

Oracle Database 12c: Analytic SQL for Data Warehousing

Student Guide

D79991GC10

Edition 1.0

September 2013

D83926

ORACLE®

Authors

Supriya Ananth

Lauran Serhal

Technical Contributors and Reviewers

Nancy Greenberg

Swarnapriya Shridhar

Hermann Baer

Angela Amor

John Haydu

Laszlo Czinkoczki

Mirella Tumolo

Anjulaponni Aazhagulekshmi

Sailaja Pasupuleti

Milos Randak

Yanti Chang

Charlie Berger

Joel Goodman

Editors

Anwesha Ray

Smita Kommini

Rashmi Rajagopal

Graphic Editor

Rajiv Chandrabhanu

Publishers

Veena Narasimhan

Jayanthy Keshavamurthy

Copyright © 2013, 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

- Objectives 1-2
- Lesson Agenda 1-3
- Course Objectives 1-4
- Course Agenda 1-5
- Lesson Agenda 1-6
- Sample Schemas Used in the Course 1-7
- Human Resources (HR) Schema 1-8
- Order Entry (OE) Schema 1-9
- Sales History (SH) Schema 1-10
- Class Account Information 1-12
- Appendixes in the Course 1-13
- SQL Environments Available in the Course 1-14
- Lesson Agenda 1-15
- What Is Oracle SQL Developer? 1-16
- SQL*Plus: Overview 1-17
- Lesson Agenda 1-18
- What Is Analytic SQL? 1-19
- Analytic SQL in Data Warehouses 1-20
- Case Study: Overview 1-21
- Lesson Agenda 1-22
- Oracle SQL and Data Warehousing Guide Documentation 1-23
- Additional Resources 1-24
- Summary 1-25
- Practice 1 Overview 1-26

2 Grouping and Aggregating Data by Using SQL

- Objectives 2-2
- Lesson Agenda 2-3
- Group Functions 2-4
- GROUP BY Clause 2-5
- HAVING Clause 2-6
- GROUP BY with ROLLUP and CUBE Operators 2-7
- Using the ROLLUP Operator 2-8
- Using the ROLLUP Operator: Example 2-9

Using the CUBE Operator 2-10
Using the CUBE Operator: Example 2-11
Lesson Agenda 2-12
Working with the GROUPING Function 2-13
Working with the GROUPING Function: Example 2-14
Working with the GROUPING_ID Function 2-15
GROUPING SETS 2-16
GROUPING SETS: Example 2 2-19
Lesson Agenda 2-21
Composite Columns 2-22
Composite Columns: Example 2-24
Concatenated Groupings 2-26
Concatenated Groupings: Example 2-27
Quiz 2-28
Summary 2-29
Practice 2 Overview 2-30

3 Hierarchical Retrieval

Objectives 3-2
Using Hierarchical Queries 3-3
Using Hierarchical Queries: Example Sample Data from the EMPLOYEES Table 3-4
Natural Tree Structure 3-5
Hierarchical Queries: Syntax 3-6
Walking the Tree: Specifying the Starting Point 3-7
Walking the Tree: Specifying the Direction of the Query 3-8
Hierarchical Query Example: Using the CONNECT BY Clause 3-9
Specifying the Direction of the Query: From the Top Down 3-10
Specifying the Direction of the Query: From the Bottom Up 3-11
Computation Using the WITH Clause 3-12
Using the CONNECT_BY_ISLEAF Pseudocolumn 3-13
Using the LEVEL Pseudocolumn 3-14
Using the LEVEL Pseudocolumn: Example 3-15
Formatting Hierarchical Reports by Using LEVEL and LPAD 3-17
Pruning Nodes and Branches 3-19
Pruning Branches: Example 1: Eliminating a Node 3-20
Pruning Branches: Example 2: Eliminating a Branch 3-21
Quiz 3-22
Summary 3-23
Practice 3 Overview 3-24

4 Working with Regular Expressions

- Objectives 4-2
- Lesson Agenda 4-3
- What Are Regular Expressions? 4-4
- Benefits of Using Regular Expressions 4-5
- Using the Regular Expressions Functions and Conditions in SQL and PL/SQL 4-6
- What Are Metacharacters? 4-7
- Using Metacharacters with Regular Expressions 4-8
- Lesson Agenda 4-10
- Regular Expressions Functions and Conditions: Syntax 4-11
- Performing a Basic Search Using the REGEXP_LIKE Condition 4-12
- Finding Patterns by Using the REGEXP_INSTR Function 4-13
- Extracting Substrings by Using the REGEXP_SUBSTR Function 4-14
- Replacing Patterns by Using the REGEXP_REPLACE Function 4-15
- Using the REGEXP_COUNT Function 4-16
- Lesson Agenda 4-17
- Working with Subexpressions 4-18
- Using Subexpressions with Regular Expression Support: Example 4-19
- Why Access the nth Subexpression? 4-20
- REGEXP_SUBSTR: Example 4-21
- Regular Expressions and Check Constraints: Examples 4-22
- Quiz 4-23
- Summary 4-24
- Practice 4 Overview 4-25

