARCHAR2,
    statement_types  IN VARCHAR2 := NULL,
    update_check     IN BOOLEAN := FALSE,
    enable           IN BOOLEAN := TRUE);
```

Parameter	Description
OBJECT_SCHEMA	Schema containing the table or view (logon user, if NULL)
OBJECT_NAME	Name of the table or view to which the policy is added
POLICY_NAME	Name of the policy to be added. For any table or view, each POLICY_NAME must be unique.
FUNCTION_SCHEMA	Schema of the policy function (logon user, if NULL)
POLICY_FUNCTION	Name of the function that generates a predicate for the policy. If the function is defined within a package, the name of the package must be present.
STATEMENT_TYPES	Statement types that the policy will apply. It can be any combination of SELECT, INSERT, UPDATE, and DELETE. The default is to apply all these statement types to the policy.
UPDATE_CHECK	Optional argument for the INSERT or UPDATE statement types. The default is FALSE. Setting update_check to TRUE causes the server to also check the policy against the value after INSERT or UPDATE.
ENABLE	Indicates whether the policy is enabled when it is added. The default is TRUE.
SEC_RELEVANT_COLS SEC_RELEVANT_COLS_OPT	Enable getting all rows from the table. However, for the relevant columns users can see only their own data; in other columns they can see NULL values.

The following is a list of the procedures contained in the DBMS_RLS package. For detailed information, refer to the *PL/SQL Packages and Types Reference*.

Procedure	Description
ADD_POLICY	Adds a fine-grained access control policy to a table or view
DROP_POLICY	Drops a fine-grained access control policy from a table or view
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 has one less driving context
ENABLE_GROUPED_POLICY	Enables or disables a row-level group security policy
REFRESH_GROUPED_POLICY	Reparses the SQL statements associated with a refreshed policy

Step 3: Defining the Policy

```
DECLARE
BEGIN
  DBMS_RLS.ADD_POLICY (
    'OE',          ← Object schema
    'CUSTOMERS',   ← Table name
    'OE_ACCESS_POLICY', ← Policy name
    'OE',          ← Function schema
    'ORDERS_APP_PKG.THE_PREDICATE', ← Policy function
    'SELECT, UPDATE, DELETE',        ← Statement types
    FALSE,           ← Update check
    TRUE);           ← Enabled
END;
/
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The OE_ACCESS_POLICY security policy is created and added with the DBMS_RLS.ADD_POLICY procedure. The predicate function that defines how the policy is to be implemented is associated with the policy being added.

This example specifies that whenever a SELECT, an UPDATE, or a DELETE statement on the OE.CUSTOMERS table is executed, the predicate function return result is appended to the WHERE clause.

Step 4: Setting Up a Logon Trigger

Create a database trigger that executes whenever anyone logs on to the database:

```
CREATE OR REPLACE TRIGGER set_id_on_logon
AFTER logon on DATABASE
BEGIN
  oe.orders_app_pkg.set_app_context;
END;
/
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

After the context is created and the security package, the predicate, and the policy are defined, you create a logon trigger to implement fine-grained access control. This trigger causes the context to be set as each user is logged on.

Note: You require the ADMINISTER DATABASE TRIGGER to create this trigger.

Example Results

Data in the CUSTOMERS table:

```
SELECT COUNT(*), account_mgr_id
FROM customers
GROUP BY account_mgr_id;

COUNT(*)          ACCOUNT_MGR_ID
-----
76                147
74                149
58                148
111               145
```

```
CONNECT AM148/oracle
SELECT customer_id, cust_last_name
FROM oe.customers;

CUSTOMER_ID CUSTOMER_LAST_NAME
-----
...
58 rows selected.
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The AM148 user who logs on sees only those rows in the CUSTOMERS table that are defined by the predicate function. The user can issue SELECT, UPDATE, and DELETE statements against the CUSTOMERS table, but only the rows defined by the predicate function can be manipulated.

```
UPDATE oe.customers
SET credit_limit = credit_limit + 5000
WHERE customer_id = 101;

0 rows updated.
```

The AM148 user does not have access to customer ID 101. Customer ID 101 has the account manager 145. Any updates, deletes, or selects attempted by user AM148 on customers that do not have him or her as the account manager are not performed. It is as though these customers do not exist.

Data Dictionary Views

View	Description
USER_POLICIES	All policies owned by the current schema
ALL_POLICIES	All policies owned or accessible by the current schema
DBA_POLICIES	All policies in the database (Its columns are the same as those in ALL_POLICIES.)
ALL_CONTEXT	All active context namespaces defined in the session
DBA_CONTEXT	All context namespace information (active and inactive)



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You can query the data dictionary views to find information about the policies available in your schema.

Using the ALL_CONTEXT Dictionary View

Use ALL_CONTEXT to see the active context namespaces defined in your session:

```
CONNECT AS AM148

SELECT *
FROM   all_context;

NAMESPACE          SCHEMA          PACKAGE
-----
ORDER_CTX          OE              ORDERS_APP_PKG
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You can use the ALL_CONTEXT dictionary view to view information about the contexts to which you have access. In the slide, the NAMESPACE column is equivalent to the context name.

Using the ALL_CONTEXT Dictionary View

Use ALL_POLICIES to view information about the policies to which you have access:

```
SELECT object_name, policy_name, pf_owner,
       package, function, sel, ins, upd, del
  FROM ALL_POLICIES;
```

OBJECT_NAME	POLICY_NAME	PF_OWNER	PACKAGE	FUNCTION	SEL	INS	UPD	DEL
CUSTOMERS	OE_ACCESS_POLICY OE		ORDERS_APP_PKG	THE_PREDICATE	YES	NO	YES	YES

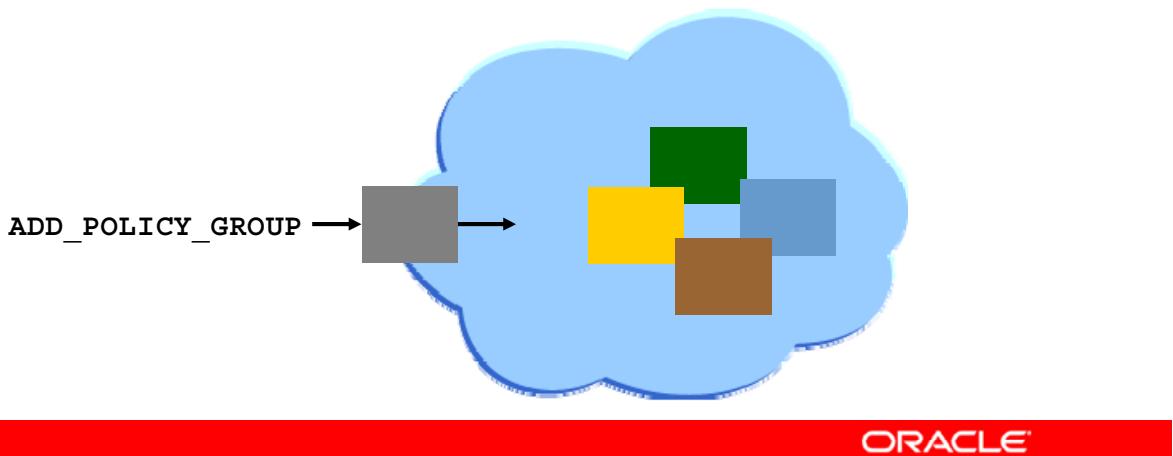


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You can use the ALL_POLICIES dictionary view to view information about the policies to which you have access. In the example in the slide, information is shown about the OE_ACCESS_POLICY policy.

Policy Groups

- Indicate a set of policies that belong to an application
- Are set up by a DBA through an application context called a driving context
- Use the `DBMS_RLS` package to administer the security policies



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Policy groups were introduced in Oracle9*i*, release 1 (9.0.1). The DBA designates an application context, called a driving context, to indicate the policy group in effect. When tables or views are accessed, the fine-grained access control engine looks up the driving context to determine the policy group in effect and enforces all associated policies that belong to that policy group.

The PL/SQL `DBMS_RLS` package enables you to administer your security policies and groups. Using this package, you can add, drop, enable, disable, and refresh the policy groups that you create.

More About Policies

- **SYS_DEFAULT** is the default policy group:
 - The **SYS_DEFAULT** group may or may not contain policies.
 - All policies belong to **SYS_DEFAULT** by default.
 - You cannot drop the **SYS_DEFAULT** policy group.
- Use `DBMS_RLS.CREATE_POLICY_GROUP` to create a new group.
- Use `DBMS_RLS.ADD_GROUPED_POLICY` to add a policy associated with a policy group.
- You can apply multiple driving contexts to the same table or view.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

A policy group is a set of security policies that belong to an application. You can designate an application context (known as a driving context) to indicate the policy group in effect. When the tables or views are accessed, the server looks up the driving context to determine the policy group in effect. It enforces all associated policies that belong to that policy group.

By default, all policies belong to the **SYS_DEFAULT** policy group. The policies defined in this group for a particular table or view are always executed along with the policy group specified by the driving context. The **SYS_DEFAULT** policy group may or may not contain policies. If you attempt to drop the **SYS_DEFAULT** policy group, an error is raised. If you add policies associated with two or more objects to the **SYS_DEFAULT** policy group, each such object has a separate **SYS_DEFAULT** policy group associated with it.

For example, the `CUSTOMERS` table in the `OE` schema has one **SYS_DEFAULT** policy group, and the `ORDERS` table in the `OE` schema has a different **SYS_DEFAULT** policy group associated with it.

```
SYS_DEFAULT
  - policy1 (OE/CUSTOMERS)
  - policy3 (OE/CUSTOMERS)
SYS_DEFAULT
  - policy2 (OE/ORDERS)
```

When adding a policy to a table or view, you can use the `DBMS_RLS.ADD_GROUPED_POLICY` interface to specify the group to which the policy belongs. To specify which policies are effective, you can add a driving context using the `DBMS_RLS.ADD_POLICY_CONTEXT` interface. If the driving context returns an unknown policy group, an error is returned.

If the driving context is not defined, all policies are executed. Likewise, if the driving context is `NULL`, the policies from all policy groups are enforced. Thus, an application that accesses the data cannot bypass the security setup module (that sets up the application context) to avoid applicable policies.

You can apply multiple driving contexts to the same table or view, and each of them is processed individually. Thus, you can configure multiple active sets of policies to be enforced.

You can create a new policy by using the `DBMS_RLS` package either from the command line or programmatically, or you can access the Oracle Policy Manager graphical user interface in Oracle Enterprise Manager.

Quiz

Which of the following statements is *not* true about fine-grained access control?

- a. Fine-grained access control enables you to enforce security through a low level of granularity.
- b. Fine-grained access control restricts users to viewing only “their” information.
- c. Fine-grained access control is implemented through a security policy attached to tables.
- d. To implement fine-grained access control, hard code the security policy into the user code.



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Answer: d

Quiz

Application context for a user is fixed and does not change with change of session.

- a. True
- b. False



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Answer: b

Quiz

Arrange the following steps used to implement a security policy:

- A. Define the policy.
 - B. Create the package associated with the context.
 - C. Set up a driving context.
 - D. Set up a logon trigger to call the package at logon time and set the context.
- a. A, B, C D
 - b. C, A, B, D
 - c. B, A, C, D
 - d. D, A, B, C



ORACLE®

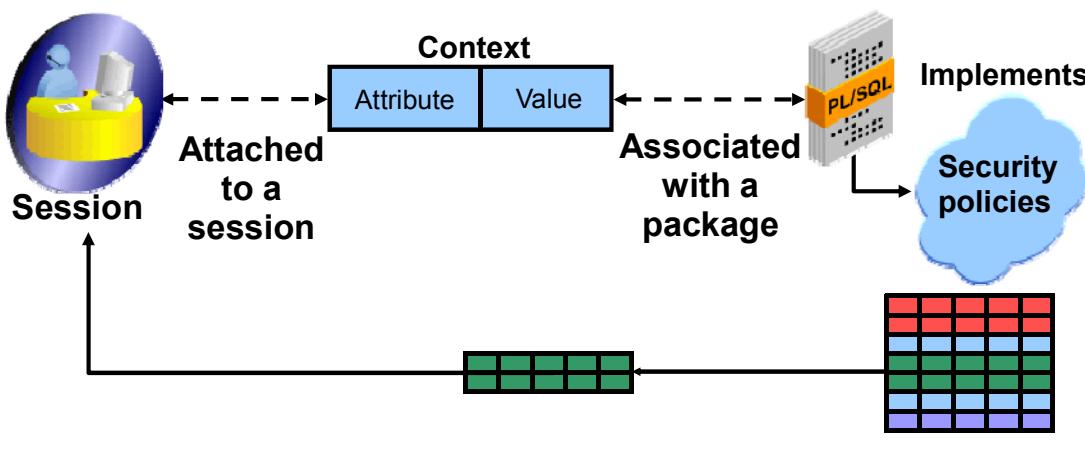
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Answer: b

Summary

In this lesson, you should have learned how to:

- Describe the process of fine-grained access control
- Implement and test fine-grained access control



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

ORACLE

In this lesson, you should have learned about fine-grained access control and the steps required to implement a virtual private database.

Practice 12: Overview

This practice covers the following topics:

- Creating an application context
- Creating a policy
- Creating a logon trigger
- Implementing a virtual private database
- Testing the virtual private database



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In this practice, you implement and test fine-grained access control.

13

Safeguarding Your Code Against SQL Injection Attacks

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Objectives

After completing this lesson, you should be able to do the following:

- Describe SQL injection
- Reduce attack surfaces
- Use DBMS_ASSERT



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In this lesson, you learn how to use the techniques and tools to strengthen your code and applications against SQL injection attacks.

Lesson Agenda

- Understanding SQL injection
- Reducing the attack surface
- Avoiding dynamic SQL
- Using bind arguments
- Filtering input with DBMS_ASSERT



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Understanding SQL Injection

SQL injection is a technique for maliciously exploiting applications that use client-supplied data in SQL statements.

- Attackers trick the SQL engine into executing unintended commands.
- SQL injection techniques may differ, but they all exploit a single vulnerability in the application.
- To immunize your code against SQL injection attacks, use bind arguments or validate and sanitize all input concatenated to dynamic SQL.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

SQL injection is a technique for maliciously exploiting applications that use client-supplied data in SQL statements.

Attackers trick the SQL engine into executing unintended commands by supplying specially crafted string input, thereby gaining unauthorized access to a database in order to view or manipulate restricted data.

SQL injection techniques may differ, but they all exploit a single vulnerability in the application.

String literals that are incorrectly validated or not validated are concatenated into a dynamic SQL statement and interpreted as code by the SQL engine.

To immunize your code against SQL injection attacks, you must use bind arguments (either automatically with static SQL or explicitly with dynamic SQL), or validate and sanitize all input concatenated to dynamic SQL.

Although a program or an application may be vulnerable to SQL injection, web applications are at a higher risk, because an attacker can perpetrate SQL injection attacks without any database or application authentication.

Identifying Types of SQL Injection Attacks

Category	Description
First-order attack	The attacker can simply enter a malicious string and cause the modified code to be executed immediately.
Second-order attack	The attacker injects into persistent storage (such as a table row), which is deemed a trusted source. An attack is subsequently executed by another activity.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The impact of SQL injection attacks may vary from gathering of sensitive data to manipulating database information, and from executing system-level commands to denial of service of the application. The impact also depends on the database on the target machine, and the roles and privileges that the SQL statement is running with.

Researchers generally divide injection attacks into two categories:

- First-order attack
- Second-order attack

SQL injection attacks do not have to return data directly to the user to be useful. “Blind” attacks (for example, that create a database user, but otherwise return no data) can still be very useful to an attacker.

In addition, hackers are known to use timing or other performance indicators, and even error messages in order to deduce the success or results of an attack.

SQL Injection: Example

```
-- First order attack
CREATE OR REPLACE PROCEDURE GET_EMAIL
  (p_last_name VARCHAR2 DEFAULT NULL)
AS
  TYPE      cv_custtyp IS REF CURSOR;
  cv        cv_custtyp;
  v_email   customers.cust_email%TYPE;
  v_stmt    VARCHAR2(400);
BEGIN
  v_stmt := 'SELECT cust_email FROM customers
            WHERE cust_last_name = ''' || p_last_name || '''';
  DBMS_OUTPUT.PUT_LINE('SQL statement: ' || v_stmt);
  OPEN cv FOR v_stmt;
  LOOP
    FETCH cv INTO v_email;
    EXIT WHEN cv%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('Email: ' || v_email);
  END LOOP;
  CLOSE cv;
EXCEPTION WHEN OTHERS THEN
  dbms_output.PUT_LINE(sqlerrm);
  dbms_output.PUT_LINE('SQL statement: ' || v_stmt);
END;
```

String literals that are incorrectly validated or not validated are concatenated into a dynamic SQL statement, and interpreted as code by the SQL engine.




Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

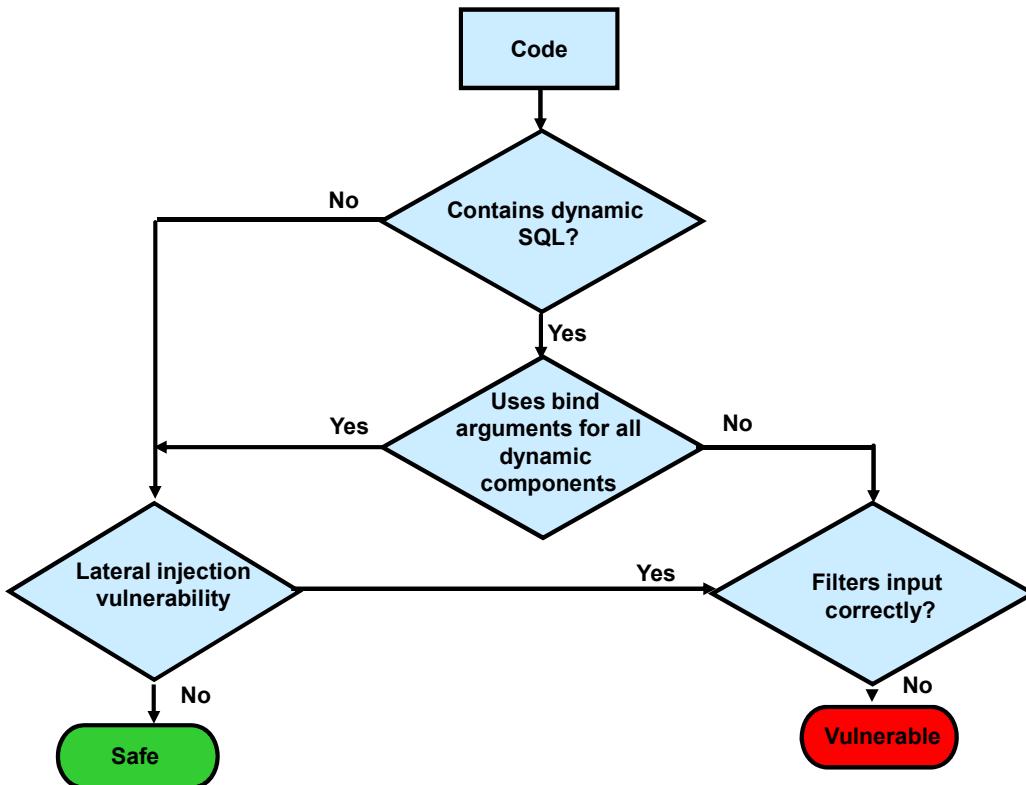
The example shown in the slide demonstrates a procedure with dynamic SQL constructed via concatenation of input value. This is vulnerable to SQL injection.

```
EXECUTE get_email('Andrews')
SQL statement: SELECT cust_email FROM customers WHERE
cust_last_name = 'Andrews'
Email: Ajay.Andrews@YELLOWTHROAT.COM
Email: Dianne.Andrews@TURNSTONE.COM

PL/SQL procedure successfully completed.
```

```
EXECUTE get_email('x' union select username from all_users
where ''x''='x')
SQL statement: SELECT cust_email FROM customers WHERE
cust_last_name = 'x' union
select username from all_users where 'x'='x'
Email: ANONYMOUS
Email: APEX_PUBLIC_USER
Email: BI
Email: CTXSYS
...
```

Assessing Vulnerability



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You must analyze your code to determine SQL injection vulnerability. The flowchart in the slide shows you how to start assessing for vulnerability.

Avoidance Strategies Against SQL Injection

Strategy	Description
Reduce the attack surface	Ensure that all excess database privileges are revoked and that only those routines that are intended for end-user access are exposed. Though this does not entirely eliminate SQL injection vulnerabilities, it does mitigate the impact of the attacks.
Avoid dynamic SQL with concatenated input	Dynamic SQL built with concatenated input values presents the easiest entry point for SQL injections. Avoid constructing dynamic SQL this way.
Use bind arguments	Parameterize queries using bind arguments. Not only do bind arguments eliminate the possibility of SQL injections, they also enhance performance.
Filter and sanitize input	The Oracle-supplied DBMS_ASSERT package contains a number of functions that can be used to sanitize user input and to guard against SQL injection in applications that use dynamic SQL built with concatenated input values. If your filtering requirements cannot be satisfied by the DBMS_ASSERT package, create your own filter.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You can use several avoidance strategies to safeguard against, or mitigate the impact of, SQL injection attacks. The slide lists high-level descriptions of each of the strategies that are examined in more detail on subsequent pages.

The available and best methods for eliminating SQL injection vulnerability may depend on the vulnerability itself. Not all methods are available for addressing every vulnerability.

Methods

- Use static SQL, if all Oracle identifiers (for example, column, table, view, trigger, program unit, or schema names) are known at code compilation time.

Note: Static SQL automatically binds arguments. Data definition language (DDL) statements cannot be executed with static SQL.
- Use dynamic SQL with bind arguments, if any WHERE clause values, VALUES clause values, or SET clause values are unknown, and any Oracle identifiers are unknown at code compilation time.

Note: Use bind arguments for the values (the literals). Use string concatenation for the validated and sanitized Oracle identifiers.
- Validate and sanitize input, if concatenating any strings and if bind arguments require additional filtering.

Protecting Against SQL Injection: Example

```
CREATE OR REPLACE PROCEDURE GET_EMAIL
  (p_last_name VARCHAR2 DEFAULT NULL)
AS
BEGIN
  FOR i IN
    (SELECT cust_email
     FROM customers
     WHERE cust_last_name = p_last_name)
  LOOP
    DBMS_OUTPUT.PUT_LINE('Email: '||i.cust_email);
  END LOOP;
END;
```

This example avoids dynamic SQL
with concatenated input values.