5 Analyzing and Reporting Data by Using SQL

- Objectives 5-2
- Lesson Agenda 5-3
- SQL for Analysis and Reporting: Overview 5-4
- Using the Analytic Functions 5-5
- Concepts Used in Analytic Functions 5-6
- Lesson Agenda 5-8
- Using the Ranking Functions 5-9
- Working with the RANK Function 5-10
- Using the RANK Function: Example 5-11
- Per-Group Ranking 5-12
- Per-Group Ranking: Example 5-13
- RANK and DENSE_RANK Functions: Example 5-14
- Per-Cube and Rollup Group Ranking 5-15
- Using CUME_DIST and PERCENT_RANK Functions 5-16

Using the CUME_DIST Function: Example 5-17
Using the PERCENT_RANK Function: Example 5-18
Using the NTILE Function: Example 5-19
Lesson Agenda 5-20
Using the RATIO_TO_REPORT Function 5-21
Using the RATIO_TO_REPORT Function: Example 5-22
Lesson Agenda 5-23
Using the LAG and LEAD Analytic Functions 5-24
Using the LAG and LEAD Analytic Functions: Example 5-25
Using the FIRST_VALUE and LAST_VALUE Functions 5-26
Using the NTH_VALUE Function 5-27
Using the NTH_VALUE Function: Example 5-28
Lesson Agenda 5-29
Using the LISTAGG Function 5-30
Using the LISTAGG Function: Example 5-31
Lesson Agenda 5-32
SQL Row-Limiting Clause 5-33
SQL Row-Limiting Clause: Example 5-34
Quiz 5-35
Summary 5-36
Practice 5 Overview 5-37

6 Performing Pivoting and Unpivoting Operations

Objectives 6-2
Benefits of Using Pivoting Operations 6-3
PIVOT and UNPIVOT Clauses of the SELECT Statement 6-4
Pivoting on the QUARTER Column: Conceptual Example 6-5
PIVOT Clause Syntax 6-6
Creating a New View: Example 6-8
Selecting the SALES_VIEW Data 6-10
Pivoting the QUARTER Column in the SH Schema: Example 6-12
Pivoting on Multiple Columns 6-14
Pivoting Using Multiple Aggregations 6-16
Distinguishing PIVOT-Generated NULLs from NULLs in the Source Data 6-17
Using the XML Keyword to Specify Pivot Values: Two Methods 6-19
Specifying Pivot Values: Using the ANY Keyword 6-20
Specifying Pivot Values: Using Subqueries 6-21
Unpivoting the QUARTER Column: Conceptual Example 6-22
Using the UNPIVOT Operator 6-23
Using the UNPIVOT Clause 6-24
Data Types of the Value Columns in an UNPIVOT Operation 6-25

UNPIVOT Clause Syntax	6-26
Creating a New Pivot Table: Example	6-27
Unpivoting the QUARTER Column in the SH Schema: Example	6-28
Unpivoting Multiple Columns in the SH Schema: Example	6-29
Unpivoting Multiple Aggregations in the SH Schema: Example	6-31
Quiz	6-33
Summary	6-34
Practice 6 Overview	6-35

7 Pattern Matching by Using SQL

Objectives	7-2
Lesson Agenda	7-3
Benefits of Pattern Matching	7-4
Pattern Matching: Overview	7-5
Keywords in Pattern Matching	7-6
Pattern Matching: Example	7-7
Using the PARTITION BY Clause	7-9
Using [ONE ROW ALL ROWS] PER MATCH Keywords	7-10
Defining Row Pattern Measure Columns	7-11
Defining the Row Pattern to be Matched	7-12
Defining Primary Pattern Variables	7-14
Restarting the Matching Process	7-15
Lesson Agenda	7-16
Using Scalar Expressions in Pattern Matching	7-17
Scalar Expressions in Pattern Matching: Example	7-18
Lesson Agenda	7-19
Row Pattern Column References	7-20
Lesson Agenda	7-21
Running Versus Final Semantics and Keywords	7-22
Lesson Agenda	7-23
Handling Empty Matches and Unmatched Rows	7-24
Excluding Patterns from the Output	7-25
Lesson Agenda	7-26
Input Table Requirements	7-27
Nesting in the MATCH_RECOGNIZE Clause	7-28
Quiz	7-29
Summary	7-30
Practice 7 Overview	7-31