```
EXECUTE get_email('Andrews')
Email: Ajay.Andrews@YELLOWTHROAT.COM
Email: Dianne.Andrews@TURNSTONE.COM

PL/SQL procedure successfully completed.

EXECUTE get_email('x'' union select username from all_users where
''x'''='x')

PL/SQL procedure successfully completed.
```

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The example in the slide with static SQL is protected against SQL injection.

Lesson Agenda

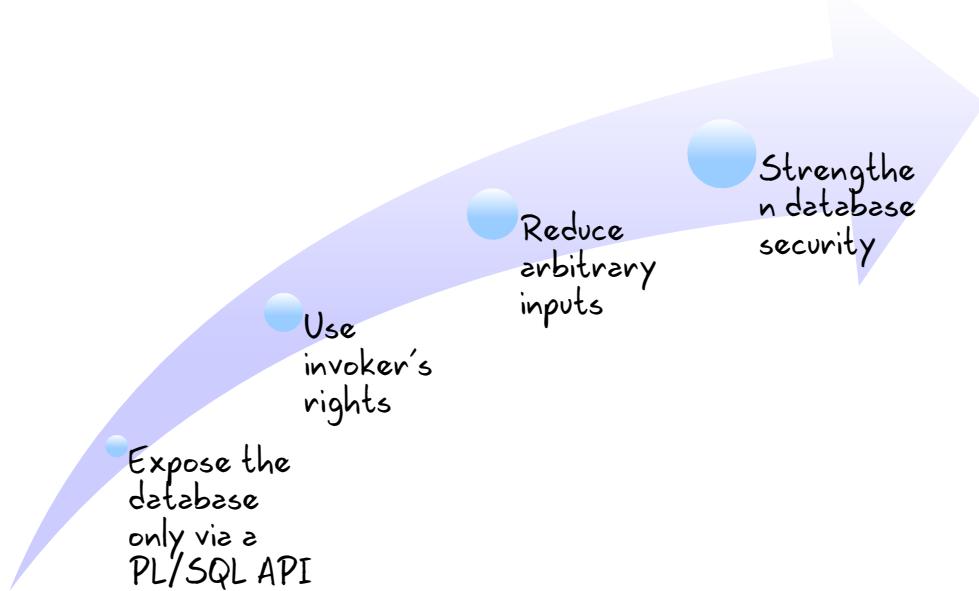
- Understanding SQL injection
- Reducing the attack surface
- Avoiding dynamic SQL
- Using bind arguments
- Filtering input with DBMS_ASSERT



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Reducing the Attack Surface

Use the following strategies to reduce the attack surface:



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

If an interface is not available to an attacker, it is clearly not available to be abused. Thus the first, and arguably the most important, line of defense is to reduce the exposed interfaces to only those absolutely required.

Expose the Database Only Via PL/SQL API

- Expose the database to clients only via a PL/SQL API.
- When you design a PL/SQL package that accesses the database, use the following paradigm:

- Establish a database user as the *only* one to which a client may connect. Hypothetically, let us call this user `myuser`.
 - `myuser` may own *only* synonyms and these synonyms may denote *only* PL/SQL units owned by other users
 - Grant the Execute privilege on *only* the denoted PL/SQL units to `myuser`



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Expose the database to clients only via a PL/SQL API. Carefully control privileges so that the client has no direct access to the application's other kinds of objects, especially tables and views.

Using Invoker's Rights

- Using invoker's rights help to:
 - Limit the privileges
 - Minimize the security exposure
- The following example does not use invoker's rights:

```
CREATE OR REPLACE
PROCEDURE change_password(p_username VARCHAR2 DEFAULT NULL,
                           p_new_password VARCHAR2 DEFAULT NULL)
IS
  v_sql_stmt VARCHAR2(500);
BEGIN
  v_sql_stmt := 'ALTER USER '||p_username||' IDENTIFIED BY '
                || p_new_password;
  EXECUTE IMMEDIATE v_sql_stmt;
END change_password;
```

Note the use of dynamic SQL with concatenated input values.

```
GRANT EXECUTE ON change_password TO OE, HR, SH;
```

1

2



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Stored program units and SQL methods execute with a set of privileges. By default, the privileges are those of a schema owner, also known as the definer. The definer's rights not only dictate the privileges, they are also used to resolve object references. If a program unit does not need to be executed with the escalated privileges of the definer, you should specify that the program unit executes with the privileges of a caller, also known as the invoker.

1. The example shown in the slide uses definer's rights. The `CHANGE_PASSWORD` procedure is created under the `SYS` schema. It accepts two parameters and uses them in the `ALTER USER` statement.
2. `SYS` grants `OE`, `HR`, and `SH` the ability to execute the `CHANGE_PASSWORD` procedure.

Using Invoker's Rights

- OE is successful at changing the SYS password, because, by default, CHANGE_PASSWORD executes with SYS privileges:

```
EXECUTE sys.change_password ('SYS', 'mine')
```

- Add the AUTHID to change the privileges to the invokers:

```
CREATE OR REPLACE
PROCEDURE change_password(p_username VARCHAR2 DEFAULT NULL,
                           p_new_password VARCHAR2 DEFAULT NULL)
AUTHID CURRENT_USER
IS
  v_sql_stmt VARCHAR2(500);
BEGIN
  v_sql_stmt := 'ALTER USER '||p_username||' IDENTIFIED BY '
               || p_new_password;
  EXECUTE IMMEDIATE v_sql_stmt;
END change_password;
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

When OE executes the CHANGE_PASSWORD procedure, it is executed under SYS privileges (the definer of the procedure), and with the code shown in the slide, OE can change the SYS password. Obviously, this is an unacceptable outcome.

To prevent another schema from changing a password that does not belong to the schema, redefine the procedure with the invoker's rights. This is done with the AUTHID CURRENT_USER option.

```
EXECUTE change_password ('SYS', 'mine')
ERROR at line 1:
ORA-01031: Insufficient privileges
ORA-06512: at "SYS.CHANGE_PASSWORD", at line 1
ORA-06512: at line 1
```

Now OE can no longer change the SYS (or any other account) password.

Notice that the CHANGE_PASSWORD procedure contains dynamic SQL with concatenated input values. This is a SQL injection vulnerability. Although using invoker's rights does not guarantee the elimination of SQL injection risks, it can help mitigate the exposure.

Reducing Arbitrary Inputs

1

Reduce the end-user interfaces to only those that are actually needed.

2

Where user input is required, make use of language features to ensure that only data of the intended type can be specified.

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Because a SQL injection attack is possible only if user input is allowed, one key measure of prevention is to limit user input.

First, you must reduce the end user's access to only those interfaces that are actually needed. For example:

- In a web application, restrict users to accessing specified webpages.
- In a PL/SQL API, expose only those routines that are intended for customer use.

Remove any debug, test, deprecated, or other unnecessary interfaces. They add nothing to product functionality, but do provide an attacker with more ways to target your application.

From Oracle Database 10.2 and later, you can use PL/SQL conditional compilation for managing self-tracing code, asserts, and so on.

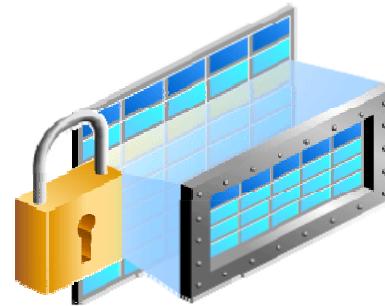
Then, where user input is required, make use of language features to ensure that only data of the intended type can be specified. For example:

- Do not specify a VARCHAR2 parameter when it will be used as a number.
- Do not use numbers if you need only positive integers; use natural instead.

Careful selection of parameter types to an API can considerably reduce the scope for attack, and make it a lot easier for customers to use.

Strengthen Database Security

- Encrypt sensitive data so that it cannot be viewed.
- Avoid:
 - Using PUBLIC privileges
 - Using EXECUTE ANY PROCEDURE privilege
 - Granting privileges the WITH ADMIN option
- Do not allow wide access to any standard Oracle packages that can operate on the operating system.
- Enforce password management.



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Oracle Database contains inherent security features that help in protecting it from many types of attacks, including SQL injection. The following is a list of some of the practices to observe when you secure the Oracle database:

- Encrypt sensitive data so that it cannot be viewed.
- Evaluate all PUBLIC privileges and revoke them where possible.
- Do not widely grant EXECUTE ANY PROCEDURE.
- Avoid granting privileges the WITH ADMIN option.
- Do not allow wide access to any standard Oracle packages that can operate on the operating system. These packages include UTL_HTTP, UTL_SMTP, UTL_TCP, DBMS_PIPE, UTL_MAIL, and UTL_FTP.
- Certain Oracle packages, such as UTL_FILE and DBMS_LOB, are governed by the privilege model of the Oracle DIRECTORY object. Protect Oracle DIRECTORY objects.
- Lock the database default accounts and expire the default passwords.
- Remove example scripts and programs from the Oracle directory.
- Run the database listener as a nonprivileged user.
- Apply basic password management rules, such as password length, history, and complexity, to all user passwords. Mandate that all users change their passwords regularly.
- Lock and expire the default user accounts and change the default user password.

Lesson Agenda

- Understanding SQL injection
- Reducing the attack surface
- **Avoiding dynamic SQL**
- Using bind arguments
- Filtering input with DBMS_ASSERT



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Using Static SQL

- Eliminates SQL injection vulnerability
- Creates schema object dependencies upon successful compilation
- Can improve performance, when compared with DBMS_SQL



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Because SQL injection is a feature of SQL statements that are dynamically constructed via user inputs, it follows that designing your application to be based on static SQL reduces the scope for attack.

There are two common situations where developers often use dynamic SQL, when static SQL would serve the purpose and be more secure:

- Handling a varying number of IN-list values in the query condition
- Handling the LIKE comparison operator in the query condition

Using Static SQL

```

CREATE OR REPLACE PROCEDURE list_products_dynamic
(p_product_name VARCHAR2 DEFAULT NULL)
AS
  TYPE cv_prodtyp IS REF CURSOR;
  cv_cv_prodtyp;
  v_prodname product_information.product_name%TYPE; You can convert this
  v_minprice product_information.min_price%TYPE; statement to static SQL.
  v_listprice product_information.list_price%TYPE;
  v_stmt VARCHAR2(400);
BEGIN
  v_stmt := 'SELECT product_name, min_price, list_price
            FROM product_information WHERE product_name LIKE
            ''%'||p_product_name||'%''';
  OPEN cv FOR v_stmt;
  dbms_output.put_line(v_stmt);
  LOOP
    FETCH cv INTO v_prodname, v_minprice, v_listprice;
    EXIT WHEN cv%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('Product Info: '||v_prodname ||', ' ||
                         v_minprice ||', '|| v_listprice);
  END LOOP;
  CLOSE cv;
END;

```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

By converting the dynamic static SQL shown in the slide, you can decrease your attack vulnerability. The code in the slide uses dynamic SQL to handle the LIKE operator in the query condition. Notice that string concatenation is used to build the SQL statement.

Examine the execution of this injection:

```

EXECUTE list_products_dynamic(''' and 1=0 union select
cast(username as varchar2(100)), null, null
from all_users --')

SELECT product_name, min_price, list_price FROM
product_information WHERE
product_name like '%' and 1=0 union select cast(username as
varchar2(100)),
null, null from all_users --%
Product Info: ANONYMOUS, ,
Product Info: APEX_PUBLIC_USER, ,
Product Info: BI, ,
...

```

Notice that the injection succeeded through the concatenation of the UNION set operator to the dynamic SQL statement.

Using Static SQL

- To use static SQL, accept the user input, and then concatenate the necessary string to a local variable.
- Pass the local variable to the static SQL statement.

```
CREATE OR REPLACE PROCEDURE list_products_static
  (p_product_name VARCHAR2 DEFAULT NULL)
AS
  v_bind  VARCHAR2(400);
BEGIN
  v_bind := '%' || p_product_name || '%';
  FOR i IN
    (SELECT product_name, min_price, list_price
     FROM product_information
      WHERE product_name like v_bind)
  LOOP
    DBMS_OUTPUT.PUT_LINE('Product Info: ' || i.product_name || ','
      || i.min_price || ', ' || i.list_price);
  END LOOP;
END list_products_static;
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In the example shown in the slide, you can avoid SQL injection by concatenating the user input and placing it into a local variable, and then using the local variable within the static SQL.

Examine the results:

```
-- desired results - normal execution
EXECUTE list_products_static('Laptop')
Product Info: Laptop 128/12/56/v90/110, 2606, 3219
Product Info: Laptop 16/8/110, 800, 999
Product Info: Laptop 32/10/56, 1542, 1749
Product Info: Laptop 48/10/56/110, 2073, 2556
Product Info: Laptop 64/10/56/220, 2275, 2768

anonymous block completed

-- this example attempts injection
EXECUTE list_products_static(''' and 1=0 union select
cast(username as varchar2(100)), null, null from all_users --
''')

anonymous block completed
```

Using Dynamic SQL

- Dynamic SQL may be unavoidable in the following types of situations:
 - You do not know the full text of the SQL statements that must be executed in a PL/SQL procedure.
 - You want to execute DDL statements and other SQL statements that are not supported in purely static SQL programs.
 - You want to write a program that can handle changes in data definitions without the need to recompile.
- If you must use dynamic SQL, try not to construct it through concatenation of input values. Instead, use bind arguments.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

- Dynamic SQL may be unavoidable in the following types of situations:
 - You do not know the full text of the SQL statements that must be executed in a PL/SQL procedure—for example, a `SELECT` statement that includes an identifier (such as table name) that is unknown at compile time or a `WHERE` clause in which the number of subclauses is unknown at compile time.
 - You want to execute DDL statements and other SQL statements that are not supported in purely static SQL programs.
 - You want to write a program that can handle changes in data definitions without the need to recompile.
- If you must use dynamic SQL, try not to construct it through concatenation of input values. Instead, use bind arguments.

If you cannot avoid input concatenation, you must validate the input values, and also consider constraining user input to a predefined list of values, preferably numeric values. Input filtering and sanitizing are covered in more detail later in this lesson.

Lesson Agenda

- Understanding SQL injection
- Reducing the attack surface
- Avoiding dynamic SQL
- Using bind arguments
- Filtering input with DBMS_ASSERT



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Using Bind Arguments with Dynamic SQL

You can rewrite the following statement

```
v_stmt :=  
'SELECT '||filter(p_column_list)||' FROM customers'||  
'WHERE account_mgr_id = '''|| p_sales_rep_id ||'''';  
  
EXECUTE IMMEDIATE v_stmt;
```

as the following dynamic SQL with a placeholder (:1) by using a bind argument (p_sales_rep_id):

```
v_stmt :=  
'SELECT '||filter(p_column_list)||' FROM customers'||  
'WHERE account_mgr_id = :1';  
  
EXECUTE IMMEDIATE v_stmt USING p_sales_rep_id;
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You can use bind arguments in the WHERE clause, the VALUES clause, or the SET clause of any SQL statement, as long as the bind arguments are not used as Oracle identifiers (such as column names or table names) or keywords.

Developers often use dynamic SQL to handle a varying number of IN-list values or LIKE comparison operators in the query condition.

Using Bind Arguments with Dynamic PL/SQL

If you must use dynamic PL/SQL, try to use bind arguments. For example, you can rewrite the following dynamic PL/SQL with concatenated string values

```
v_stmt :=  
  'BEGIN  
    get_phone ('' || p_fname ||  
               '''', '' || p_lname || '')'; END;';  
  
EXECUTE IMMEDIATE v_stmt;
```

as the following dynamic PL/SQL with placeholders (:1, :2) by using bind arguments (p_fname, p_lname):

```
v_stmt :=  
  'BEGIN  
    get_phone(:1, :2); END;';  
  
EXECUTE IMMEDIATE v_stmt USING p_fname, p_lname;
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

As with dynamic SQL, you should avoid constructing dynamic PL/SQL with string concatenation. The impact of SQL injection vulnerabilities in dynamic PL/SQL is more serious than in dynamic SQL, because, with dynamic PL/SQL, multiple statements (such as DELETE or DROP) can be batched together and injected.

What if You Cannot Use Bind Arguments?

- Bind arguments cannot be used with:
 - DDL statements
 - Oracle identifiers
- If bind arguments cannot be used with the dynamic SQL or PL/SQL, you must filter and sanitize all input concatenated to the dynamic statement.



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Although you should strive to use bind arguments with all dynamic SQL and PL/SQL statements, there are instances where bind arguments cannot be used:

- DDL statements (such as CREATE, DROP, and ALTER)
- Oracle identifiers (such as names of columns, tables, schemas, database links, packages, procedures, and functions)

If bind arguments cannot be used with the dynamic SQL or PL/SQL, you must filter and sanitize all input concatenated to the dynamic statement. In the following slides, you learn how to use the Oracle-supplied DBMS_ASSERT package functions to filter and sanitize input values.

Lesson Agenda

- Understanding SQL injection
- Reducing the attack surface
- Avoiding dynamic SQL
- Using bind arguments
- Filtering input with DBMS_ASSERT



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Understanding DBMS_ASSERT

DBMS_ASSERT functions:

Function	Description
ENQUOTE_LITERAL	Encloses string literal in single quotes
SIMPLE_SQL_NAME	Verifies that the string is a simple SQL name



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

To guard against SQL injection in applications that do not use bind arguments with dynamic SQL, you must filter and sanitize concatenated strings. The primary use case for dynamic SQL with string concatenation is when an Oracle identifier (such as a table name) is unknown at code compilation time.

DBMS_ASSERT is an Oracle-supplied PL/SQL package containing seven functions that can be used to filter and sanitize input strings, particularly those that are meant to be used as Oracle identifiers.

When using the DBMS_ASSERT package, always specify the SYS schema rather than relying on a public synonym.

Formatting Oracle Identifiers

- Example 1: The object name used as an identifier:

```
SELECT count(*) records FROM orders;
```

- Example 2: The object name used as a literal:

```
SELECT num_rows FROM user_tables  
WHERE table_name = 'ORDERS';
```

- Example 3: The object name used as a quoted (normal format) identifier:

- The "orders" table referenced in example 3 is a different table compared to the `orders` table in examples 1 and 2.
 - It is vulnerable to SQL injection.

```
SELECT count(*) records FROM "orders";
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

To use `DBMS_ASSERT` effectively, you must understand how Oracle identifiers can be specified and used.

In a SQL statement, you specify the name of an object with an unquoted or quoted identifier.

- An unquoted (or internal format) identifier is not surrounded by punctuation. It must begin with a letter, and may be followed by letters, numbers, or a small set of special characters. This is how identifiers are most often specified, and how object names are stored in data dictionary tables.
- A quoted (or normal format) identifier begins and ends with double quotation marks. The identifier can include almost any character within the double quotes. This format is user supplied.

SQL injection attacks can use the quoted method to attempt to subvert code that is written to expect only the unquoted, more common, method.

Working with Identifiers in Dynamic SQL

For your identifiers, determine:

Where will the input come from: user or data dictionary?

What verification is required?

How will the result be used, as an identifier or a literal value?

These three factors affect:

What preprocessing is required (if any) prior to calling the verification functions

Which DBMS_ASSERT verification function is required

What post-processing is required before the identifier can actually be used



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

When working with identifiers in dynamic SQL statements, you must first determine where the data is coming from, either user data or data dictionary data. For verification, you must determine whether the object must exist and the type of identifier. For types of identifiers, you have:

- **SQL Literal:** A SQL literal is a constant value, written at compile time and is read-only at run time. There are three kinds of SQL literal: text, datetime, and numeric.
- **Simple:** A simple SQL name is a string that conforms to the following basic characteristics:
 - The first character of the name is alphabetic.
 - The name contains only alphanumeric characters or the _, \$, and # characters.
 - Quoted names must be enclosed within double quotation marks and may contain any characters, including quotation marks, provided they are represented by two quotation marks in a row.
- **Qualified SQL name:** A qualified SQL name is one or more simple SQL names that may be followed by a database link.

Finally, determine how the result will be used, as an identifier or a literal value.

Answering these questions will impact your preprocessing, post-processing, and the use of DBMS_ASSERT.

Choosing a Verification Route

Identifier Type	Verification
SQL literal	Verify whether the literal is a well-formed SQL literal by using DBMS_ASSERT.ENQUOTE_LITERAL.
Simple SQL name	Verify that the input string conforms to the basic characteristics of a simple SQL name by using DBMS_ASSERT.SIMPLE_SQL_NAME.
Qualified SQL name	Step 1: Decompose the qualified SQL name into its simple SQL names by using DBMS.Utility.Name_Tokenize(). Step 2: Verify each of the simple SQL names by using DBMS_ASSERT.SIMPLE_SQL_NAME.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

After determining the type of identifier that you must verify, follow the guidelines in the slide's table to select the appropriate verification routine.