8 Modeling Data by Using SQL

Objectives 8-2

Lesson Agenda 8-3

Benefits of Integrating Interrow Calculations in SQL 8-4

Why Use SQL Modeling? 8-5

Partitions, Measures, and Dimensions 8-6

Learning the Concepts 8-7

Learning the Concepts: Example 8-8

Reviewing the Sample Data 8-9

MODEL Syntax: Example 8-10

Cell References 8-12

Positional and Symbolic Cell References 8-13

Using Cell Reference: Example 8-14

Range References 8-15

Complete Query: Example 8-16

Lesson Agenda 8-17

CV() Function 8-18

CV()Function: Example 8-19

Using the FOR Construct with IN List Operator 8-20

Using the FOR Construct with Incremental Values 8-22

Using the FOR Construct with a Subquery 8-23

Using Analytic Functions in the SQL MODEL Clause 8-24

Distinguishing Missing Cells from NULLs 8-25

Distinguishing Missing Cells from NULLs: Using Conditions, Functions, and Clauses 8-26

Using the IS PRESENT Condition: Example 8-27

Lesson Agenda 8-29

Using the UPDATE, UPSERT, and UPSERT ALL Options 8-30

Using the Per-Rule Basis Option 8-31

Using the UPSERT and UPSERT ALL Options 8-32

Order of Evaluation of Rules 8-33

Nested Cell References 8-35

Lesson Agenda 8-36

Reference Models 8-37

Reference Models: Example 8-39

Cyclic Rules in Models 8-43

Cyclic References and Iterations 8-44

Cycles and Simultaneous Equations 8-45

Quiz 8-47

Summary 8-48

Practice 8 Overview 8-49

Appendix A: Data Warehousing Concepts: Overview

- Objectives A-2
- Lesson Agenda A-3
- Characteristics of a Data Warehouse A-4
- Comparing Online Transactional Processing (OLTP) with Data Warehouses (DW) A-5
- Data Warehouses Versus OLTP A-6
- Lesson Agenda A-8
- Data Warehouse: Design Phases A-9
- Data Warehousing Objects A-11
- Characteristics of Fact Tables A-12
- Dimensions and Hierarchies A-13
- Identify Hierarchies for Dimensions A-15
- Using Hierarchies to Drill on Data A-16
- Data Warehousing Schemas A-17
- Schema Characteristics A-18
- Star Schema Model: Central Fact Table and Denormalized Dimension Tables A-20
- Star Dimensional Schema: Advantages A-21
- Star Dimensional Modeling: The SH Schema A-22
- Snowflake Schema Model A-23
- Lesson Agenda A-25
- Using Summaries to Improve Performance A-26
- Summary Management: Overview A-27
- Summary Management Capabilities A-28
- Using Summary Management A-29
- Query Rewrite: Overview A-30
- The Optimizer Query Rewrite Process A-31
- The Available Query Rewrite Types A-32
- What Can Be Rewritten? A-33
- Requirements for Query Rewrite A-34
- Verifying Whether Query Rewrite Occurred A-35
- Analyzing Materialized View Capabilities: The DBMS_MVIEW.EXPLAIN_MVIEW Procedure A-36
- Summary A-37

Appendix B: Table Descriptions

Appendix C: Using SQL Developer

- Objectives C-2
- What Is Oracle SQL Developer? C-3
- Specifications of SQL Developer C-4

SQL Developer 3.2 Interface	C-5
Creating a Database Connection	C-7
Browsing Database Objects	C-10
Displaying the Table Structure	C-11
Browsing Files	C-12
Creating a Schema Object	C-13
Creating a New Table: Example	C-14
Using the SQL Worksheet	C-15
Executing SQL Statements	C-19
Saving SQL Scripts	C-20
Executing Saved Script Files: Method 1	C-21
Executing Saved Script Files: Method 2	C-22
Formatting the SQL Code	C-23
Using Snippets	C-24
Using Snippets: Example	C-25
Using Recycle Bin	C-26
Debugging Procedures and Functions	C-27
Database Reporting	C-28
Creating a User-Defined Report	C-29
Search Engines and External Tools	C-30
Setting Preferences	C-31
Resetting the SQL Developer Layout	C-33
Data Modeler in SQL Developer	C-34
Summary	C-35