For example, you must write a procedure that allows a user to change his or her password. To choose the correct DBMS_ASSERT function to use, you must determine:

- Where the input will come from, the user or the data dictionary
- What verification is required (Does the object need to exist and if so, what type of identifier is it?)
- How the identifier will be used, as an identifier or as a literal

Avoiding Injection by Using DBMS_ASSERT.ENQUOTE_LITERAL

```

CREATE OR REPLACE PROCEDURE Count_Rows(w in varchar2)
authid definer as
Quote constant varchar2(1) := '"';
Quote_Quote constant varchar2(2) := Quote||Quote;
Safe_Literal varchar2(32767) :=
    Quote||replace(w,Quote,Quote_Quote)||Quote;
Stmt constant varchar2(32767) :=
    'SELECT count(*) FROM t WHERE a=' || 
    DBMS_ASSERT.ENQUOTE_LITERAL(Safe_Literal);
Row_Count number;
BEGIN
    EXECUTE IMMEDIATE Stmt INTO Row_Count;
    DBMS_OUTPUT.PUT_LINE(Row_Count||' rows');
END;
/

```

Verify whether the literal is well-formed.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The Complete Code

```

DROP USER testuser CASCADE;
DROP USER eviluser CASCADE;
--Create the testuser and the eviluser
GRANT Unlimited Tablespace, Create Session, Create Table,
   Create Procedure TO testuser identified by testuser;
GRANT Unlimited Tablespace, Create Session, Create Table,
   Create Procedure TO eviluser identified by eviluser;
--Connect to the testuser, create a table and insert some
data
CONNECT testuser/testuser
SET SERVEROUTPUT ON
CREATE TABLE t(a varchar2(10));
BEGIN
    INSERT INTO t (a) values ('a');
    INSERT INTO t (a) values ('b');
    INSERT INTO t (a) values ('b');
    INSERT INTO t (a) values ('c''d');
    commit;
END;

```

```
--Create a procedure that uses dynamic SQL to query table 't'
CREATE OR REPLACE PROCEDURE Count_Rows(w in varchar2)
authid definer as
-- Useful constant
Quote constant varchar2(1) := '''';
-- The statement we will execute
Stmt constant varchar2(32767) :=
    'SELECT count(*) FROM t WHERE a=' || 
        Quote||w||Quote;
-- The count of rows returned by the statement
Row_Count number;
BEGIN
    EXECUTE IMMEDIATE Stmt INTO Row_Count;
    DBMS_OUTPUT.PUT_LINE('The Statement is: '||Stmt);
    DBMS_OUTPUT.PUT_LINE(Row_Count||' rows');
END;
/
GRANT EXECUTE ON Count_Rows TO PUBLIC;

--Connect to the eviluser and exploit the vulnerable procedure
connect eviluser/eviluser
SET SERVEROUTPUT ON
CREATE OR REPLACE FUNCTION f RETURN varchar2 authid
current_user as pragma autonomous_transaction;
BEGIN
    -- Execute immediate because t is not granted to public
    EXECUTE IMMEDIATE 'DELETE FROM t';
    COMMIT;
    RETURN 'a';
END;
/
GRANT EXECUTE ON f TO PUBLIC;
--Injection successful
BEGIN testuser.Count_Rows('a'' and eviluser.f=''a'); END;
/
--Re-code testuser.Count_Rows with DBMS_ASSERT.ENQUOTE_LITERAL

connect testuser/testuser
SELECT * FROM t;
BEGIN
    INSERT INTO t (a) values ('a');
    INSERT INTO t (a) values ('b');
    INSERT INTO t (a) values ('b');
    INSERT INTO t (a) values ('c''d');
    commit;
END;
```

```
CREATE OR REPLACE PROCEDURE Count_Rows(w in varchar2)
authid definer as

    -- Useful constants
    Quote      constant varchar2(1) := '"';
    Quote_Que  constant varchar2(2) := 
                    Quote||Quote;
    Safe_Literal      varchar2(32767) := 
                    Quote||replace(w,Quote,Quote_Que)||Quote;

    -- The statement we will execute
    Stmt  constant varchar2(32767) := 
        'SELECT count(*) FROM t WHERE a=' || 
        DBMS_ASSERT.ENQUOTE_LITERAL(Safe_Literal);

    -- The count of rows returned by the statement
    Row_Count  number;
BEGIN
    EXECUTE IMMEDIATE Stmt INTO Row_Count;
    DBMS_OUTPUT.PUT_LINE('The Statement is: '||Stmt);
    DBMS_OUTPUT.PUT_LINE(Row_Count||' rows');
END;
/

-- Now lets repeat the attack

connect eviluser/eviluser
SET SERVEROUTPUT ON
BEGIN testuser.Count_Rows('a'' and eviluser.f=''a'); END;
/
BEGIN testuser.Count_Rows('b'); END;
/
--Injection averted

connect testuser/testuser

set serveroutput on

SELECT * FROM t;
```

Avoiding Injection by Using DBMS_ASSERT.SIMPLE_SQL_NAME

```

CREATE OR REPLACE PROCEDURE show_col2 (p_colname varchar2,
p tablename  varchar2)
AS
type t is varray(200) of varchar2(25);
Results t;
Stmt CONSTANT VARCHAR2(4000) :=  

  'SELECT '||dbms_assert.simple_sql_name( p_colname ) || ' FROM  

' || dbms_assert.simple_sql_name( p_tablename ) ;  

BEGIN
  DBMS_Output.Put_Line ('SQL Stmt: ' || Stmt);
  EXECUTE IMMEDIATE Stmt bulk collect into Results;
  for j in 1..Results.Count() loop
    DBMS_Output.Put_Line(Results(j));
  end loop;
END show_col2;

```

Verify that the input string conforms to the basic characteristics of a simple SQL name.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The Complete Code

```

CONN hr/hr
SET SERVEROUTPUT ON
CREATE OR REPLACE
PROCEDURE show_col (p_colname varchar2, p_tablename
varchar2)
AS
type t is varray(200) of varchar2(25);
Results t;
Stmt CONSTANT VARCHAR2(4000) :=  

  'SELECT '|| p_colname || ' FROM '|| p_tablename ;
BEGIN
  DBMS_Output.Put_Line ('SQL Stmt: ' || Stmt);
  EXECUTE IMMEDIATE Stmt bulk collect into Results;
  for j in 1..Results.Count() loop
    DBMS_Output.Put_Line(Results(j));
  end loop;
  --EXCEPTION WHEN OTHERS THEN
  --Raise_Application_Error(-20000, 'Wrong table name');
END show_col;
/

```

```
execute show_col('Email','EMPLOYEES');
execute show_col('Email','EMP');
execute show_col('Email','EMPLOYEES where 1=2 union select
Username c1 from All_Users --');

CREATE OR REPLACE
PROCEDURE show_col2 (p_colname varchar2, p tablename
varchar2)
AS
type t is varray(200) of varchar2(25);
Results t;
Stmt CONSTANT VARCHAR2(4000) :=
  'SELECT '||dbms_assert.simple_sql_name( p_colname ) || '
FROM '|| dbms_assert.simple_sql_name( p tablename ) ;

BEGIN
  DBMS_Output.Put_Line ('SQL Stmt: ' || Stmt);
  EXECUTE IMMEDIATE Stmt bulk collect into Results;
for j in 1..Results.Count() loop
  DBMS_Output.Put_Line(Results(j));
end loop;
--EXCEPTION WHEN OTHERS THEN
  --Raise_Application_Error(-20000, 'Wrong table name');
END show_col2;
/


execute show_col2('Email','EMPLOYEES');
execute show_col2('Email','EMP');
execute show_col2('Email','EMPLOYEES where 1=2 union select
Username c1 from All_Users --');
```

DBMS_ASSERT Guidelines

- Do not perform unnecessary uppercase conversions on identifiers.

```
--Bad:  
SAFE_SCHEMA := sys.dbms_assert.SIMPLE_SQL_NAME(UPPER(MY_SCHEMA));  
--Good:  
SAFE_SCHEMA := sys.dbms_assert.SIMPLE_SQL_NAME(MY_SCHEMA);  
--Best:  
SAFE_SCHEMA := sys.dbms_assert.ENQUOTE_NAME(  
SAFE_SCHEMA := sys.dbms_assert.ENQUOTE_LITERAL(  
    sys.dbms_assert.SIMPLE_SQL_NAME(MY_SCHEMA));
```

- When using ENQUOTE_LITERAL, do not add unnecessary double quotation marks around identifiers.

```
--Bad:  
my_trace_routine('||sys.dbms_assert.ENQUOTE_LITERAL(  
my_procedure_name)||');'||...  
--Good:  
my_trace_routine('||sys.dbms_assert.ENQUOTE_LITERAL(  
replace(my_procedure_name,'''','''')||');'||...
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Choosing the correct DBMS_ASSERT verification routines is important, and using them correctly is just as important.

Limitations

DBMS_ASSERT is not a panacea for all sorts of PL/SQL evils. It is essentially a collection of pattern-matching routines that confirm whether the supplied string matches expected patterns. It can be used to protect against certain kinds of malicious input, but it cannot comprehensively defend against all such inputs.

Here are some instances where DBMS_ASSERT may not help:

- It contains no routines to validate TNS connect strings, for example, “((description =...”.
- It is not designed nor is it intended to be a defense against cross-site scripting attacks.
- It does not check for input string lengths, and therefore, cannot be used as any kind of defense against a buffer overflow attack.
- It does not guarantee that a SQL name is, in fact, a parseable SQL name.
- It does not protect against parsing as the wrong user or other security risks due to inappropriate privilege management.

DBMS_ASSERT Guidelines

- Check and reject NULL or empty return results from DBMS_ASSERT (test for NULL, ' ', and ''''').
- Prefix all calls to DBMS_ASSERT with the owning schema, SYS.
- Protect all injectable parameters and code paths.



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Protect all injectable parameters and code paths.

- **Bad**

```
FUNCTION name_elided
  (LAYER VARCHAR2, OWNER VARCHAR2, FIELD VARCHAR2)
RETURN BOOLEAN IS
  CRS INTEGER;
BEGIN
  CRS := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQLPARSE(CRS, 'select ||FIELD|| from
  ||OWNER||.|'|||
  sys.dbms_assert.QUALIFIED_SQL_NAME(LAYER)||'_elided',
  DBMS_SQL.NATIVE);
```

- **Good**

```
FUNCTION name_elided
(LAYER VARCHAR2, OWNER VARCHAR2, FIELD VARCHAR2)
RETURN BOOLEAN IS
CRS INTEGER;
BEGIN
CRS := DBMS_SQL.OPEN_CURSOR;
DBMS_SQLPARSE(CRS, 'select
'||sys.dbms_assert.SIMPLE_SQL_NAME(FIELD)||' from
'||sys.dbms_assert.SIMPLE_SQL_NAME(OWNER)||'.' ||
sys.dbms_assert.SIMPLE_SQL_NAME(LAYER)||'_elided',
DBMS_SQL.NATIVE);
```

DBMS_ASSERT Guidelines

- If DBMS_ASSERT exceptions are raised from a number of input strings, define and raise exceptions explicitly to ease debugging during application development.

```
-- Bad
CREATE OR REPLACE PROCEDURE change_password3
  (username VARCHAR2, password VARCHAR2)
AS
BEGIN
  ...
EXCEPTION WHEN OTHERS THEN
  RAISE;
END;
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You can use PL/SQL conditional compilation from Oracle Database 10.2 and later to manage self-tracing code. For production deployment, the debug messages can be turned off by setting the PLSQL_CCFLAGS parameter for tracing to FALSE. You must ensure that error messages displayed in production deployment do not reveal information that is useful to hackers.

```
ALTER SESSION SET Plsql_CCFlags = 'Tracing:true';

CREATE OR REPLACE PROCEDURE change_password3
  (p_username VARCHAR2, p_password VARCHAR2) AS
BEGIN
  ...
EXCEPTION
  WHEN sys.dbms_assert.INVALID_SCHEMA_NAME THEN
    $if $$Tracing $then dbms_output.put_line('Invalid user.');
    $else dbms_output.put_line('Authentication failed.');
  $end
  WHEN sys.dbms_assert.INVALID_SQL_NAME THEN
    $if $$Tracing $then dbms_output.put_line('Invalid pw.');
    $else dbms_output.put_line('Authentication failed.');
  $end
  WHEN OTHERS THEN
    dbms_output.put_line('Something else went wrong');
END;
```

Quiz

Code that is most vulnerable to SQL Injection attack contains:

- a. Input parameters
- b. Dynamic SQL with bind arguments
- c. Dynamic SQL with concatenated input values
- d. Calls to external functions



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Answer: c

Quiz

By default, a stored procedure executes with the privileges of its owner (definer's rights).

- a. True
- b. False



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Answer: a

Quiz

If you must use dynamic SQL, avoid using input concatenation to build the dynamic SQL.

- a. True
- b. False



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Answer: a

Quiz

In the statement `SELECT total FROM orders WHERE ord_id=p_ord_id`, the table name `orders` is being used as which of the following ?

- a. A literal
- b. An identifier
- c. A placeholder
- d. An argument



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Answer: b

Summary

In this lesson, you should have learned how to:

- Detect SQL injection vulnerabilities
- Reduce attack surfaces
- Use DBMS_ASSERT



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

This lesson showed you techniques and tools to strengthen your code and applications against SQL injection attacks.

For more information, see “Appendix D, Designing and Testing Your Code to Avoid SQL Injection Attacks.”

Practice 13: Overview

This practice covers the following topics:

- Testing your knowledge of SQL injection
- Rewriting code to protect against SQL injection



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Using the OE, you examine PL/SQL code, test it for SQL injection, and rewrite it to protect against SQL injection vulnerabilities.

Use the OE schema for this practice.

Note: For more information about examining, testing, and rewriting a PL/SQL code, see “Appendix D: Designing and Testing Your Code to Avoid SQL Injection Attacks.”

THESE eKIT MATERIALS ARE FOR YOUR USE IN THIS CLASSROOM ONLY. COPYING eKIT MATERIALS FROM THIS COMPUTER IS STRICTLY PROHIBITED

Oracle University and Error : You are not a Valid Partner use only

Table Descriptions and Data



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Schema Descriptions

Overall Description

The sample company portrayed by the Oracle Database Sample Schemas operates worldwide to fulfill orders for several different products. The company has several divisions:

- The Human Resources division tracks information about the employees and facilities of the company.
- The Order Entry division tracks product inventories and sales of the products of the company through various channels.
- The Sales History division tracks business statistics to facilitate business decisions.

Each division is represented by a schema. In this course, you have access to the objects in all these schemas. However, the examples, demonstrations, and practices utilize the Order Entry (OE) schema.

All scripts necessary to create the sample schemas reside in the \$ORACLE_HOME/demo/schema/ folder.

Schema Descriptions (continued)

Order Entry (OE)

The company sells several categories of products, including computer hardware and software, music, clothing, and tools. The company maintains product information that includes product identification numbers, the category into which the product falls, the weight group (for shipping purposes), the warranty period if applicable, the supplier, the status of the product, a list price, a minimum price at which a product will be sold, and a URL for manufacturer information.

Inventory information is also recorded for all products, including the warehouse where the product is available and the quantity on hand. Because products are sold worldwide, the company maintains the names of the products and their descriptions in different languages.

The company maintains warehouses in several locations to facilitate filling customer orders. Each warehouse has a warehouse identification number, name, and location identification number.

Customer information is tracked in some detail. Each customer is assigned an identification number. Customer records include name, street address, city or province, country, phone numbers (up to five phone numbers for each customer), and postal code. Some customers order through the Internet, so email addresses are also recorded. Because of language differences among customers, the company records the National Language Support (NLS) language and territory of each customer. The company places a credit limit on its customers to limit the amount for which they can purchase at one time. Some customers have account managers, whom the company monitors. It keeps track of a customer's phone number. These days, you never know how many phone numbers a customer might have, but you try to keep track of all of them. Because of the language differences of the customers, you identify the language and territory of each customer.

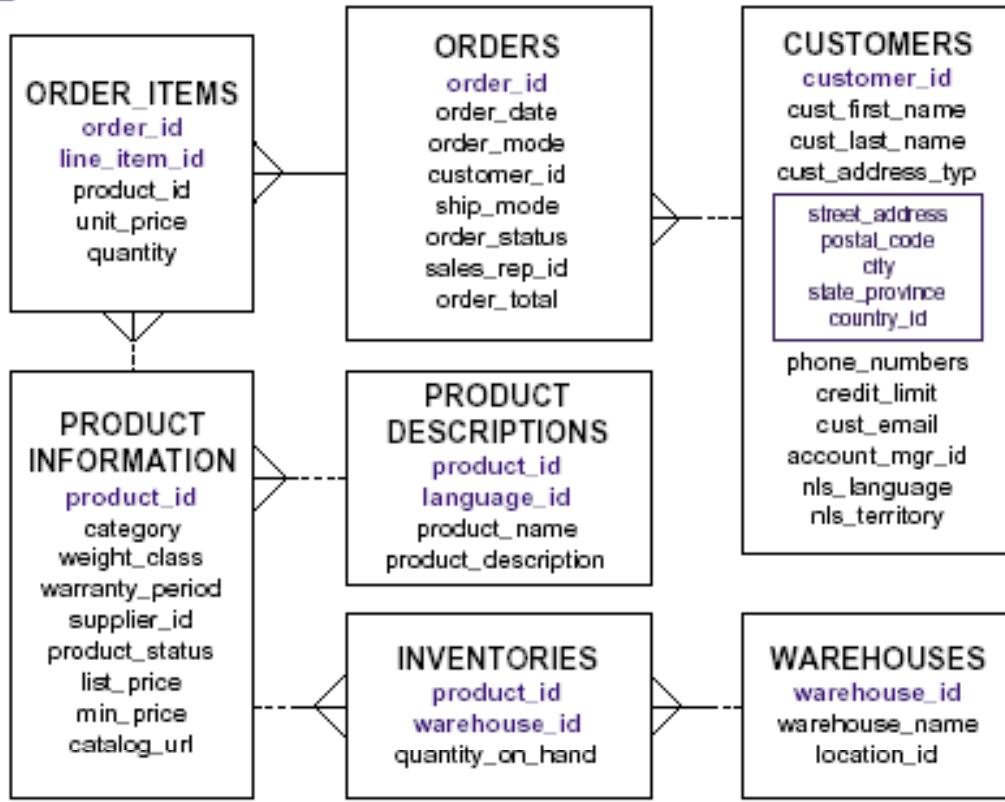
When a customer places an order, the company tracks the date of the order, the mode of the order, status, shipping mode, total amount of the order, and the sales representative who helped place the order. This may be the same individual as the account manager for a customer, it may be a different individual, or, in the case of an order over the Internet, the sales representative is not recorded. In addition to the order information, the company also tracks the number of items ordered, the unit price, and the products ordered.

For each country in which it does business, the company records the country name, currency symbol, currency name, and the region where the country resides geographically. This data is useful to interact with customers living in different geographic regions around the world.

Schema Descriptions (continued)

Order Entry (OE)

OE



Schema Descriptions (continued)

Order Entry (OE) Row Counts

```
SELECT COUNT(*) FROM customers;  
COUNT(*)
```

```
-----
```

```
319
```

```
SELECT COUNT(*) FROM inventories;  
COUNT(*)
```

```
-----
```

```
1112
```

```
SELECT COUNT(*) FROM orders;  
COUNT(*)
```

```
-----
```

```
105
```

```
SELECT COUNT(*) FROM order_items;  
COUNT(*)
```

```
-----
```

```
665
```

```
SELECT COUNT(*) FROM product_descriptions;  
COUNT(*)
```

```
-----
```

```
8640
```

```
SELECT COUNT(*) FROM product_information;  
COUNT(*)
```

```
-----
```

```
288
```

```
SELECT COUNT(*) FROM warehouses;  
COUNT(*)
```

```
-----
```

```
9
```

Schema Descriptions (continued)

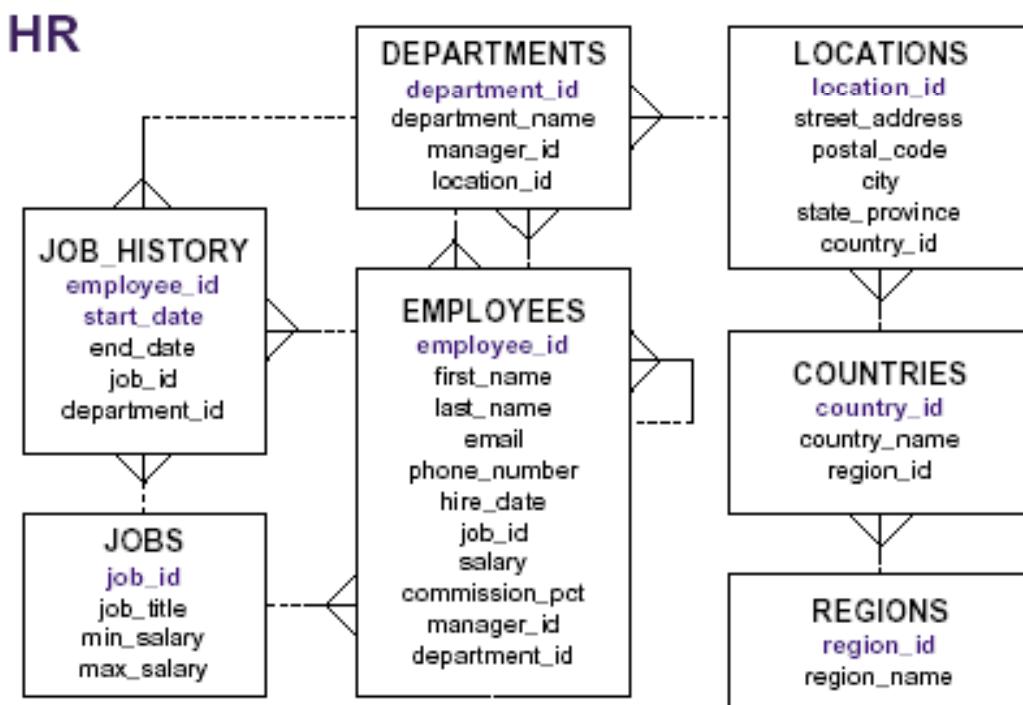
Human Resources (HR)

In the human resource records, each employee has an identification number, email address, job identification code, salary, and manager. Some employees earn a commission in addition to their salary.

The company also tracks information about the jobs within the organization. Each job has an identification code, job title, and a minimum and maximum salary range for the job. Some employees have been with the company for a long time and have held different positions within the company. When an employee switches jobs, the company records the start date and end date of the former job, the job identification number, and the department.

The sample company is regionally diverse, so it tracks the locations of not only its warehouses but also its departments. Each company employee is assigned to a department. Each department is identified by a unique department number and a short name. Each department is associated with one location. Each location has a full address that includes the street address, postal code, city, state or province, and country code.

For each location where it has facilities, the company records the country name, currency symbol, currency name, and the region where the country resides geographically.



Schema Descriptions (continued)

Human Resources (HR) Row Counts

```
SELECT COUNT(*) FROM employees;
```

```
COUNT(*)
```

```
-----
```

```
107
```

```
SELECT COUNT(*) FROM departments;
```

```
COUNT(*)
```

```
-----
```

```
27
```

```
SELECT COUNT(*) FROM locations;
```

```
COUNT(*)
```

```
-----
```

```
23
```

```
SELECT COUNT(*) FROM countries;
```

```
COUNT(*)
```

```
-----
```

```
25
```

```
SELECT COUNT(*) FROM regions;
```

```
COUNT(*)
```

```
-----
```

```
4
```

```
SELECT COUNT(*) FROM jobs;
```

```
COUNT(*)
```

```
-----
```

```
19
```

```
SELECT COUNT(*) FROM job_history;
```

```
COUNT(*)
```

```
-----
```

```
10
```

THESE eKIT MATERIALS ARE FOR YOUR USE IN THIS CLASSROOM ONLY. COPYING eKIT MATERIALS FROM THIS COMPUTER IS STRICTLY PROHIBITED

Oracle University and Error : You are not a Valid Partner use only

Using SQL Developer

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Objectives

After completing this appendix, you should be able to do the following:

- List the key features of Oracle SQL Developer
- Identify the menu items of Oracle SQL Developer
- Create a database connection
- Manage database objects
- Use SQL Worksheet
- Save and run SQL scripts
- Create and save reports
- Browse the Data Modeling options in SQL Developer

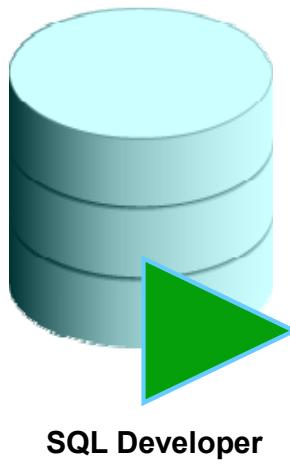


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In this appendix, you are introduced to the graphical tool called SQL Developer. You learn how to use SQL Developer for your database development tasks. You learn how to use SQL Worksheet to execute SQL statements and SQL scripts.