Appendix D: Using SQL*Plus

Objectives	D-2
SQL and SQL*Plus Interaction	D-3
SQL Statements Versus SQL*Plus Commands	D-4
Overview of SQL*Plus	D-5
Logging In to SQL*Plus	D-6
Displaying the Table Structure	D-7
SQL*Plus Editing Commands	D-9
Using LIST, n, and APPEND	D-11
Using the CHANGE Command	D-12
SQL*Plus File Commands	D-13
Using the SAVE, START Commands	D-14
SERVERROUTPUT Command	D-15
Using the SQL*Plus SPOOL Command	D-16
Using the AUTOTRACE Command	D-17
Summary	D-18

Appendix E: Manipulating Large Data Sets

- Objectives E-2
- Using Subqueries to Manipulate Data E-3
- Retrieving Data Using a Subquery as Source E-4
- Inserting Using a Subquery as a Target E-6
- Inserting Using a Subquery as a Target: Viewing the Results E-7
- Using the WITH CHECK OPTION Keyword on DML Statements E-8
- Overview of the Explicit Default Feature E-10
- Using Explicit Default Values E-11
- Copying Rows from Another Table E-12
- Overview of Multitable INSERT Statements E-13
- Types of Multitable INSERT Statements E-15
- Multitable INSERT Statements E-16
- Unconditional INSERT ALL E-18
- Conditional INSERT ALL: Example E-20
- Conditional INSERT ALL E-21
- Conditional INSERT FIRST: Example E-22
- Conditional INSERT FIRST E-23
- Pivoting INSERT E-25
- MERGE Statement E-28
- MERGE Statement Syntax E-29
- Merging Rows: Example E-30
- MERGE Improvements E-32
- MERGE Extensions E-33
- Allowing Conditional Updates E-34
- Using Conditional Updates E-35
- Using the DELETE Clause to Cleanse Data E-38
- Tracking Changes in Data E-39
- Flashback Version Query: Example E-40
- VERSIONS BETWEEN Clause E-42
- Summary E-43

Appendix F: Densifying Data and Performing Time Period Comparison

- Objectives F-2
- Densifying Data with Partitioned Outer Joins F-3
- The Partitioned Outer Join Syntax F-5
- Sparse Data Sample F-6
- Densifying Data: Example F-7
- Repeating Data Values to Fill Gaps F-9
- Repeating Data Values to Fill Gaps Results F-11
- Computing Data Values to Fill Gaps F-12

Time Series Calculations on Densified Data	F-14
Time Series Calculations: Results	F-15
Period-to-Period Comparison of One Time Level	F-16
Period-to-Period Comparison for Multiple Time Levels: Example	F-19
View of the TIME Dimension	F-20
View of the SALES Cube	F-21
Materialized View of SALES	F-22
Period-to-Period Query	F-23
Period-to-Period Query Results	F-25
Summary	F-26

1

Introduction

ORACLE®

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Objectives

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

- Discuss the goals of the course
- Identify the appendixes in this course
- Identify the environments that are used in this course
- Review using SQL Developer and SQL*Plus environments
- Describe the database schemas and tables that are used in the course
- Review performing Analytical SQL operations in the database
- List the documentation and resources



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In this lesson, students are introduced to the course organization. Students identify the appendixes, schemas, and the SQL environments that are available in the course. Students briefly review and use the SQL Developer and the SQL*Plus tool. They also access the Oracle Database 12c documentation page and the online SQL Developer tutorial as needed.

Lesson Agenda

- Course objectives and course agenda
- Schemas, appendixes, and SQL environments used in this course
- Overview of Oracle SQL Developer and SQL*Plus environments
- Overview of Analytic SQL and case study
- Oracle documentation and additional resources



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Course Objectives

After completing this course, you should be able to:

- Identify the benefits of using Analytic SQL
- Group and aggregate data by using the ROLLUP and CUBE operators, the GROUPING function, and composite columns
- Interpret the concept of a hierarchical query, create a tree-structured report, and format hierarchical data
- Use regular expressions to search for, match, replace strings, and match patterns
- Analyze and report data by using Ranking functions, the LAG/LEAD functions, and the PIVOT and UNPIVOT clauses
- Use the MODEL clause to create and use a multidimensional array



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You can develop modularized applications with database procedures by using database objects such as the following:

- Procedures and functions
- Packages
- Database triggers

Modular applications improve:

- Functionality
- Security
- Overall performance

Course Agenda

- Lesson 1: Introduction
- Lesson 2: Grouping and Aggregating Data by Using SQL
- Lesson 3: Hierarchical Retrieval
- Lesson 4: Working with Regular Expressions
- Lesson 5: Analyzing and Reporting Data by Using SQL
- Lesson 6: Performing Pivoting and Unpivoting Operations
- Lesson 7: Pattern Matching by Using SQL
- Lesson 8: Modeling Data by Using SQL



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Lesson Agenda

- Course objectives and course agenda
- Schemas, appendixes, and SQL environments used in this course
- Overview of Oracle SQL Developer and SQL*Plus environments
- Overview of Analytic SQL and case study
- Oracle documentation and additional resources



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Sample Schemas Used in the Course

The sample schemas that are used in this course are:

- Human Resources (HR) schema
- Order Entry (OE) schema
- Sales History (SH) schema



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The sample company portrayed by 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 employees and facilities of the company.
- The Order Entry division tracks product inventories and sales of the company's products through various channels.
- The Sales History division tracks business statistics to facilitate business decisions.