What Is Oracle SQL Developer?

- Oracle SQL Developer is a graphical tool that enhances productivity and simplifies database development tasks.
- You can connect to any target Oracle database schema by using standard Oracle database authentication.



SQL Developer

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Oracle SQL Developer is a free graphical tool designed to improve your productivity and simplify the development of everyday database tasks. With just a few clicks, you can easily create and debug stored procedures, test SQL statements, and view optimizer plans.

SQL Developer, which is the visual tool for database development, simplifies the following tasks:

- Browsing and managing database objects
- Executing SQL statements and scripts
- Editing and debugging PL/SQL statements
- Creating reports

You can connect to any target Oracle database schema by using standard Oracle database authentication. When connected, you can perform operations on objects in the database.

SQL Developer is the interface to administer the Oracle Application Express Listener. The new interface enables you to specify global settings and multiple database settings with different database connections for the Application Express Listener. SQL Developer provides the option to drag objects by table or column name to the worksheet. It provides improved DB Diff comparison options, GRANT statements support in the SQL editor, and DB Doc reporting. Additionally, SQL Developer includes support for Oracle Database 12c features.

Specifications of SQL Developer

- Is shipped along with Oracle Database 12c Release 1
- Is developed in Java
- Supports Windows, Linux, and Mac OS X platforms
- Enables default connectivity using the JDBC Thin driver
- Connects to Oracle Database 9*i* and later



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

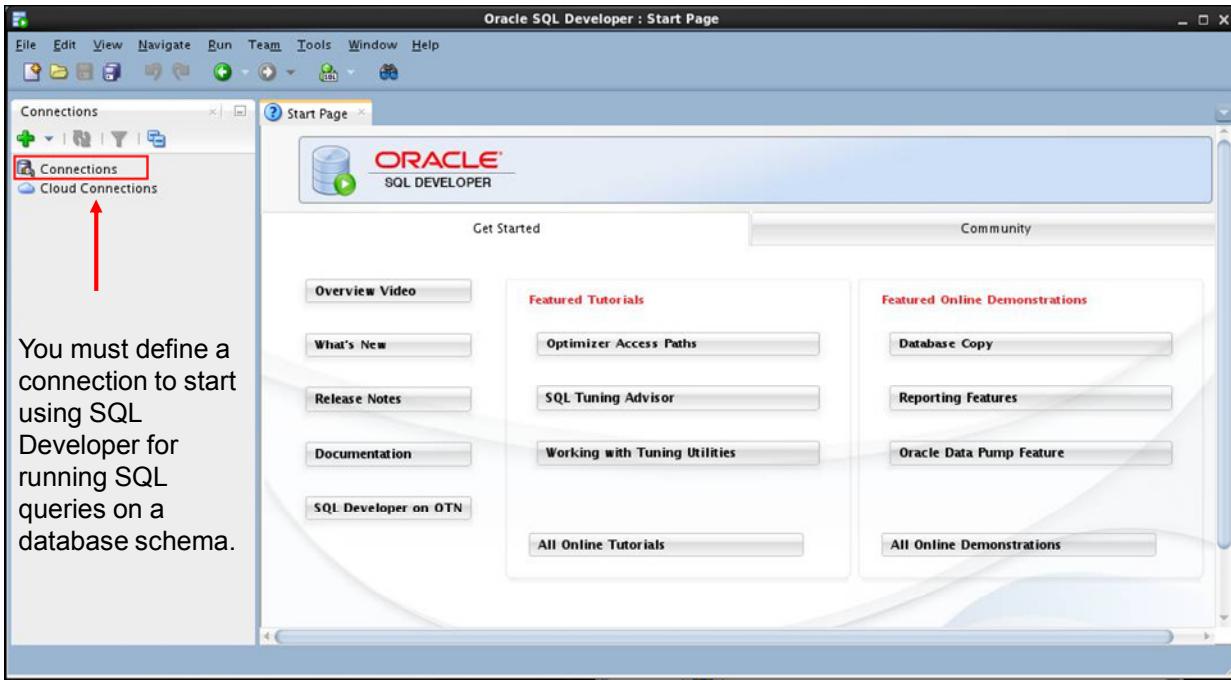
Oracle SQL Developer is shipped along with Oracle Database 12c Release 1 by default. SQL Developer is developed in Java, leveraging the Oracle JDeveloper integrated development environment (IDE). Therefore, it is a cross-platform tool. The tool runs on Windows, Linux, and Mac operating system (OS) X platforms.

The default connectivity to the database is through the Java Database Connectivity (JDBC) Thin driver, and therefore, no Oracle Home is required. SQL Developer does not require an installer and you need to simply unzip the downloaded file. With SQL Developer, users can connect to Oracle Databases 9.2.0.1 and later, and all Oracle database editions, including Express Edition.

Note

- For Oracle Database 12c Release 1, you will have to download and install SQL Developer. SQL Developer is freely downloadable from the following link:
<http://www.oracle.com/technetwork/developer-tools/sql-developer/downloads/index.html>
- For instructions on how to install SQL Developer, see the website at:
<http://www.oracle.com/technetwork/developer-tools/sql-developer/overview/index.html>

SQL Developer 4.0 Interface



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The SQL Developer 4.0 interface contains three main navigation tabs:

- **Connections:** By using this tab, you can browse database objects and users to which you have access. The Connections tab is displayed by default.
- **Reports:** Identified by the Reports icon, this tab enables you to run predefined reports or create and add your own reports. To view the Reports Navigator, click the View tab and select Files, or select **View > Reports**.
- **Files:** Identified by the Files folder icon, this tab enables you to access files from your local machine without having to use the File > Open menu. To view the File Navigator, click the View tab and select Files, or select **View > Files**. You can double-click or drag files to open them, and you can edit and save the files. For example, if you open a .sql file, it is displayed in a SQL Worksheet window. The Files navigator is especially useful if you are using versioning with SQL Developer.

General Navigation and Use

SQL Developer uses the left side for navigation to find and select objects, and the right side to display information about selected objects. You can customize many aspects of the appearance and behavior of SQL Developer by setting preferences.

Note: You need to define at least one connection to be able to connect to a database schema and issue SQL queries or run procedures/functions.

Menus

The following menus contain standard entries, plus entries for features specific to SQL Developer:

- **Edit:** Contains options for editing files.
 - Extended Paste: Displays the Paste dialog box, in which you select a clipboard item (from potentially many) to be pasted into the current location
 - Duplicate Selection: When you have selected text while editing a function or procedure, creates a copy of the selected text at the current location
 - Wrap Selection: When you have selected text while editing a function or procedure, wraps the selected text
- **View:** Contains options that affect what is displayed in the SQL Developer interface
- **Navigate:** Contains options for navigating to panes and for executing subprograms
- **Run:** Contains the Run File and Execution Profile options that are relevant when a function or procedure is selected, and also debugging options
- **Team:** Contains options related to support for the Subversion version management and source control system, and for any other such systems (such as CVS) that you have added as extensions to SQL Developer through the "check for updates" feature
- **Versioning:** Provides integrated support for the following versioning and source control systems: CVS (Concurrent Versions System) and Subversion
- **Tools:** Invokes SQL Developer tools such as SQL*Plus, Preferences, and SQL Worksheet. It also contains options related to migrating third-party databases to Oracle.
- **Window:** Contains options relating to the appearance and behavior of the user interface
- **Help:** Displays help about SQL Developer and enables you to check for SQL Developer updates

Note: The Run menu also contains options that are relevant when a function or procedure is selected for debugging.

Creating a Database Connection

- You must have at least one database connection to use SQL Developer.
- You can create and test connections for multiple:
 - Databases
 - Schemas
- SQL Developer automatically imports any connections defined in the `tnsnames.ora` file on your system.
- You can export connections to an Extensible Markup Language (XML) file.
- Each additional database connection created is listed in the Connections Navigator hierarchy.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

A connection is a SQL Developer object that specifies the necessary information for connecting to a specific database as a specific user of that database. To use SQL Developer, you must have at least one database connection, which may be existing, created, or imported.

You can create and test connections for multiple databases and for multiple schemas.

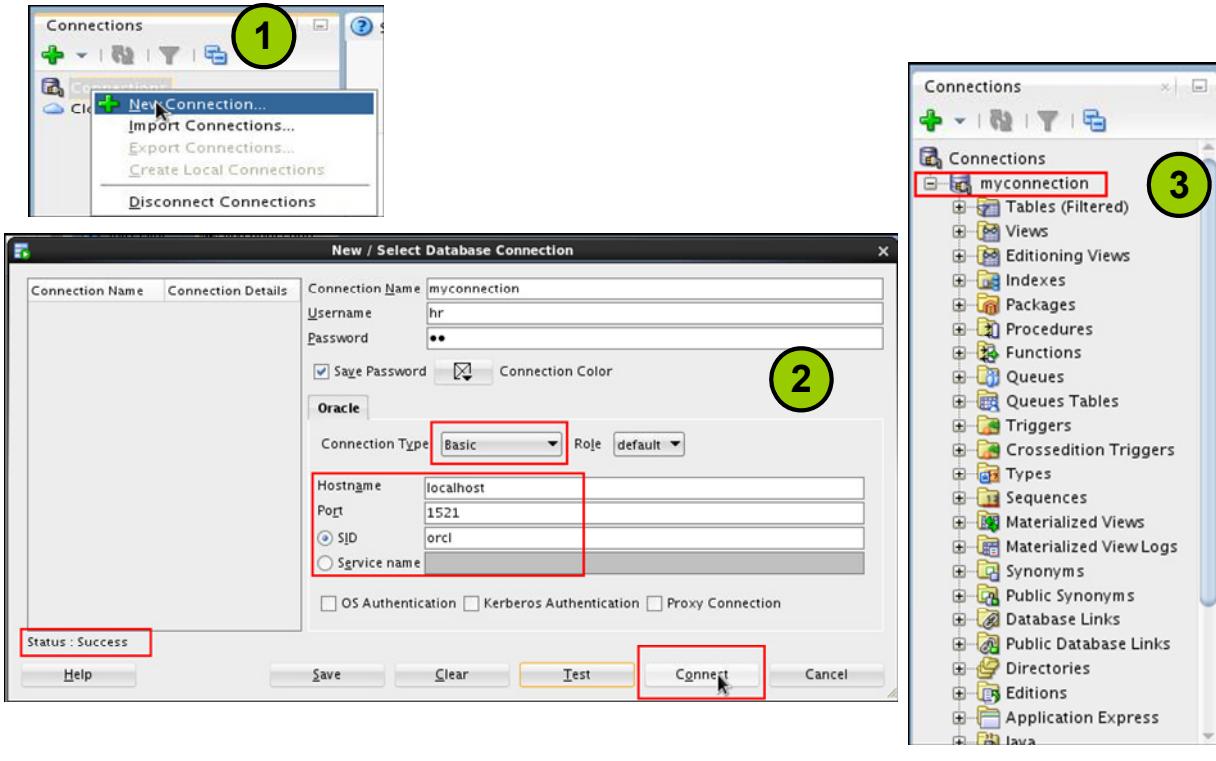
By default, the `tnsnames.ora` file is located in the `$ORACLE_HOME/network/admin` directory, but it can also be in the directory specified by the `TNS_ADMIN` environment variable or registry value. When you start SQL Developer and open the Database Connections dialog box, SQL Developer automatically imports any connections defined in the `tnsnames.ora` file on your system.

Note: On Windows, if the `tnsnames.ora` file exists, but its connections are not being used by SQL Developer, define `TNS_ADMIN` as a system environment variable.

You can export connections to an XML file so that you can reuse it.

You can create additional connections as different users to the same database or to connect to the different databases.

Creating a Database Connection



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

To create a database connection, perform the following steps:

1. On the Connections tabbed page, right-click Connections and select New Connection.
2. In the “New / Select Database Connection” window, enter the connection name. Enter the username and password of the schema that you want to connect to.
 - a. From the Role drop-down list, you can select either *default* or *SYSDBA*. (You choose *SYSDBA* for the *sys* user or any user with database administrator privileges.)
 - b. You can select the connection type as:
 - Basic:** In this type, enter the host name and SID for the database that you want to connect to. Port is already set to 1521. You can also choose to enter the Service name directly if you use a remote database connection.
 - TNS:** You can select any one of the database aliases imported from the *tnsnames.ora* file.
 - LDAP:** You can look up database services in Oracle Internet Directory, which is a component of Oracle Identity Management.
 - Advanced:** You can define a custom Java Database Connectivity (JDBC) URL to connect to the database.

Local/Bequeath: If the client and database exist on the same computer, a client connection can be passed directly to a dedicated server process without going through the listener.

- c. Click Test to ensure that the connection has been set correctly.
- d. Click Connect.

If you select the Save Password check box, the password is saved to an XML file. So, after you close the SQL Developer connection and open it again, you are not prompted for the password.

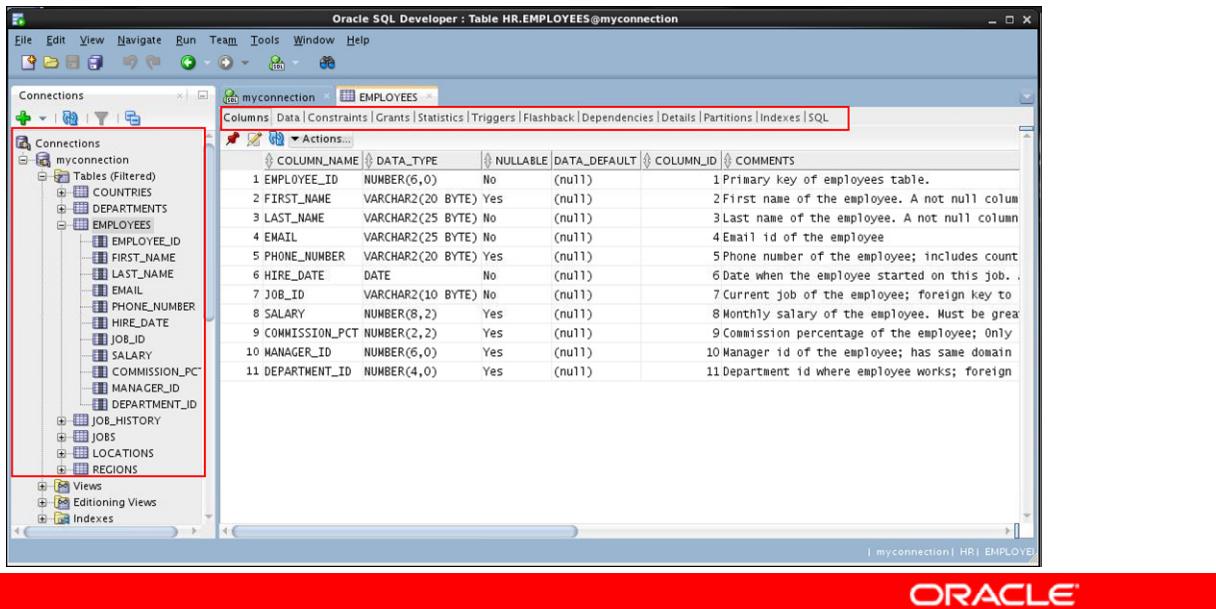
3. The connection gets added in the Connections Navigator. You can expand the connection to view the database objects and view object definitions (dependencies, details, statistics, and so on).

Note: From the same “New / Select Database Connection” window, you can define connections to non-Oracle data sources by using the Access, MySQL, and SQL Server tabs. However, these connections are read-only connections that enable you to browse objects and data in that data source.

Browsing Database Objects

Use the Connections Navigator to:

- Browse through many objects in a database schema
- Review the definitions of objects at a glance



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

ORACLE

After you create a database connection, you can use the Connections Navigator to browse through many objects in a database schema, including Tables, Views, Indexes, Packages, Procedures, Triggers, and Types.

SQL Developer uses the left side for navigation to find and select objects, and the right side to display information about the selected objects. You can customize many aspects of the appearance of SQL Developer by setting preferences.

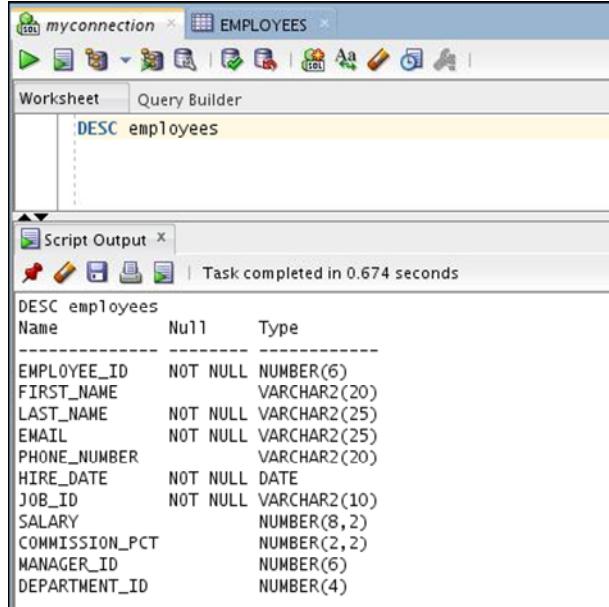
You can see the definition of the objects broken into tabs of information that is pulled out of the data dictionary. For example, if you select a table in the Navigator, details about columns, constraints, grants, statistics, triggers, and so on are displayed on an easy-to-read tabbed page.

If you want to see the definition of the EMPLOYEES table as shown in the slide, perform the following steps:

1. Expand the Connections node in the Connections Navigator.
2. Expand Tables.
3. Click EMPLOYEES. By default, the Columns tab is selected. It shows the column description of the table. Using the Data tab, you can view the table data and also enter new rows, update data, and commit these changes to the database.

Displaying the Table Structure

Use the DESCRIBE command to display the structure of a table:



The screenshot shows the Oracle SQL Developer interface. In the top-left corner, it says "myconnection" and "EMPLOYEES". Below that, there are tabs for "Worksheet" and "Query Builder", with "Worksheet" selected. In the main workspace, the command "DESC employees" is entered. To the right, under the heading "Script Output", the results of the DESCRIBE command are displayed in a table format:

Name	Null	Type
EMPLOYEE_ID	NOT NULL	NUMBER(6)
FIRST_NAME		VARCHAR2(20)
LAST_NAME	NOT NULL	VARCHAR2(25)
EMAIL	NOT NULL	VARCHAR2(25)
PHONE_NUMBER		VARCHAR2(20)
HIRE_DATE	NOT NULL	DATE
JOB_ID	NOT NULL	VARCHAR2(10)
SALARY		NUMBER(8,2)
COMMISSION_PCT		NUMBER(2,2)
MANAGER_ID		NUMBER(6)
DEPARTMENT_ID		NUMBER(4)

Below the table, a message says "Task completed in 0.674 seconds".

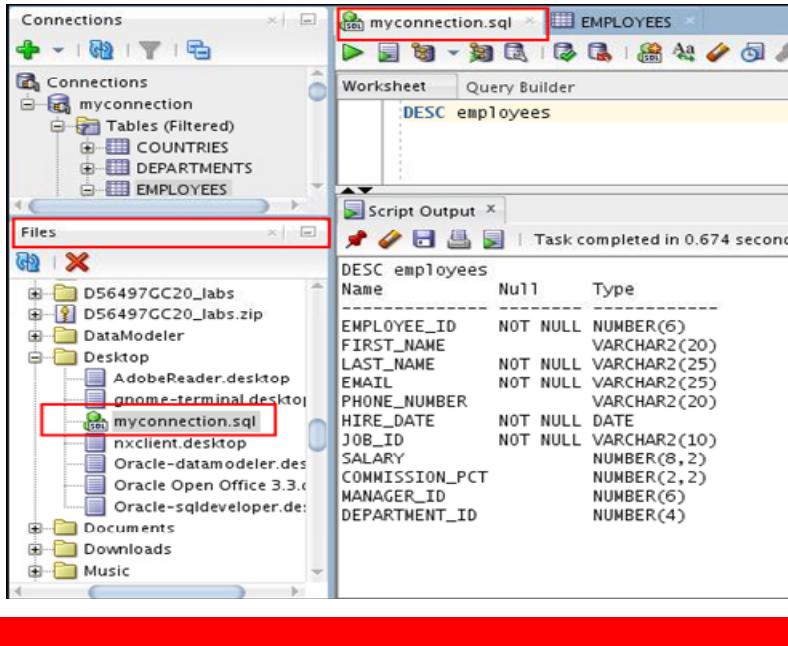
ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In SQL Developer, you can also display the structure of a table by using the DESCRIBE command. The result of the command is a display of column names and data types, as well as an indication of whether a column must contain data.

Browsing Files

Use the File Navigator to explore the file system and open system files.



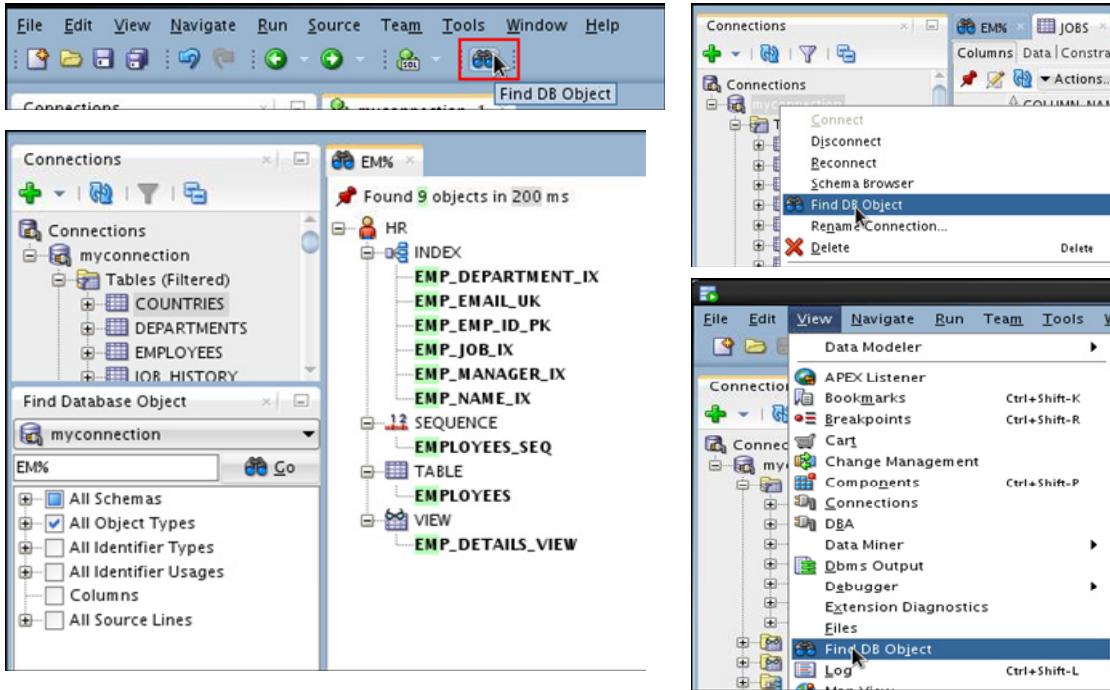
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Browsing Database Objects

You can use the File Navigator to browse and open system files.

- To view the File Navigator, click the View tab and select Files, or select View > Files.
- To view the contents of a file, double-click a file name to display its contents in the SQL Worksheet area.

Finding Database Objects



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You can find various types of objects (tables, columns, declarations within functions or procedures, and so on) associated with an Oracle database connection, and open editing panes to work with those objects. To open the Find Database Object pane, right-click a connection name in the Connections navigator and select **Find DB Object**. You can also click View, and then select **Find DB Object**, or you can click the **Find DB Objects** icon in the menu bar.

Select the desired database connection, the types of objects for which to search, and a search string. The following figure shows Find Database Objects pane with results from a search for all objects associated with a connection named **myconnection** where the object name starts with EM.

In this example, the search finds nine objects across four object types. In the display of results, you can click any of the object names to bring up the object in an appropriate editor.

Connection: Database connection to use for the search

Name: An object name or a string containing one or more wildcard characters—for example, EM% for all names starting with EM

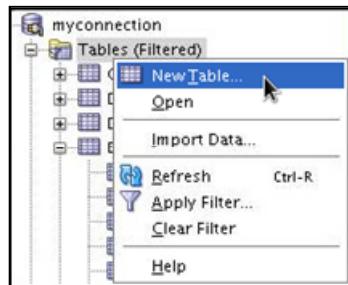
All Schemas: Select one or more schemas, or all schemas, in which to perform the search. In this example, the blue solid square reflects the fact that the schema you selected is the schema to search; however, you click to select all schemas, or you can expand All Schemas and individually select schemas.

Types: Type of object for which to restrict the search. You can search all columns, or all or individual identifier types, identifier usages, and source lines.

Click the Go icon to display the objects that meet the specified criteria. To view or edit one of the objects (or the parent object that contains the specified object), double-click or right-click its name in the results display.

Creating a Schema Object

- SQL Developer supports the creation of any schema object by:
 - Executing a SQL statement in SQL Worksheet
 - Using the context menu
- Edit the objects by using an edit dialog box or one of the many context-sensitive menus.
- View the data definition language (DDL) for adjustments such as creating a new object or editing an existing schema object.



ORACLE

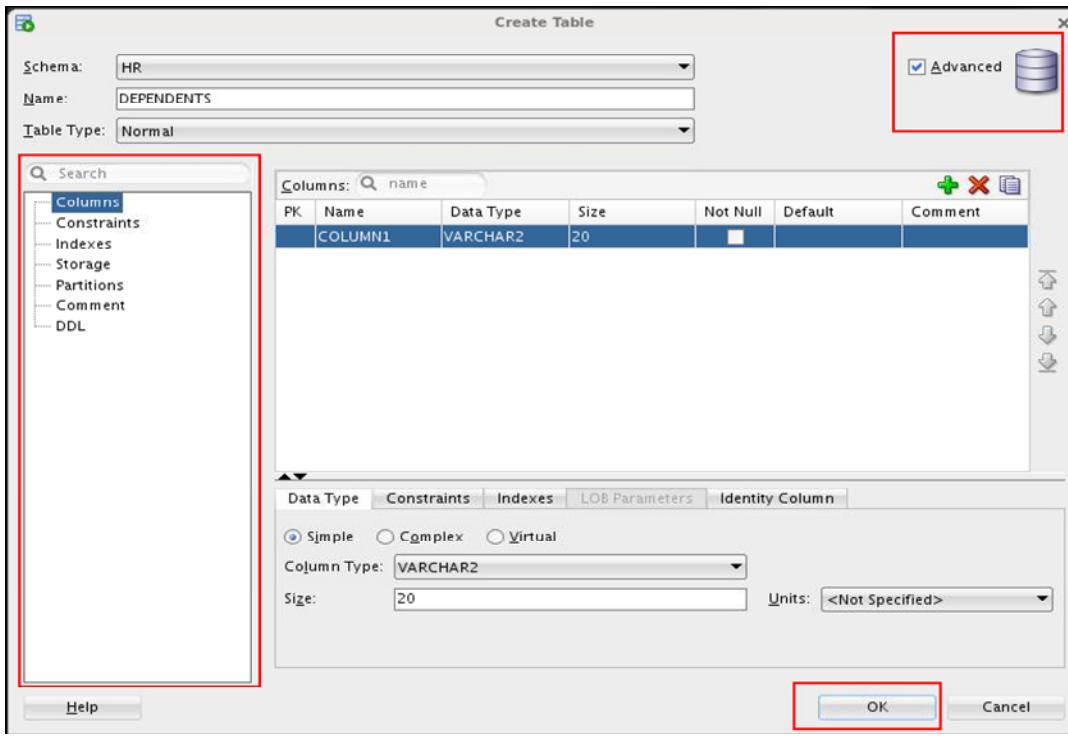
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

SQL Developer supports the creation of any schema object by executing a SQL statement in SQL Worksheet. Alternatively, you can create objects by using the context menus. When created, you can edit objects using an edit dialog box or one of the many context-sensitive menus.

As new objects are created or existing objects are edited, the DDL for those adjustments is available for review. An Export DDL option is available if you want to create the full DDL for one or more objects in the schema.

The slide shows how to create a table using the context menu. To open a dialog box for creating a new table, right-click Tables and select New Table. The dialog boxes to create and edit database objects have multiple tabs, each reflecting a logical grouping of properties for that type of object.

Creating a New Table: Example



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In the Create Table dialog box, if you do not select the Advanced check box, you can create a table quickly by specifying columns and some frequently used features.

If you select the Advanced check box, the Create Table dialog box changes to one with multiple options, in which you can specify an extended set of features while you create the table.

The example in the slide shows how to create the DEPENDENTS table by selecting the Advanced check box.

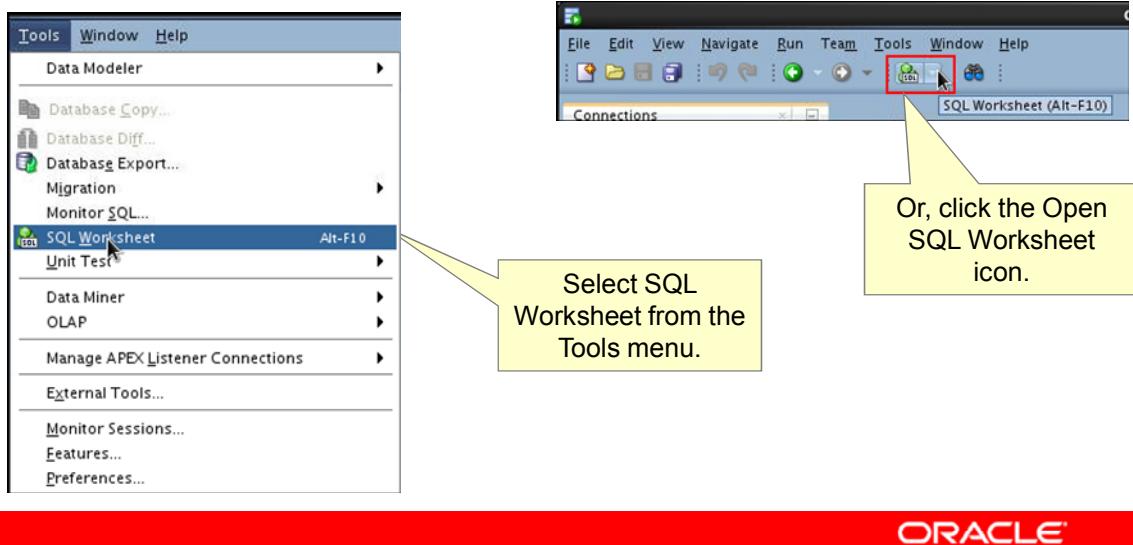
To create a new table, perform the following steps:

1. In the Connections Navigator, right-click Tables and select **New Table**.
2. In the Create Table dialog box, select **Advanced**.
3. Specify the column information.
4. Click **OK**.

Although it is not required, you should also specify a primary key by using the Primary Key tab in the dialog box. Sometimes, you may want to edit the table that you have created; to do so, right-click the table in the Connections Navigator and select Edit.

Using the SQL Worksheet

- Use the SQL Worksheet to enter and execute SQL, PL/SQL, and SQL*Plus statements.
- Specify any actions that can be processed by the database connection associated with the worksheet.



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

When you connect to a database, a SQL Worksheet window for that connection automatically opens. You can use the SQL Worksheet to enter and execute SQL, PL/SQL, and SQL*Plus statements. The SQL Worksheet supports SQL*Plus statements to a certain extent. SQL*Plus statements that are not supported by the SQL Worksheet are ignored and not passed to the database.

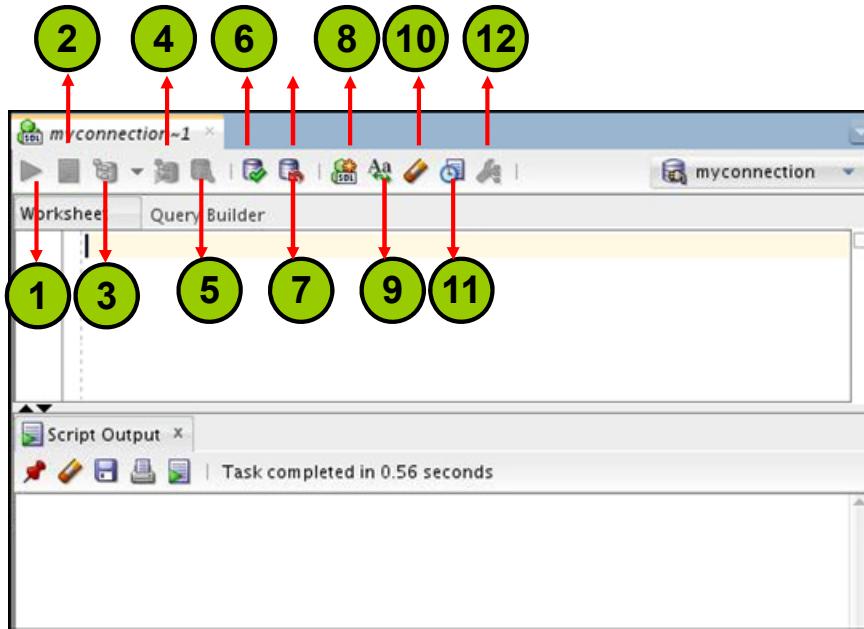
You can specify the actions that can be processed by the database connection associated with the worksheet, such as:

- Creating a table
- Inserting data
- Creating and editing a trigger
- Selecting data from a table
- Saving the selected data to a file

You can display a SQL Worksheet by using one of the following:

- Select **Tools > SQL Worksheet**.
- Click the Open SQL Worksheet icon.

Using the SQL Worksheet



ORACLE®

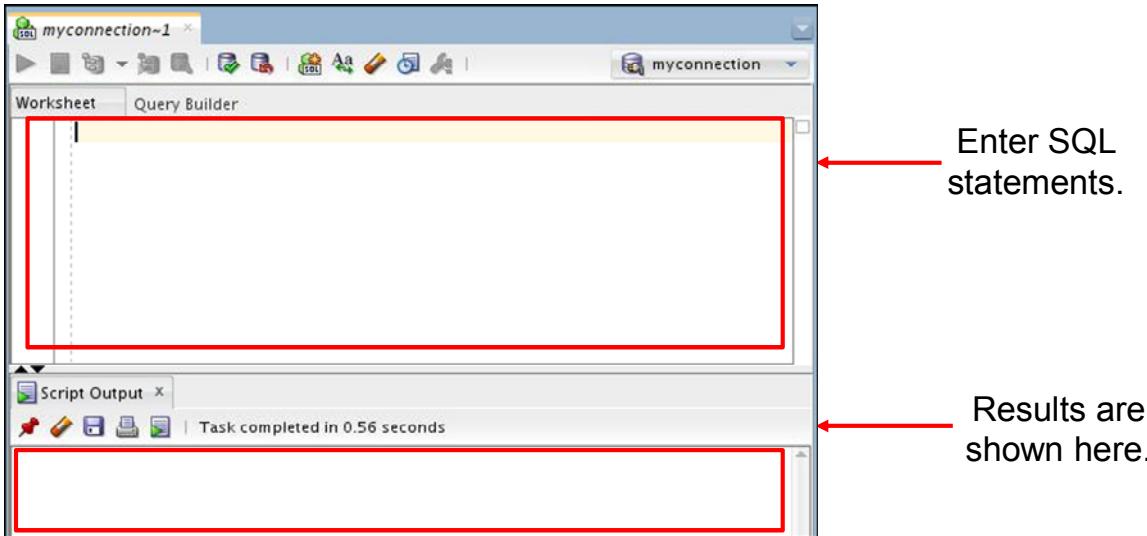
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You may want to use the shortcut keys or icons to perform certain tasks such as executing a SQL statement, running a script, and viewing the history of the SQL statements that you have executed. You can use the SQL Worksheet toolbar that contains icons to perform the following tasks:

1. **Run Statement:** Executes the statement where the cursor is located in the Enter SQL Statement box. You can use bind variables in the SQL statements, but not substitution variables.
2. **Run Script:** Executes all the statements in the Enter SQL Statement box by using the Script Runner. You can use substitution variables in the SQL statements, but not bind variables.
3. **Autotrace:** Generates trace information for the statement
4. **Explain Plan:** Generates the execution plan, which you can see by clicking the Explain tab
5. **SQL Tuning Advisory:** Analyzes high-volume SQL statements and offers tuning recommendations
6. **Commit:** Writes any changes to the database and ends the transaction
7. **Rollback:** Discards any changes to the database, without writing them to the database, and ends the transaction

8. **Unshared SQL Worksheet:** Creates a separate unshared SQL Worksheet for a connection
9. **To Upper/Lower/InitCap:** Changes the selected text to uppercase, lowercase, or initcap, respectively
10. **Clear:** Erases the statement or statements in the Enter SQL Statement box
11. **SQL History:** Displays a dialog box with information about the SQL statements that you have executed

Using the SQL Worksheet



ORACLE®

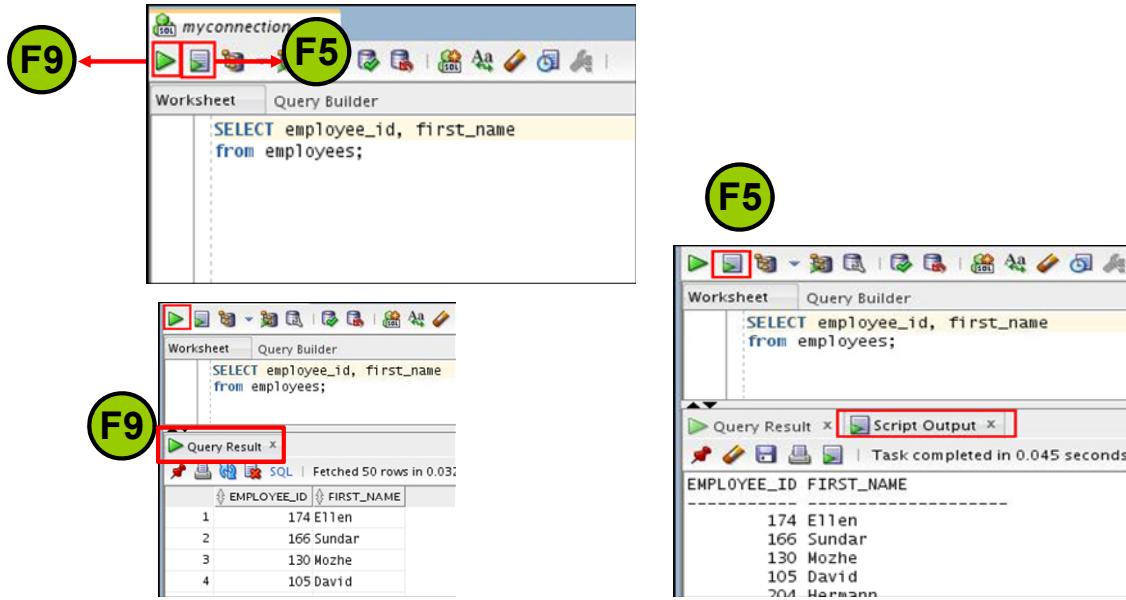
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

When you connect to a database, a SQL Worksheet window for that connection automatically opens. You can use the SQL Worksheet to enter and execute SQL, PL/SQL, and SQL*Plus statements. All SQL and PL/SQL commands are supported as they are passed directly from the SQL Worksheet to the Oracle database. The SQL*Plus commands that are used in SQL Developer must be interpreted by the SQL Worksheet before being passed to the database.

The SQL Worksheet currently supports a number of SQL*Plus commands. Commands that are not supported by the SQL Worksheet are ignored and not sent to the Oracle database. Through the SQL Worksheet, you can execute the SQL statements and some of the SQL*Plus commands.

Executing SQL Statements

Use the Enter SQL Statement box to enter single or multiple SQL statements.

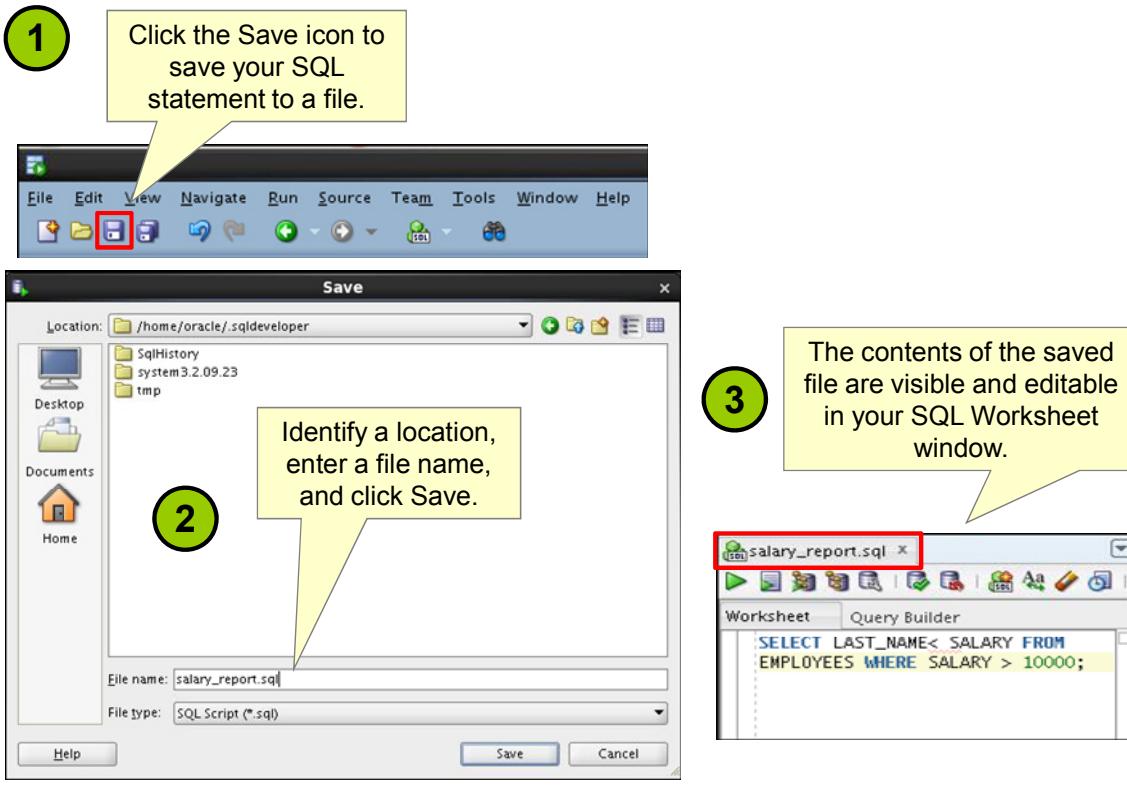


ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The example in the slide shows the difference in output for the same query when the F9 key or Execute Statement is used versus the output when F5 or Run Script is used.

Saving SQL Scripts



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You can save your SQL statements from the SQL Worksheet to a text file. To save the contents of the Enter SQL Statement box, perform the following steps:

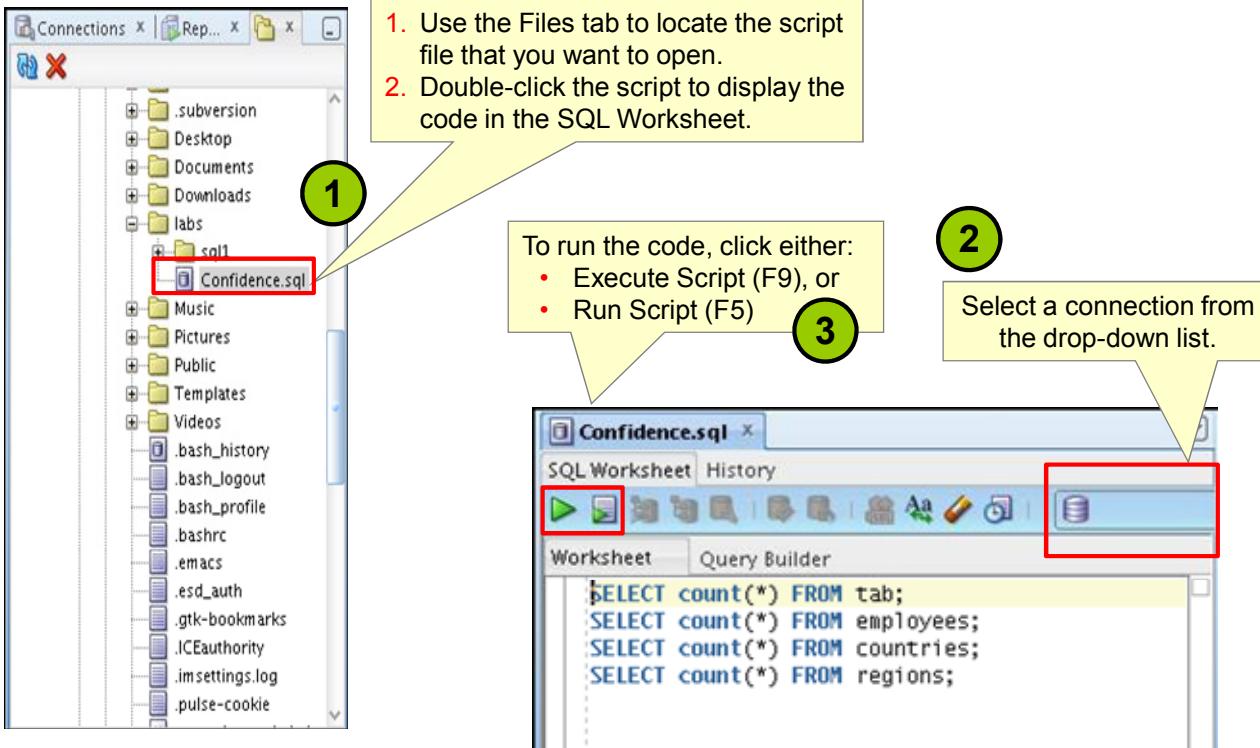
1. Click the Save icon or use the File > Save menu item.
2. In the Save dialog box, enter a file name and the location where you want the file saved.
3. Click Save.