Each of these divisions is represented by a schema.

All scripts necessary to create the SH schema reside in the \$ORACLE_HOME/demo/schema/sales_history folder.

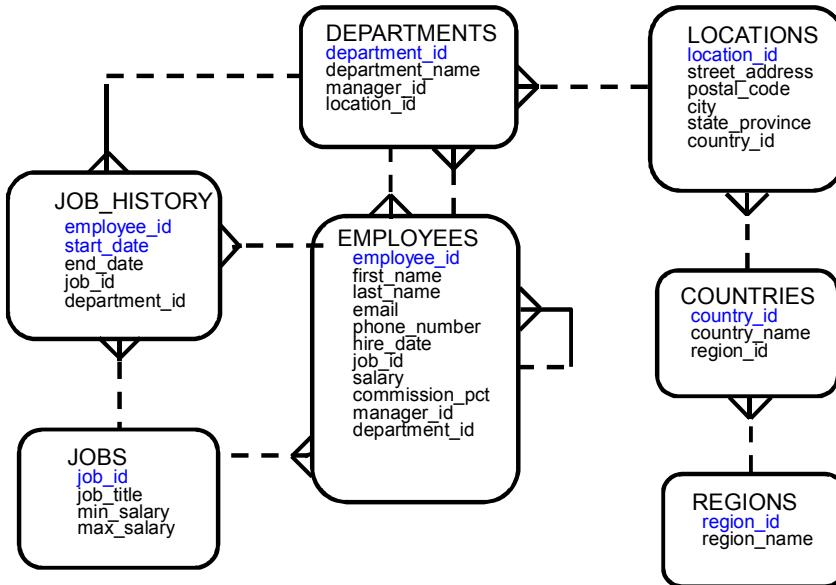
All scripts necessary to create the OE schema reside in the \$ORACLE_HOME/demo/schema/order_entry folder.

All scripts necessary to create the HR schema reside in the \$ORACLE_HOME/demo/schema/human_resources folder.

Notes:

- Appendix B, “Table Descriptions” contains more information about the sample schemas.
- The code examples and practices in this course specify the schema that must be used.

Human Resources (HR) Schema



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

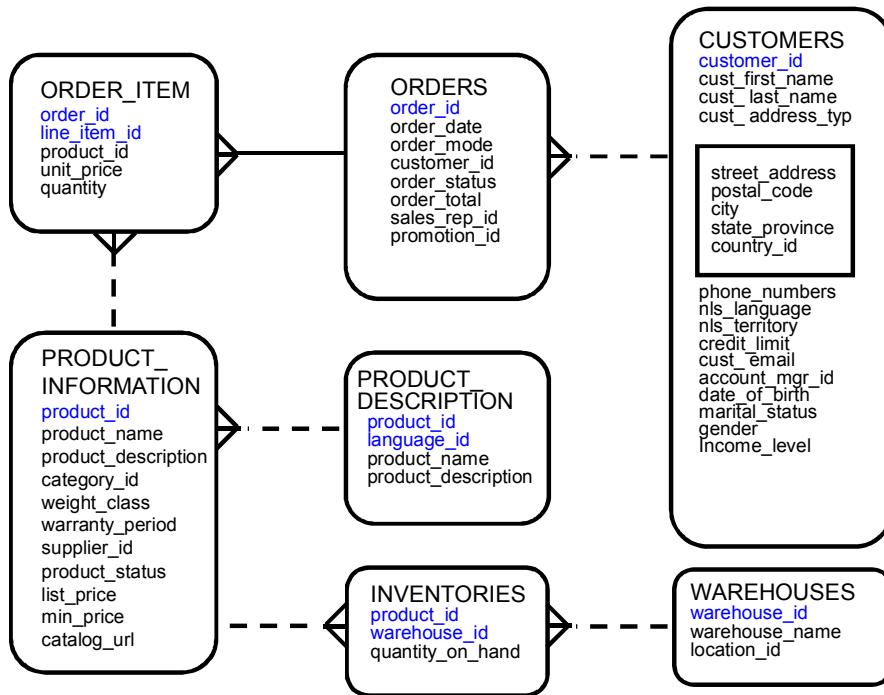
In the HR records, each employee has an identification number, email address, job identification code, salary, and manager. Some employees earn commissions in addition to their salary.