After you save the contents to a file, the Enter SQL Statement window displays a tabbed page of your file contents. You can have multiple files open at the same time. Each file displays as a tabbed page.

Script Pathing

You can select a default path to look for scripts and to save scripts. Under Tools > Preferences > Database > Worksheet Parameters, enter a value in the “Select default path to look for scripts” field.

Executing Saved Script Files: Method 1



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

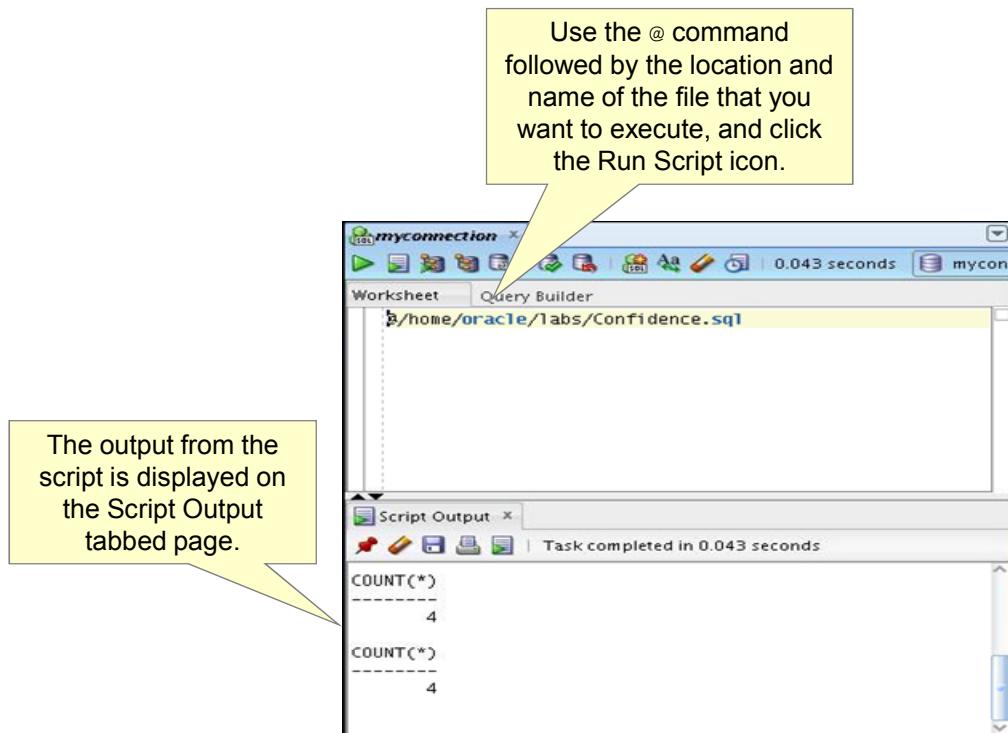
To open a script file and display the code in the SQL Worksheet area, perform the following steps:

1. In the Files navigator, select (or navigate to) the script file that you want to open.
2. Double-click the file to open it. The code of the script file is displayed in the SQL Worksheet area.
3. Select a connection from the connection drop-down list.
4. To run the code, click the Run Script (F5) icon on the SQL Worksheet toolbar. If you have not selected a connection from the connection drop-down list, a connection dialog box will appear. Select the connection that you want to use for the script execution.

Alternatively, you can also do the following:

1. Select File > Open. The Open dialog box is displayed.
2. In the Open dialog box, select (or navigate to) the script file that you want to open.
3. Click Open. The code of the script file is displayed in the SQL Worksheet area.
4. Select a connection from the connection drop-down list.
5. To run the code, click the Run Script (F5) icon on the SQL Worksheet toolbar. If you have not selected a connection from the connection drop-down list, a connection dialog box will appear. Select the connection that you want to use for the script execution.

Executing Saved Script Files: Method 2



ORACLE

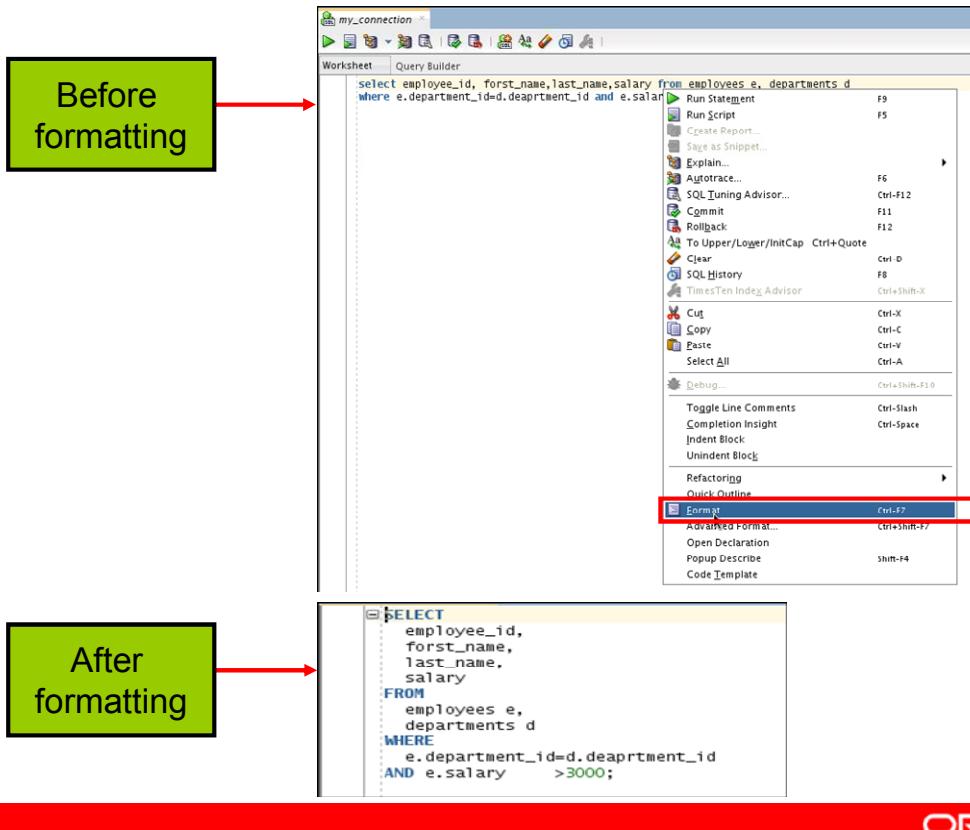
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

To run a saved SQL script, perform the following steps:

1. Use the @ command followed by the location and the name of the file that you want to run in the Enter SQL Statement window.
2. Click the Run Script icon.

The results from running the file are displayed on the Script Output tabbed page. You can also save the script output by clicking the Save icon on the Script Output tabbed page. The File Save dialog box appears and you can identify a name and location for your file.

Formatting the SQL Code



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

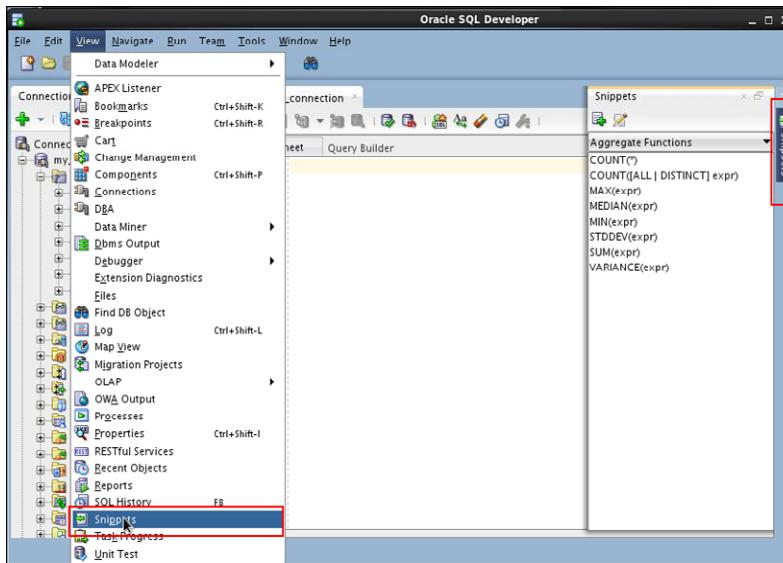
If you want to format the indentation, spacing, capitalization, and line separation of the SQL code, you can use SQL Developer.

To format the SQL code, right-click in the statement area and select Format.

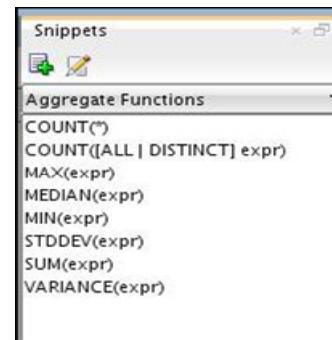
In the example in the slide, before formatting, the SQL code has the keywords not capitalized and the statement not properly indented. After formatting, the SQL code is beautified with the keywords capitalized and the statement properly indented.

Using Snippets

Snippets are code fragments that may be just syntax or examples.



When you place your cursor here, it shows the Snippets window. From the drop-down list, you can select the functions category that you want.



ORACLE

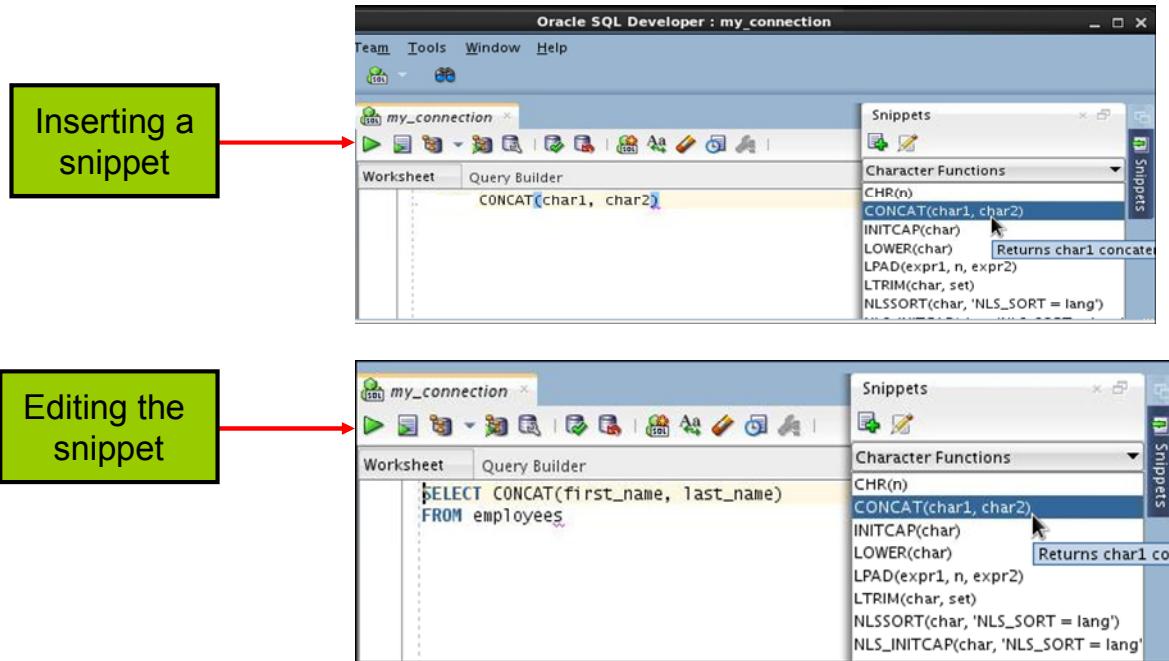
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You may want to use certain code fragments when you use the SQL Worksheet or create or edit a PL/SQL function or procedure. SQL Developer has the feature called Snippets. Snippets are code fragments, such as SQL functions, Optimizer hints, and miscellaneous PL/SQL programming techniques. You can drag snippets into the Editor window.

To display Snippets, select View > Snippets.

The Snippets window is displayed at the right. You can use the drop-down list to select a group. A Snippets button is placed in the right window margin, so that you can display the Snippets window if it becomes hidden.

Using Snippets: Example



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

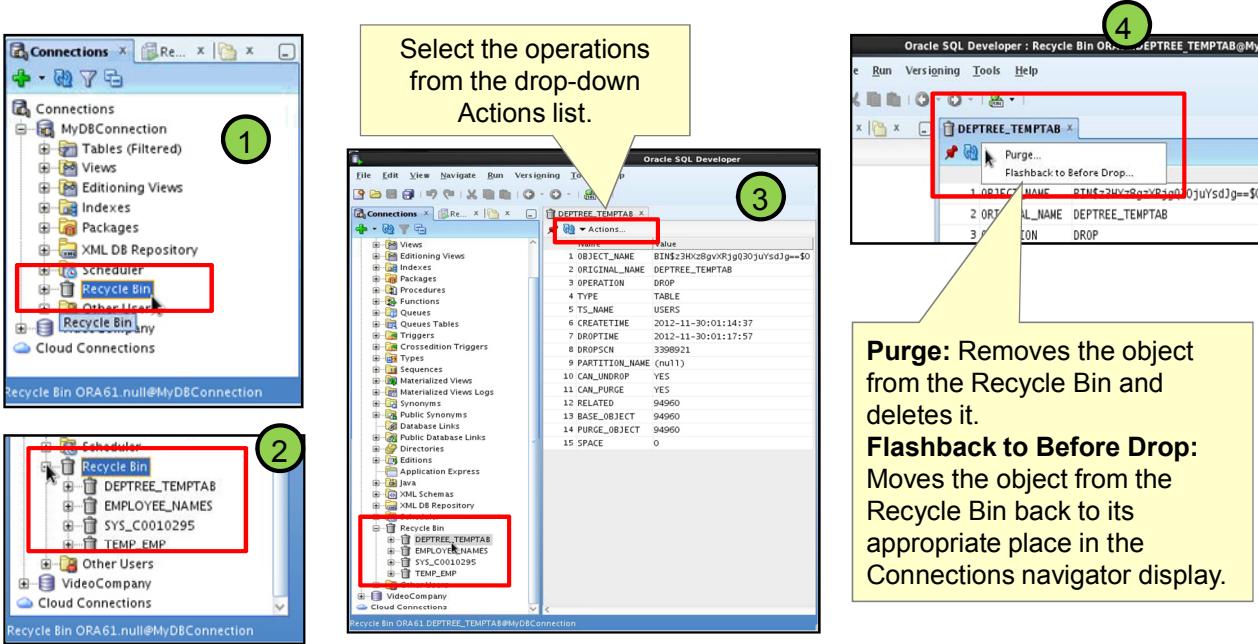
To insert a Snippet into your code in a SQL Worksheet or in a PL/SQL function or procedure, drag the snippet from the Snippets window to the desired place in your code. Then you can edit the syntax so that the SQL function is valid in the current context. To see a brief description of a SQL function in a tooltip, place the cursor over the function name.

The example in the slide shows that `CONCAT(char1, char2)` is dragged from the Character Functions group in the Snippets window. Then the `CONCAT` function syntax is edited and the rest of the statement is added as in the following:

```
SELECT CONCAT(first_name, last_name)
FROM employees;
```

Using Recycle Bin

The Recycle Bin holds objects that have been dropped.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

ORACLE

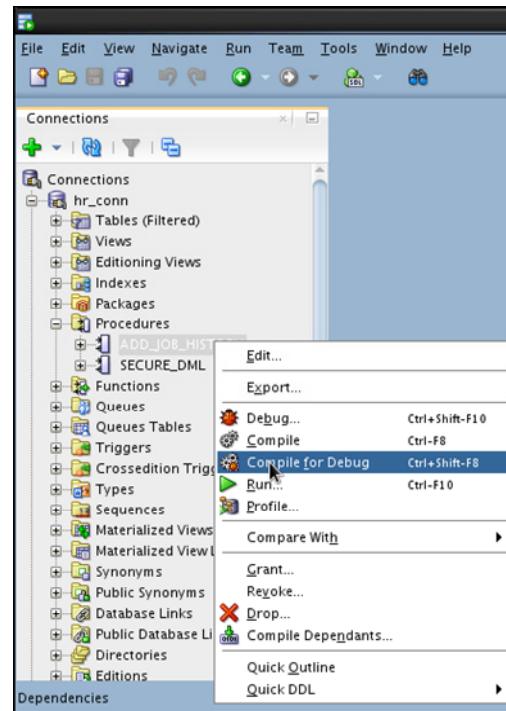
The Recycle Bin is a data dictionary table containing information about dropped objects. Dropped tables and any associated objects such as indexes, constraints, nested tables, and the likes are not removed and still occupy space. They continue to count against user space quotas, until specifically purged from the Recycle Bin or the unlikely situation where they must be purged by the database because of tablespace space constraints.

To use the Recycle Bin, perform the following steps:

1. In the Connections navigator, select (or navigate to) Recycle Bin.
2. Expand Recycle Bin and click the object name. The object details are displayed in the SQL Worksheet area.
3. Click the Actions drop-down list and select the operation that you want to perform on the object.

Debugging Procedures and Functions

- Use SQL Developer to debug PL/SQL functions and procedures.
- Use the “Compile for Debug” option to perform a PL/SQL compilation so that the procedure can be debugged.
- Use the Debug menu options to set breakpoints, and to perform step into and step over tasks.



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In SQL Developer, you can debug PL/SQL procedures and functions. Using the Debug menu options, you can perform the following debugging tasks:

- **Find Execution Point** goes to the next execution point.
- **Resume** continues execution.
- **Step Over** bypasses the next method and goes to the next statement after the method.
- **Step Into** goes to the first statement in the next method.
- **Step Out** leaves the current method and goes to the next statement.
- **Step to End of Method** goes to the last statement of the current method.
- **Pause** halts execution, but does not exit, thus allowing you to resume execution.
- **Terminate** halts and exits the execution. You cannot resume execution from this point; instead, to start running or debugging from the beginning of the function or procedure, click the Run or Debug icon on the Source tab toolbar.
- **Garbage Collection** removes invalid objects from the cache in favor of more frequently accessed and more valid objects.

These options are also available as icons on the debugging toolbar.

Database Reporting

SQL Developer provides a number of predefined reports about the database and its objects.

Owner	Name	Type	Referenced_Owner	Referenced_Name	Referenced_Type
1 APEX_040200	APEX	PROCEDURE	APEX_040200	WW_FLOW	PACKAGE
2 APEX_040200	APEX	PROCEDURE	APEX_040200	WW_FLOW_ISC	PACKAGE
3 APEX_040200	APEX	PROCEDURE	APEX_040200	WW_FLOW_SECURITY	PACKAGE
4 APEX_040200	APEX	PROCEDURE	SYS	STANDARD	PACKAGE
5 APEX_040200	APEX	PROCEDURE	SYS	SYS_STUB_FOR_PURITY_ANALYSIS	PACKAGE
6 APEX_040200	APEXWS	PACKAGE	SYS	STANDARD	PACKAGE
7 APEX_040200	APEX_ADMIN	PROCEDURE	APEX_040200	F	PROCEDURE
8 APEX_040200	APEX_ADMIN	PROCEDURE	SYS	STANDARD	PACKAGE
9 APEX_040200	APEX_ADMIN	PROCEDURE	SYS	SYS_STUB_FOR_PURITY_ANALYSIS	PACKAGE
10 APEX_040200	APEX_APPLICATIONS	VIEW	APEX_040200	NV	FUNCTION
11 APEX_040200	APEX_APPLICATIONS	VIEW	APEX_040200	WW_FLOWS	TABLE
12 APEX_040200	APEX_APPLICATIONS	VIEW	APEX_040200	WW_FLOW_APPLICATION_GROUPS	TABLE
13 APEX_040200	APEX_APPLICATIONS	VIEW	APEX_040200	WW_FLOW_AUTHENTICATIONS	TABLE
14 APEX_040200	APEX_APPLICATIONS	VIEW	APEX_040200	WW_FLOW_COMPANIES	TABLE
15 APEX_040200	APEX_APPLICATIONS	VIEW	APEX_040200	WW_FLOW_COMPANY_SCHEMAS	TABLE
16 APEX_040200	APEX_APPLICATIONS	VIEW	APEX_040200	WW_FLOW_COMPUTATIONS	TABLE
17 APEX_040200	APEX_APPLICATIONS	VIEW	APEX_040200	WW_FLOW_ICON_BAR	TABLE
18 APEX_040200	APEX_APPLICATIONS	VIEW	APEX_040200	WW_FLOW_INSTALL_SCRIPTS	TABLE
19 APEX_040200	APEX_APPLICATIONS	VIEW	APEX_040200	WW_FLOW_ITEMS	TABLE

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

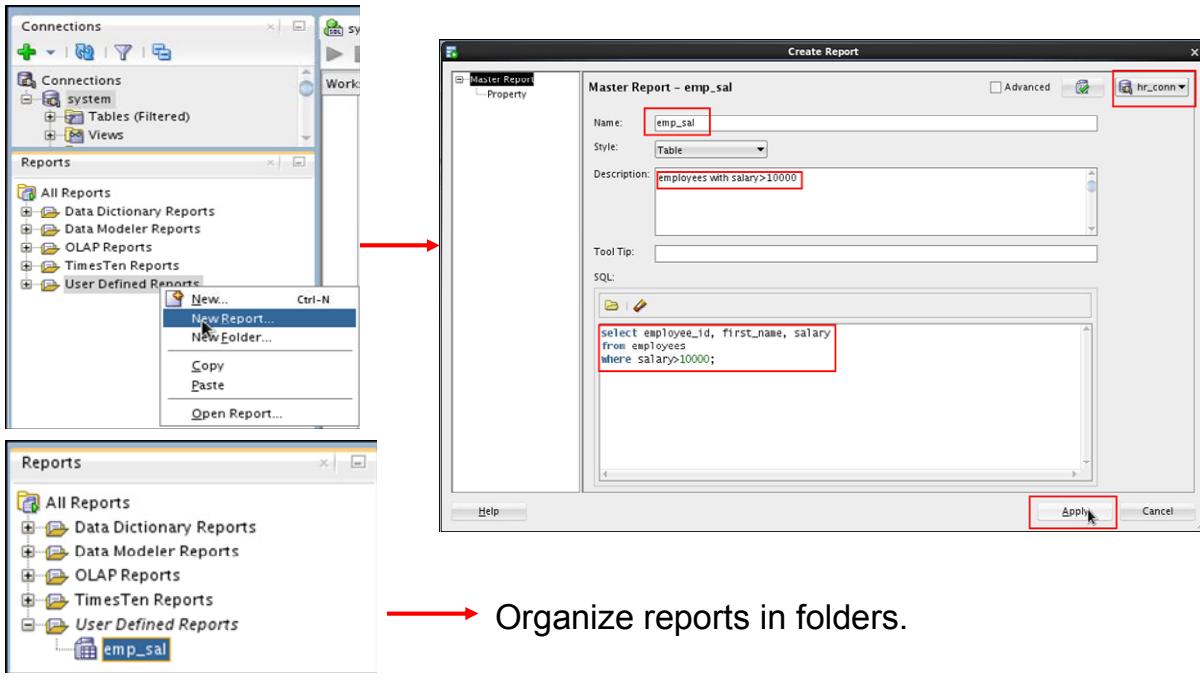
SQL Developer provides many reports about the database and its objects. These reports can be grouped into the following categories:

- About Your Database reports
- Database Administration reports
- Table reports
- PL/SQL reports
- Security reports
- XML reports
- Jobs reports
- Streams reports
- All Objects reports
- Data Dictionary reports
- User-Defined reports

To display reports, click the Reports tab at the left of the window. Individual reports are displayed in tabbed panes at the right of the window; and for each report, you can select (using a drop-down list) the database connection for which to display the report. For reports about objects, the objects shown are only those visible to the database user associated with the selected database connection, and the rows are usually ordered by Owner. You can also create your own user-defined reports.