The company also tracks information about 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 resigns, the duration the employee was working, the job identification number, and the department are recorded.

The sample company is regionally diverse, so it tracks the locations of its warehouses and departments. Each employee is assigned to a department and each department is identified by a unique department number or a short name. Each department is associated with one location and each location has a full address that includes the street name, postal code, city, state or province, and the country code.

In places where the departments and warehouses are located, the company records details such as the country name, currency symbol, currency name, and the region where the country is located geographically.

Order Entry (OE) Schema

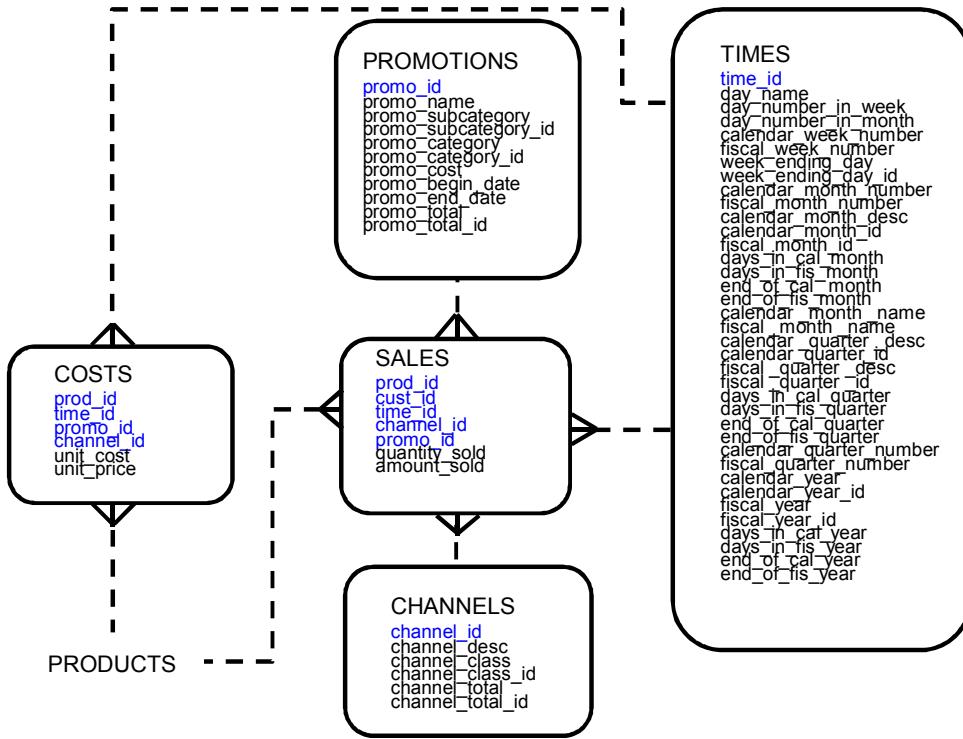


ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

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 is sold, and a URL for manufacturer information.

Sales History (SH) Schema



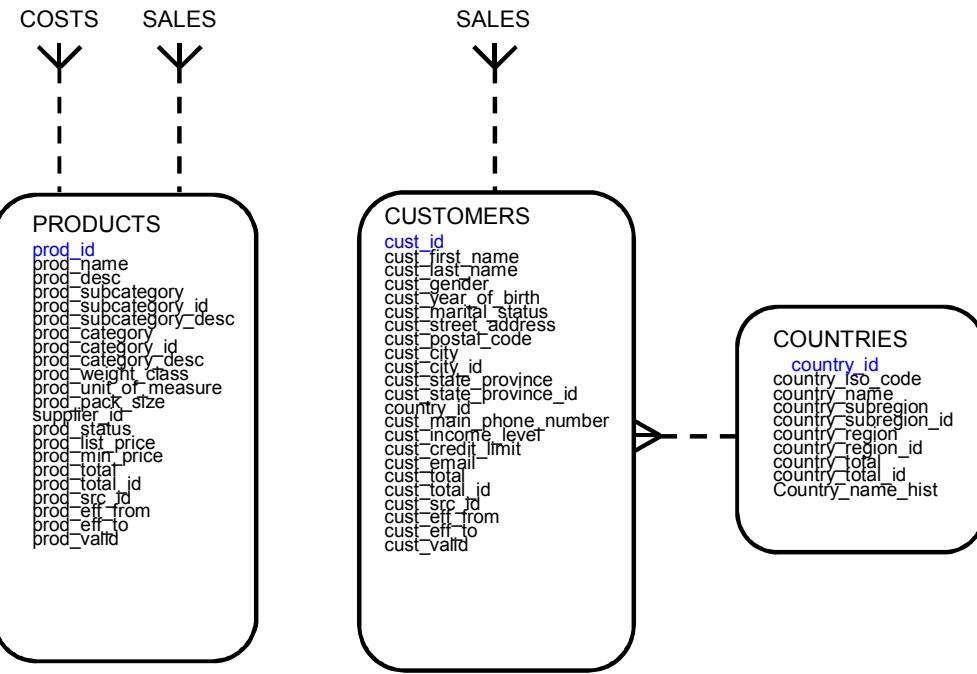
ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The company does a high volume of business, so it runs business statistics reports to aid in decision support. Many of these reports are time-based and nonvolatile. That is, they analyze past data trends. The company loads data into its data warehouse regularly to gather statistics for the reports. The reports include annual, quarterly, monthly, and weekly sales figures by product.