Creating a User-Defined Report

Create and save user-defined reports for repeated use.



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

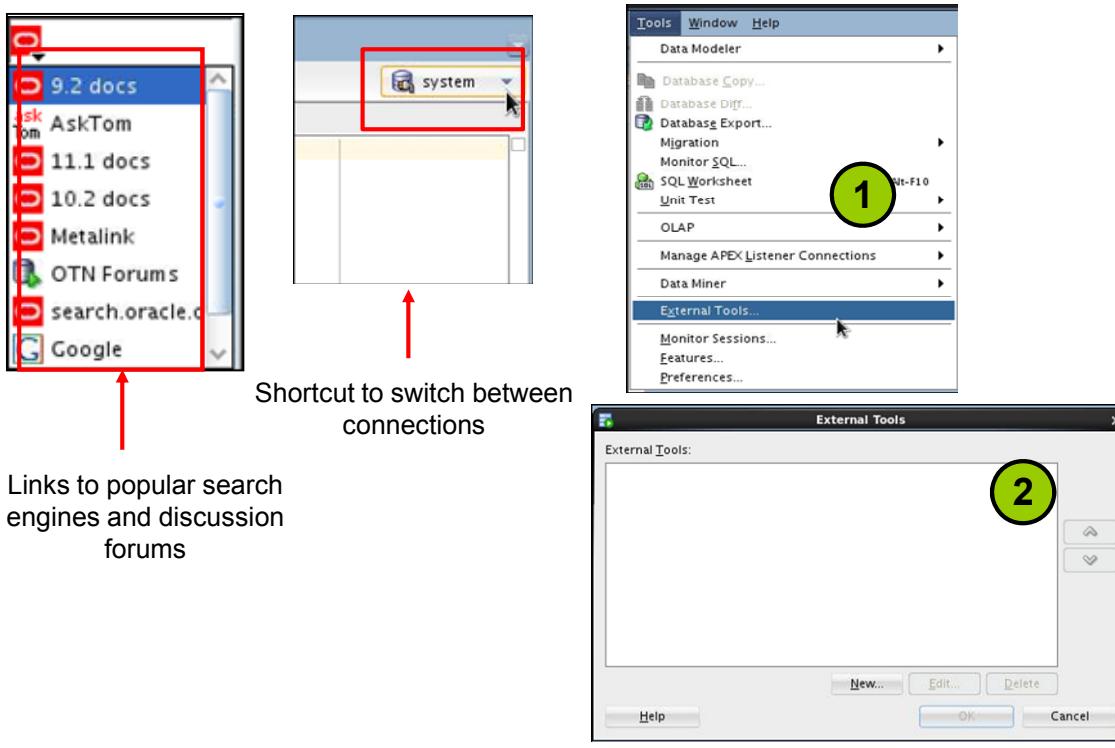
User-defined reports are created by SQL Developer users. To create a user-defined report:

1. Right-click the **User Defined Reports** node under Reports, and select **Add Report**.
2. In the Create Report dialog box, specify the report name and the SQL query to retrieve information for the report. Then click **Apply**.

In the example in the slide, the report name is specified as `emp_sal`. An optional description is provided indicating that the report contains details of employees with `salary >= 10000`. The complete SQL statement for retrieving the information to be displayed in the user-defined report is specified in the SQL box. You can also include an optional tooltip to be displayed when the cursor stays briefly over the report name in the Reports navigator display. And you can also select the connection from the connection drop-down list.

You can organize user-defined reports in folders, and you can create a hierarchy of folders and subfolders. To create a folder for user-defined reports, right-click the User Defined Reports node or any folder name under that node and select **New Folder**. Information about user-defined reports, including any folders for these reports, is stored in a file named `UserReports.xml` under the directory for user-specific information.

Search Engines and External Tools



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

ORACLE

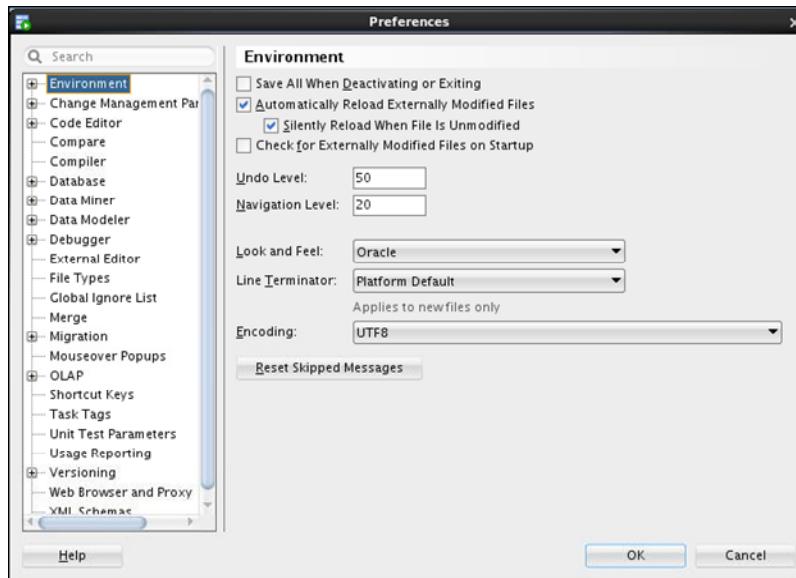
To enhance productivity of SQL developers, SQL Developer has added quick links to popular search engines and discussion forums, such as AskTom, Google, and so on. Also, you have shortcut icons to some of the frequently used tools, such as Notepad, Microsoft Word, and Dreamweaver.

You can add external tools to the existing list or even delete shortcuts to tools that you do not use frequently. To do so:

1. From the Tools menu, select External Tools.
2. In the External Tools dialog box, select New to add new tools. Select Delete to remove any tool from the list.

Setting Preferences

- Customize the SQL Developer interface and environment.
- In the Tools menu, select Preferences.



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You can customize many aspects of the SQL Developer interface and environment by modifying SQL Developer preferences according to your preferences and needs. To modify SQL Developer preferences, select **Tools**, then **Preferences**.

The preferences are grouped into the following categories:

- Environment
- Change Management parameter
- Code Editors
- Compare and Merge
- Database
- Data Miner
- Data Modeler
- Debugger
- Migration
- OLAP
- Versioning

Resetting the SQL Developer Layout



The screenshot shows a terminal window titled "oracle@EDRSR19P1:~/sqldeveloper/system4.0.0.13.80/o.ide.12.1.3.0.4". The window contains the following command-line session:

```
[oracle@EDRSR19P1 Desktop]$ locate windowlayout.xml
/home/oracle/.sqldeveloper/system4.0.0.13.80/o.ide.12.1.3.0.41.131202.1730/windowinglayoutDefault.xml
/home/oracle/.sqldeveloper/system4.0.0.13.80/o.ide.12.1.3.0.41.131202.1730/windowinglayoutDefault.xml
[oracle@EDRSR19P1 Desktop]$ ^C
[oracle@EDRSR19P1 Desktop]$ cd /home/oracle/.sqldeveloper/system4.0.0.13.80/o.ide.12.1.3.0.41.131202.1730
[oracle@EDRSR19P1 o.ide.12.1.3.0.41.131202.1730]$ ls
Debugging.layout  Editing.layout  runStatus.xml
dtcache.xml      projects        windowinglayoutDefault.xml
[oracle@EDRSR19P1 o.ide.12.1.3.0.41.131202.1730]$ rm windowinglayoutDefault.xml
```

ORACLE

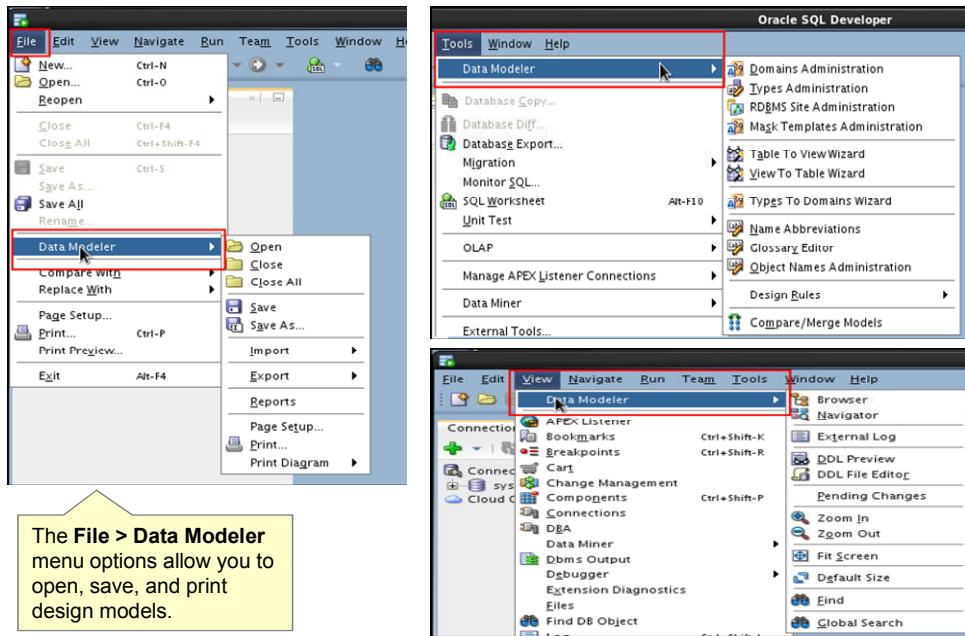
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

While working with SQL Developer, if the Connections navigator disappears or if you cannot dock the Log window in its original place, perform the following steps to fix the problem:

1. Exit from SQL Developer.
2. Open a terminal window and use the locate command to find the location of windowinglayout.xml.
3. Go to the directory that has windowinglayout.xml and delete it.
4. Restart SQL Developer.

Data Modeler in SQL Developer

SQL Developer includes an integrated version of SQL Developer Data Modeler.



The **Tools > Data Modeler** menu options provide the administration and wizard options.

The **View > Data Modeler** menu options provide navigation and view options.

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Using the integrated version of SQL Developer Data Modeler, you can:

- Create, open, import, and save a database design
- Create, modify, and delete Data Modeler objects

To display Data Modeler in a pane, click Tools, then Data Modeler. The Data Modeler menu under Tools includes additional commands, for example, enabling you to specify design rules and preferences.

Summary

In this appendix, you should have learned how to use SQL Developer to do the following:

- Browse, create, and edit database objects
- Execute SQL statements and scripts in SQL Worksheet
- Create and save custom reports
- Browse the Data Modeling options in SQL Developer



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

SQL Developer is a free graphical tool to simplify database development tasks. Using SQL Developer, you can browse, create, and edit database objects. You can use SQL Worksheet to run SQL statements and scripts. SQL Developer enables you to create and save your own special set of reports for repeated use.

Q

Using SQL*Plus

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Objectives

After completing this appendix, you should be able to do the following:

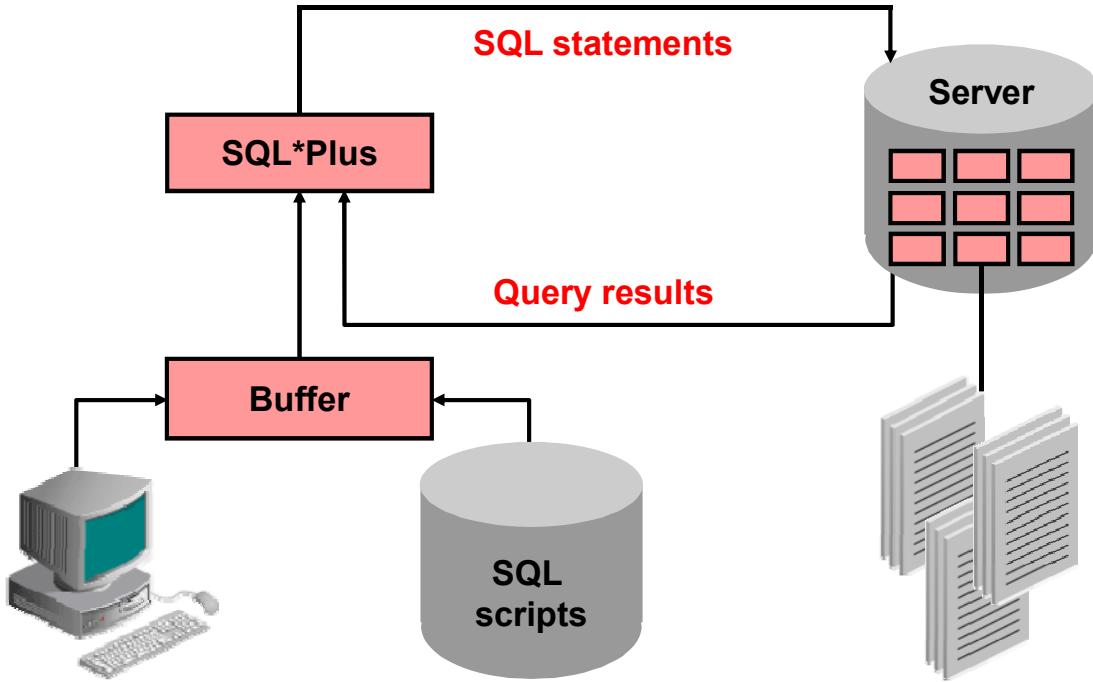
- Log in to SQL*Plus
- Edit SQL commands
- Format the output by using SQL*Plus commands
- Interact with script files



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You might want to create `SELECT` statements that can be used again and again. This appendix covers the use of SQL*Plus commands to execute SQL statements. You learn how to format output by using SQL*Plus commands, edit SQL commands, and save scripts in SQL*Plus.

SQL and SQL*Plus Interaction



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

SQL is a command language used for communication with the Oracle server from any tool or application. Oracle SQL contains many extensions. When you enter a SQL statement, it is stored in a part of memory called the *SQL buffer* and remains there until you enter a new SQL statement. SQL*Plus is an Oracle tool that recognizes and submits SQL statements to the Oracle9i Server for execution. It contains its own command language.

Features of SQL

- Can be used by a range of users, including those with little or no programming experience
- Is a nonprocedural language
- Reduces the amount of time required for creating and maintaining systems
- Is an English-like language

Features of SQL*Plus

- Accepts ad hoc entry of statements
- Accepts SQL input from files
- Provides a line editor for modifying SQL statements
- Controls environmental settings
- Formats query results into basic reports
- Accesses local and remote databases

SQL Statements Versus SQL*Plus Commands

SQL

- A language
- ANSI-standard
- Keywords cannot be abbreviated.
- Statements manipulate data and table definitions in the database.

SQL*Plus

- An environment
- Oracle-proprietary
- Keywords can be abbreviated.
- Commands do not allow manipulation of values in the database.



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

SQL Versus SQL*Plus

SQL	SQL*Plus
Is a language for communicating with the Oracle server to access data	Recognizes SQL statements and sends them to the server
Is based on American National Standards Institute (ANSI)-standard SQL	Is the Oracle-proprietary interface for executing SQL statements
Manipulates data and table definitions in the database	Does not allow manipulation of values in the database
Is entered into the SQL buffer on one or more lines	Is entered one line at a time, not stored in the SQL buffer
Does not have a continuation character	Uses a dash (-) as a continuation character if the command is longer than one line
Cannot be abbreviated	Can be abbreviated
Uses a termination character to execute commands immediately	Does not require termination characters; executes commands immediately
Uses functions to perform some formatting	Uses commands to format data



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The table in the slide compares SQL and SQL*Plus.

Using SQL*Plus

- Log in to SQL*Plus.
- Describe the table structure.
- Edit your SQL statement.
- Execute SQL from SQL*Plus.
- Save SQL statements to files and append SQL statements to files.
- Execute saved files.
- Load commands to be edited from the file to the buffer.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

SQL*Plus

SQL*Plus is an environment in which you can:

- Execute SQL statements to retrieve, modify, add, and remove data from the database
- Format, perform calculations on, store, and print query results in the form of reports
- Create script files to store SQL statements for repeated use in the future

SQL Plus Commands: Categories

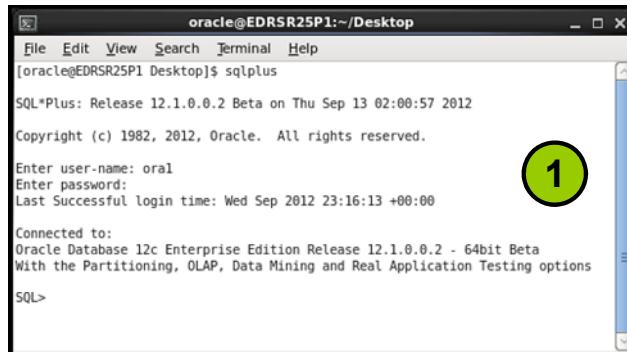
Category	Purpose
Environment	Affect the general behavior of SQL statements for the session.
Format	Format query results.
File manipulation	Save, load, and run script files.
Execution	Send SQL statements from the SQL buffer to the Oracle server.
Edit	Modify SQL statements in the buffer.
Interaction	Create and pass variables to SQL statements, print variable values, and print messages to the screen.
Miscellaneous	Connect to the database, manipulate the SQL*Plus environment, and display column definitions.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

SQL*Plus commands can be divided into seven main categories as listed in the slide.

Logging In to SQL*Plus



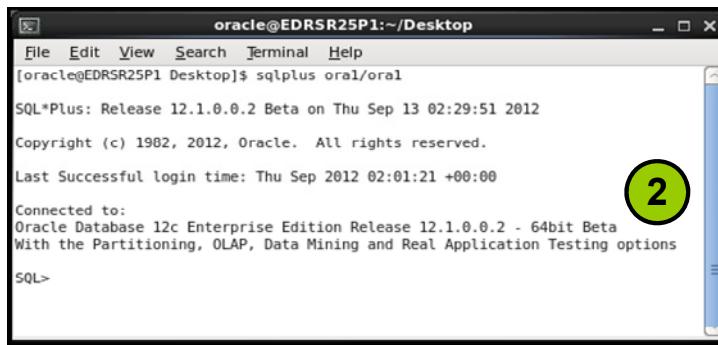
```
oracle@EDRSR25P1:~/Desktop
[oracle@EDRSR25P1 Desktop]$ sqlplus
SQL*Plus: Release 12.1.0.0.2 Beta on Thu Sep 13 02:00:57 2012
Copyright (c) 1982, 2012, Oracle. All rights reserved.

Enter user-name: oral
Enter password:
Last Successful login time: Wed Sep 2012 23:16:13 +00:00

Connected to:
Oracle Database 12c Enterprise Edition Release 12.1.0.0.2 - 64bit Beta
With the Partitioning, OLAP, Data Mining and Real Application Testing options

SQL>
```

```
sqlplus [username[/password[@database]]]
```



```
oracle@EDRSR25P1:~/Desktop
[oracle@EDRSR25P1 Desktop]$ sqlplus oral/oral
SQL*Plus: Release 12.1.0.0.2 Beta on Thu Sep 13 02:29:51 2012
Copyright (c) 1982, 2012, Oracle. All rights reserved.

Last Successful login time: Thu Sep 2012 02:01:21 +00:00

Connected to:
Oracle Database 12c Enterprise Edition Release 12.1.0.0.2 - 64bit Beta
With the Partitioning, OLAP, Data Mining and Real Application Testing options

SQL>
```

ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

How you invoke SQL*Plus depends on which type of operating system you are running Oracle Database on.

To log in from a Linux environment:

1. Right-click your Linux desktop and select terminal.
2. Enter the `sqlplus` command shown in the slide.
3. Enter the username, password, and database name.

In the syntax:

<code>username</code>	Your database username
<code>password</code>	Your database password (Your password is visible if you enter it here.)
<code>@database</code>	The database connect string

Note: To ensure the integrity of your password, do not enter it at the operating system prompt. Instead, enter only your username. Enter your password at the password prompt.

Displaying the Table Structure

Use the SQL*Plus DESCRIBE command to display the structure of a table:

```
DESC [RIBE] tablename
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In SQL*Plus, you can display the structure of a table by using the DESCRIBE command. The result of the command is a display of column names and data types as well as an indication of whether a column must contain data.

In the syntax:

`tablename` The name of any existing table, view, or synonym that is accessible to the user

To describe the DEPARTMENTS table, use this command:

```
SQL> DESCRIBE DEPARTMENTS
      Name          Null?    Type
-----  -----
DEPARTMENT_ID      NOT NULL NUMBER(4)
DEPARTMENT_NAME    NOT NULL VARCHAR2(30)
MANAGER_ID         NUMBER(6)
LOCATION_ID        NUMBER(4)
```

Displaying the Table Structure

```
DESCRIBE departments
```

Name	Null	Type
DEPARTMENT_ID	NOT NULL	NUMBER (4)
DEPARTMENT_NAME	NOT NULL	VARCHAR2 (30)
MANAGER_ID		NUMBER (6)
LOCATION_ID		NUMBER (4)



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The example code in the slide displays the information about the structure of the DEPARTMENTS table. In the result:

Null?: Specifies whether a column must contain data (NOT NULL indicates that a column must contain data.)

Type: Displays the data type for a column

SQL*Plus Editing Commands

Command	Description
A [PPEND] text	Adds text to the end of the current line
C [HANGE] / old / new	Changes old text to new in the current line
C [HANGE] / text /	Deletes text from the current line
CL [EAR] BUFF [ER]	Deletes all lines from the SQL buffer
DEL	Deletes current line
DEL n	Deletes line n
DEL m n	Deletes lines <i>m</i> to <i>n</i> inclusive



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

SQL*Plus commands are entered one line at a time and are not stored in the SQL buffer.

Guidelines

- If you press Enter before completing a command, SQL*Plus prompts you with a line number.
- You terminate the SQL buffer either by entering one of the terminator characters (semicolon or slash) or by pressing Enter twice. The SQL prompt then appears.

SQL*Plus Editing Commands

Command	Description
I[NPUT]	Inserts an indefinite number of lines
I[NPUT] text	Inserts a line consisting of text
L[IST]	Lists all lines in the SQL buffer
L[IST] n	Lists one line (specified by n)
L[IST] m n	Lists a range of lines (m to n) inclusive
R[UN]	Displays and runs the current SQL statement in the buffer
n	Specifies the line to make the current line
n text	Replaces line n with text
0 text	Inserts a line before line 1



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Note: You can enter only one SQL*Plus command for each SQL prompt. SQL*Plus commands are not stored in the buffer. To continue a SQL*Plus command on the next line, end the first line with a hyphen (-).

Using LIST, n, and APPEND

```
LIST
1  SELECT last_name
2* FROM employees
```

```
1
1* SELECT last_name
```

```
A , job_id
1* SELECT last_name, job_id
```

```
LIST
1  SELECT last_name, job_id
2* FROM employees
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

- Use the L [IST] command to display the contents of the SQL buffer. The asterisk (*) beside line 2 in the buffer indicates that line 2 is the current line. Any edits that you made apply to the current line.
- Change the number of the current line by entering the number (n) of the line that you want to edit. The new current line is displayed.
- Use the A [PPEND] command to add text to the current line. The newly edited line is displayed. Verify the new contents of the buffer by using the LIST command.

Note: Many SQL*Plus commands, including LIST and APPEND, can be abbreviated to just their first letter. LIST can be abbreviated to L; APPEND can be abbreviated to A.

Using the CHANGE Command

```
LIST
```

```
1* SELECT * from employees
```

```
c/employees/departments
```

```
1* SELECT * from departments
```

```
LIST
```

```
1* SELECT * from departments
```

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

- Use `L[IST]` to display the contents of the buffer.
- Use the `C[HANGE]` command to alter the contents of the current line in the SQL buffer. In this case, replace the `employees` table with the `departments` table. The new current line is displayed.
- Use the `L[IST]` command to verify the new contents of the buffer.

SQL*Plus File Commands

Command	Description
SAV [E] filename [.ext] [REP [LACE]APP [END]]	Saves the current contents of the SQL buffer to a file. Use APPEND to add to an existing file; use REPLACE to overwrite an existing file. The default extension is .sql.
GET filename [.ext]	Writes the contents of a previously saved file to the SQL buffer. The default extension for the file name is .sql.
STA[RT] filename [.ext]	Runs a previously saved command file
@ filename	Runs a previously saved command file (same as START)
ED [IT]	Invokes the editor and saves the buffer contents to a file named afiedt.buf
ED [IT] [filename [.ext]]	Invokes the editor to edit the contents of a saved file
SPO [OL] [filename [.ext]] OFF OUT	Stores query results in a file. OFF closes the spool file. OUT closes the spool file and sends the file results to the printer.
Exit	Exits SQL Plus



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

SQL statements communicate with the Oracle server. SQL*Plus commands control the environment, format query results, and manage files. You can use the commands described in the table as shown in the slide.

Using the SAVE and START Commands

```
LIST
```

```
1  SELECT last_name, manager_id, department_id
2* FROM employees
```

```
SAVE my_query
```

```
Created file my_query
```

```
START my_query
```

```
LAST_NAME
```

```
MANAGER_ID DEPARTMENT_ID
```

```
-----
```

```
King
```

```
90
```

```
Kochhar
```

```
100
```

```
90
```

```
...
```

```
107 rows selected.
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

SAVE

Use the `SAVE` command to store the current contents of the buffer in a file. In this way, you can store frequently used scripts for use in the future.

START

Use the `START` command to run a script in SQL*Plus. Alternatively, you can use the symbol `@` to run a script:

```
@my_query
```

SERVEROUTPUT Command

- Use the SET SERVEROUT [PUT] command to control whether to display the output of stored procedures or PL/SQL blocks in SQL*Plus.
- The DBMS_OUTPUT line length limit is increased from 255 bytes to 32767 bytes.
- The default size is now unlimited.
- Resources are not preallocated when SERVEROUTPUT is set.
- Because there is no performance penalty, use UNLIMITED unless you want to conserve physical memory.

```
SET SERVEROUT [PUT] {ON | OFF} [SIZE {n | UNL[IMITED]}]
[FOR [MAT] {WRA[PPED] | WOR[D_WWRAPPED] | TRU[NCATED]}]
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Most of the PL/SQL programs perform input and output through SQL statements to store data in database tables or query those tables. All other PL/SQL input/output is done through APIs that interact with other programs. For example, the DBMS_OUTPUT package has procedures such as PUT_LINE. To see the result outside of PL/SQL, you require another program, such as SQL*Plus, to read and display the data passed to DBMS_OUTPUT.

SQL*Plus does not display DBMS_OUTPUT data unless you first issue the SQL*Plus command SET SERVEROUTPUT ON as follows:

```
SET SERVEROUTPUT ON
```

Note

- SIZE sets the number of bytes of the output that can be buffered within the Oracle Database server. The default is UNLIMITED. *n* cannot be less than 2000 or greater than 1,000,000.
- For additional information about SERVEROUTPUT, see the *Oracle Database PL/SQL User's Guide and Reference 12c*.

Using the SQL*Plus SPOOL Command

```
SPO [OL] [file_name[.ext]] [CRE [ATE] | REP [LACE] |
APP [END]] | OFF | OUT]
```

Option	Description
file_name [.ext]	Spools output to the specified file name
CRE [ATE]	Creates a new file with the name specified
REP [LACE]	Replaces the contents of an existing file. If the file does not exist, REPLACE creates the file.
APP [END]	Adds the contents of the buffer to the end of the file that you specify
OFF	Stops spooling
OUT	Stops spooling and sends the file to your computer's standard (default) printer



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

The SPOOL command stores query results in a file or optionally sends the file to a printer. The SPOOL command has been enhanced. You can now append to, or replace an existing file, where previously you could only use SPOOL to create (and replace) a file. REPLACE is the default.

To spool the output generated by commands in a script without displaying the output on the screen, use SET TERMOUT OFF. SET TERMOUT OFF does not affect output from commands that run interactively.

You must use quotation marks around file names containing white space. To create a valid HTML file using SPOOL APPEND commands, you must use PROMPT or a similar command to create the HTML page header and footer. The SPOOL APPEND command does not parse HTML tags. Set SQLPLUSCOMPAT [IBILITY] to 9.2 or earlier to disable the CREATE, APPEND, and SAVE parameters.

Using the AUTOTRACE Command

- It displays a report after the successful execution of SQL data manipulation statements (DML) statements such as SELECT, INSERT, UPDATE, or DELETE.
- The report can now include execution statistics and the query execution path.

```
SET AUTOT [RACE] {ON | OFF | TRACE [ONLY] } [EXP [LAIN] ]  
[STATISTICS]
```

```
SET AUTOTRACE ON  
-- The AUTOTRACE report includes both the optimizer  
-- execution path and the SQL statement execution  
-- statistics
```



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

EXPLAIN shows the query execution path by performing an EXPLAIN PLAN. STATISTICS displays SQL statement statistics. The formatting of your AUTOTRACE report may vary depending on the version of the server to which you are connected and the configuration of the server. The DBMS_XPLAN package provides an easy way to display the output of the EXPLAIN PLAN command in several predefined formats.

Note

- For additional information about the package and subprograms, see the *Oracle Database PL/SQL Packages and Types Reference 12c* guide.
- For additional information about the EXPLAIN PLAN, see *Oracle Database SQL Reference 12c*.
- For additional information about execution plans and their statistics, see the *Oracle Database Performance Tuning Guide 12c*.

Summary

In this appendix, you should have learned how to use SQL*Plus as an environment to do the following:

- Execute SQL statements
- Edit SQL statements
- Format the output
- Interact with script files



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

SQL*Plus is an execution environment that you can use to send SQL commands to the database server, and to edit and save SQL commands. You can execute commands from the SQL prompt or from a script file.

Designing and Testing Your Code to Avoid SQL Injection Attacks

ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Objectives

After completing this appendix, you should be able to do the following:

- Design immune code
- Test code for SQL injection flaws

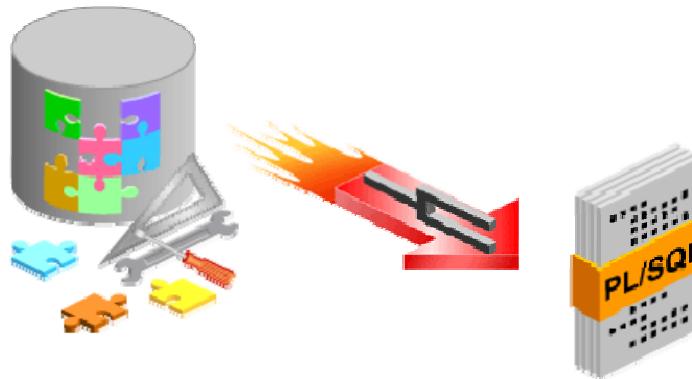


Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

In this appendix, you learn about effective PL/SQL code designing and testing to protect your applications from SQL injection attacks.

Using Bind Arguments

- Most common vulnerability:
 - Dynamic SQL with string concatenation
- Your code design must:
 - Avoid input string concatenation in dynamic SQL
 - Use bind arguments, whether automatically via static SQL or explicitly via dynamic SQL statements



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Poor application design can lead to “designed in” vulnerabilities where there are no coding problems as such and everything works as intended.

However, you must design your code such that it is (ideally) entirely free of SQL injection vulnerabilities, or contains measures that mitigate the impact of a successful attack.

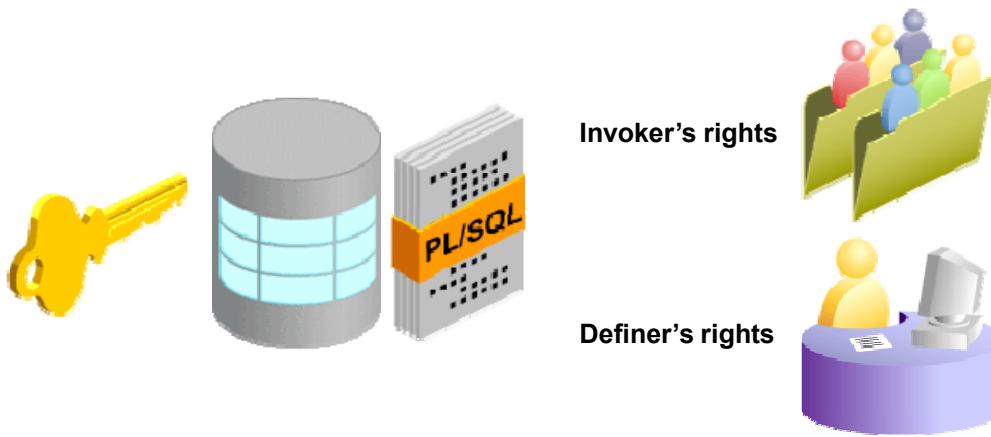
As you have seen with the examples presented thus far, the common flaw of all code vulnerable to SQL injection is the construction of dynamic SQL using string concatenation. Complete immunity from SQL injection attack can be achieved only through the elimination of input string concatenation in dynamic SQL.

- Avoid input string concatenation.
- Use bind arguments, whether automatically via static SQL or explicitly via dynamic SQL statements.

Design your code to use bind arguments wherever possible. The only exceptions should be when you must concatenate identifiers or keywords, because you have no other choice.

Avoiding Privilege Escalation

- Give out privileges appropriately.
- Run code with invoker's rights when possible.
- Ensure that the database privilege model is upheld when using definer's rights.



ORACLE

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Unless carefully designed, routines may effectively grant users more privileges than intended. Wherever possible, run code with invoker's rights to minimize the scope for privilege escalation attacks and to mitigate the impact of a successful SQL injection attack.

Where this is not possible, the routines that are run with definer's rights should be carefully reviewed to ensure that the database privilege model is upheld.

If none of the methods of execution (definer's rights or invoker's rights) appears suitable, consider implementing specific bypass checks for the duration of the call.

Beware of Filter Parameters

- Filter parameter:
 - P_WHERE_CLAUSE is a filter.
 - It is difficult to protect against SQL injection.

```
stmt := 'SELECT session_id FROM sessions
        WHERE' || p_where_clause;
```

- Prevention methods:
 - Do not specify APIs that allow arbitrary query parameters to be exposed.
 - Any existing APIs with this type of functionality must be deprecated and replaced with safe alternatives.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Occasionally, routines may require a parameter that is used to form an expression in a query or other PL/SQL statements. Typically, such parameters are referred to as “filters.”

How do you test that the provided expression is free from SQL injection? You cannot. In these cases, the only sure solution is prevention.

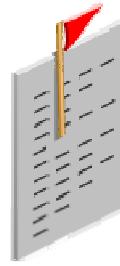
- Do not specify APIs that allow arbitrary query parameters to be exposed.
- Any existing APIs with this type of functionality must be deprecated and replaced with safe alternatives.

If you must make use of filter parameters, contact your security compliance team for approval.

Any such routines must be registered as possible sinks of dynamic SQL and recorded in static code analysis rulesets. This helps to ensure that any users of these APIs are well behaved. Contact your security compliance team to register these routines.

Trapping and Handling Exceptions

- Design your code to trap and handle exceptions appropriately.
- Before deploying your application:
 - Remove all code tracing
 - Remove all debug messages



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Normal SQL injection attacks depend, in large measure, on an attacker's reverse-engineering portions of the original SQL query by using information gained from error messages. Therefore, keep application error messages succinct and do not divulge metadata information (such as column names and table names).

Note: From Oracle Database 10.2 and later, you can use PL/SQL conditional compilation for managing self-tracing code, asserts, and so on.

Coding Review and Testing Strategy

- Test:
 - Dynamic testing
 - Static testing
- Review:
 - Peer and self reviews
 - Analysis tools



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

You can use a number of strategies to test for SQL injection vulnerability. Using a combination of these strategies should be regarded as a sensible minimum to get some degree of confidence in freedom from vulnerabilities.

Effective testing of complex products is essentially a process of investigation, and not merely a matter of creating and following routine procedure. Code reviews or walk-throughs are referred to as “static testing,” whereas actually running the program with a given set of test cases in a given development stage is often referred to as “dynamic testing.”

Testing for SQL injection flaws requires both static and dynamic testing. For static testing, you can begin with a peer (or self) code review and/or make use of a static code analysis tool. After finding and fixing the semantical SQL injection bugs, you must perform dynamic testing by using tools that generate random input (fuzzing), and also run through test cases that you define specifically for SQL injection detection within your code.

Reviewing Code

Language	Look for...
PL/SQL	<code>EXECUTE IMMEDIATE OPEN cursor_var FOR ... DBMS_SQL DBMS_SYS_SQL</code>
C	<p>String substitutions such as:</p> <pre>static const oratext createsq[] = "CREATE SEQUENCE \"%.*s\".\"%.*s\" start with %.*s increment by %.*s";</pre> <p>Followed by usage such as:</p> <pre>DISCARD lstrprintf(sql_txt, createsq, ownerl, owner, seqnaml, seqnam, sizeof(start), start, sinc_byl, sincrement_by);</pre>
Java	<p>String concatenations such as:</p> <pre>sqltext = "DROP VIEW " + this.username + "." + this.viewName;</pre>

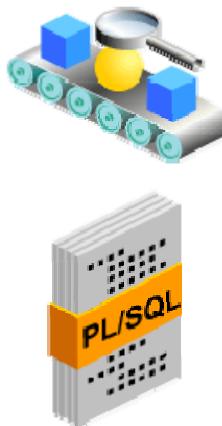
Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

When reviewing code, first identify all dynamic SQL statements. Depending on your programming language, some key indicators that dynamic SQL is in use are shown in the slide.

Next, check to make sure that bind arguments are used in all possible and practical instances. Where bind arguments cannot be used, make sure that the correct routines are used to filter and sanitize inputs.

Running Static Code Analysis

- Generally performed by an automated tool
- Can be performed on some versions of the source code
- Can be performed on some forms of the object code
- Should be used as one of the initial steps of testing code



ORACLE®

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

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.

Because SQL injections arise from dynamically generated SQL, static code analysis tools may have a hard job identifying all categories of SQL injection with some certainty, because this may require knowledge of object names and contents not available at analysis time. For example, a PL/SQL function might construct a string containing parts of a SQL statement that might then be combined with a malicious table name, thereby leading to an injection.

Static code analysis tools should not be used for any kind of security sign-off. Instead, it should be one of the initial steps in the code testing process.

Testing with Fuzzing Tools

- Fuzzing tools:
 - Is a software testing technique that provides random data (“fuzz”) to the inputs of a program
 - Can enhance software security and software safety, because it often finds odd oversights and defects that human testers would fail to find
 - Must not be used as a substitute for exhaustive testing or formal methods
- Although tools can be used to automate fuzz testing, it may be necessary to customize the test data and application context to get the best results from such tools.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

Fuzz testing or fuzzing is a software testing technique that provides random data (“fuzz”) to the inputs of a program. If the program fails (for example, by crashing or by failing built-in code assertions), the defects can be noted. The great advantage of fuzz testing is that the test design is extremely simple and free of preconceptions about system behavior.

Fuzz testing is thought to enhance software security and software safety, because it often finds odd oversights and defects that human testers would fail to find, and even careful human test designers would fail to create tests for.

However, fuzz testing is not a substitute for exhaustive testing or formal methods; it can only provide a random sample of the system’s behavior, and in many cases, passing a fuzz test may only demonstrate that a piece of software handles exceptions without crashing, rather than behaving correctly. Thus, fuzz testing can be regarded only as a bug-finding tool rather than an assurance of quality.

Although tools can be used to automate fuzz testing, it may be necessary to customize the test data and application context to get the best results from such tools.

Generating Test Cases

- Test each input parameter individually.
- When testing each parameter, leave all the other parameters unchanged with valid data as their arguments.
- Omitting parameters or supplying bad arguments to other parameters while you are testing another for SQL injection can break the application in ways that prevent you from determining whether SQL injection is possible.
- Always use the full parameter line, supplying every parameter, except the one that you are testing, with a legitimate value.
- Certain object and schema names help you uncover SQL injection vulnerabilities.



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

When designing SQL injection test cases, keep in mind that each input parameter must be tested individually. When testing each parameter, leave all the other parameters unchanged with valid data as their arguments.

It can be tempting to delete everything that you are not working with to make things appear simpler, particularly with applications that have parameter lines going into many thousands of characters. Omitting parameters or supplying bad arguments to other parameters while you are testing another for SQL injection can break the application in ways that prevent you from determining whether SQL injection is possible.

So, when testing for SQL injection, always use the full parameter line, supplying every parameter, except the one that you are testing, with a legitimate value.

Certain object and schema names help you uncover SQL injection vulnerabilities. The following page provides a list of names as input values in your tests.

Name	Test
AAAAAAAAAAABB BBBBBBCCCCCCCC	Maximum size identifiers should be tested as both schema and object names. This checks that temporary object names can be created.
AAAAAAAAAAAAAA AAAAAAAAAAAAAA AA & AAAAAAAAAAAAAA AAAAAAAAAAAAAA A	Maximum and “nearly” maximum object names should be handled as separate objects. This checks that any name truncation operation does not result in non-unique names.
schema1.a schema2.a	Ensures that objects with the same name in different schemas are properly differentiated
AAA "aaa"	Ensures that objects with different case names are clearly differentiated
object@dblink	Ensures that link names (both valid and invalid) do not defeat security tests; also ensures that links are rejected unless specifically allowable
Rename "ABC" to "XYZ"	Renames objects during tests to ensure that applications continue to function correctly
"quoted"	Ensures that quoted schema and object names can be handled
"AAA" " AAA" "AAA "	Ensures that quoted objects do not lead to second-order injections
"\$IF"	Refers to a variable and not a preprocessor directive
"BEGIN"	Refers to an object and not a reserved word
"A%TYPE"	Refers to an object and not an object attribute
"a/*x*/ <>A>>, %s-- %00\AAAA ", "a' '' */ .@\? :x&y} {} (><) [!=;A"	Ensures that special characters in quoted variables do not have other side effects
"xxx" chr(8) "yyy", "xxx" chr(9) "yyy", "xxx" chr(10) "yyy", "xxx" chr(13) "yyy", "xxx" chr(4) "yyy", "xxx" chr(26) "yyy", "xxx" chr(0) "yyy"	Ensures that object names with embedded control characters are handled correctly, and checks that all output correctly distinguishes these objects, especially any output written to files. Note that some object names may be (correctly) rejected as invalid syntax.
NULL	Null and empty strings may behave differently on different Oracle Database ports or versions.

Summary

In this appendix, you should have learned how to:

- Design immune code
- Test code for SQL injection flaws



Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

This appendix showed you the effective ways of PL/SQL code designing and testing to protect your applications from SQL injection attacks.

THESE eKIT MATERIALS ARE FOR YOUR USE IN THIS CLASSROOM ONLY. COPYING eKIT MATERIALS FROM THIS COMPUTER IS STRICTLY PROHIBITED

Oracle University and Error : You are not a Valid Partner use only