The company also runs reports on the distribution channels through which its sales are delivered. When the company runs special promotions on its products, it analyzes the impact of the promotions on sales. It also analyzes sales by geographical area.

Sales History (SH) Schema

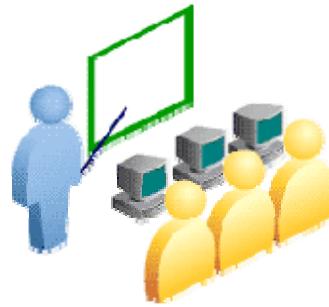


ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Class Account Information

- Your account IDs are sh, hr, and oe.
- The password matches your account ID.
- Each machine has a stand-alone installation of Oracle Database12c Enterprise Edition with access to the SH, HR, and OE sample schemas.



ORACLE®

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Appendices in the Course

- Appendix A: Data Warehousing Concepts: Overview
- Appendix B: Table Descriptions
- Appendix C: Using SQL Developer
- Appendix D: Using SQL*Plus
- Appendix E: Manipulating Large Data Sets
- Appendix F: Densifying Data and Performing Time Period Comparison



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

SQL Environments Available in the Course

This course setup provides the following tools for developing SQL code:

- Oracle SQL Developer (used in this course)
- Oracle SQL*Plus



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

There are many tools that provide an environment for executing SQL statements. Oracle provides several tools that can be used to execute SQL statements. Some of the tools that are available for use in this course are:

- **Oracle SQL Developer:** A graphical tool
- **Oracle SQL*Plus:** A window or command-line application

Note: The code and screen examples presented in the course are from output in the SQL Developer environment.

Lesson Agenda

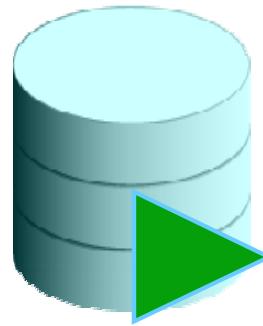
- Course objectives and course agenda
- Schemas, appendixes, and SQL environments used in this course
- Overview of Oracle SQL Developer and SQL*Plus environments
- Overview of Analytic SQL and case study
- Oracle documentation and additional resources



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

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 © 2013, 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 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 you are 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 onto the worksheet. It provides improved DB Diff comparison options, GRANT statement support in the SQL editor, and DB Doc reporting. Also, SQL Developer supports 12c database features.

Refer to Appendix C: “Using SQL Developer” for more information about SQL Developer.

SQL*Plus: Overview

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



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

SQL*Plus commands can be divided into the following main 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

Refer to Appendix D, “Using SQL*Plus” for more information about SQL*Plus.

Lesson Agenda

- Course objectives and course agenda
- Schemas, appendixes, and SQL environments used in this course
- Overview of Oracle SQL Developer and SQL*Plus environments
- Overview of Analytic SQL and case study
- Oracle documentation and additional resources



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

What Is Analytic SQL?

- Oracle has introduced many SQL operations for performing analytic operations in the database.
- These operations include ranking, moving averages, cumulative sums, ratio-to-reports, and period-over-period comparisons.
- Although some of these calculations were previously possible using SQL, this syntax offers ease of use as well as performance improvements.



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Analytic SQL in Data Warehouses

- Using SQL for aggregation in data warehouses:
 - CUBE and ROLLUP extensions to the GROUP BY clause
 - GROUPING functions
 - GROUPING SETS expression
- Using SQL for analysis and reporting in data warehouses:
 - Rankings and percentiles
 - Performing pivoting operations
 - Moving window calculations
 - Lag/lead and First/last analysis
- Using SQL for modeling data and pattern matching in data warehouses:
 - MODEL clause
 - Navigation operations



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Case Study: Overview

In this case study you:

- Create statistical reports to aid decision support for a company named ABC Sales Consultants
- Perform business intelligence queries
- Create company organizational charts



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In this case study, you will perform business intelligence queries on the data available in the SH schema, HR schema, and OE schema, for a company named ABC Sales Consultants. This company handles high volumes of business information. One of their requirements involves creating business statistics reports to aid in decision support. To achieve this requirement, the IT Database (ITDB) Support group loads sales and customer-related data into their data warehouse on a quarterly basis.

Note: The case study is introduced at the end of each practice. In this case study, you help the IT Database (ITDB) staff to perform business intelligence queries and create statistical reports.

Lesson Agenda

- Course objectives and course agenda
- Schemas, appendixes, and SQL environments used in this course
- Overview of Oracle SQL Developer and SQL*Plus environments
- Overview of Analytic SQL and case study
- Oracle documentation and additional resources



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Oracle SQL and Data Warehousing Guide Documentation

- Oracle Database 2 Day + Data Warehousing
- Oracle Database Data Warehousing Guide
- Oracle Database SQL Developer User's Guide
- Oracle Database Reference
- Oracle Database New Features Guide
- Oracle Database SQL Language Reference
- SQL*Plus User's Guide and Reference
- Oracle Database Concepts
- Oracle Database Sample Schemas



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Navigate to <http://www.oracle.com/pls/db121/homepage> and click the Master Book List link in the left frame.

Additional Resources

For additional information about the Oracle SQL and PL/SQL new features, refer to:

- Oracle Database: New Features self study
- Oracle by Example series (OBE) from the Oracle Learning Library



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The following are the recommended next courses:

- *Oracle Database: Implement and Administer a Data Warehouse* four-day instructor-led training (ILT) course
- *Oracle Database: SQL Tuning Workshop* three-day instructor-led training (ILT)

Summary

In this lesson, you should have learned how to:

- Discuss the goals of the course
- Identify the appendixes in this course
- Describe the database schemas and tables that are used in the course
- Identify the environments that can be used in this course
- Review using SQL Developer and SQL*Plus to execute SQL statements
- Review using Analytic SQL operations in the database
- List the documentation and resources



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Practice 1 Overview

This practice covers the following topics:

- Reviewing the SQL Developer resources
- Starting SQL Developer, creating new database connections, and browsing tables in the HR, SH, and OE schemas
- Setting some SQL Developer preferences
- Accessing and bookmarking the Oracle database documentation and other useful websites



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In this practice, you use SQL Developer to execute SQL statements to examine data in the HR schema.

Note:

- All written practices use SQL Developer as the development environment. Although it is recommended that you use SQL Developer, you can also use the SQL*Plus environment that is available in this course.
- All practices in this course and the practice solutions assume that you run scripts using the SQL Worksheet area in SQL Developer.

Grouping and Aggregating Data by Using SQL



ORACLE®

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Objectives

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

- Use the ROLLUP operator to produce subtotal and grand total values
- Use the CUBE operator to produce cross-tabulation values
- Use the GROUPING function to identify the row values created by ROLLUP or CUBE
- Use GROUPING SETS to produce a single result set
- Use the GROUPING_ID function to determine the GROUP BY level in a SELECT query
- Work with composite columns
- Use concatenated groupings to generate useful combinations of groupings



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In this lesson, you learn how to group data to do the following:

- Obtain subtotal and grant total values by using the ROLLUP operator
- Obtain cross-tabulation values by using the CUBE operator
- Use the GROUPING function to identify the level of aggregation in the result set produced by a ROLLUP or CUBE operator
- Use the GROUPING_ID function to identify the GROUP BY level of a row in a query
- Use GROUPING SETS to produce a single result set that is equivalent to a UNION ALL approach

Lesson Agenda

- Using SQL for aggregation in data warehouses:
 - CUBE and ROLLUP extensions to the GROUP BY clause
 - GROUPING function
 - GROUPING SETS expression
- Working with composite columns
- Using concatenated groupings



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Group Functions

Group functions operate on sets of rows to give one result per group.

```
SELECT      [column,] group_function(column) . . .
FROM        table
[WHERE      condition]
[GROUP BY   group_by_expression]
[ORDER BY   column];
```

```
SELECT AVG(salary), STDDEV(salary),
       COUNT(commission_pct), MAX(hire_date)
  FROM hr.employees
 WHERE job_id LIKE 'SA%';
```

	AVG(SALARY)	STDDEV(SALARY)	COUNT(COMMISSION_PCT)	MAX(HIRE_DATE)
1	8900	2030.647535089412281325559194205199066	35	21-APR-08



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You can use the GROUP BY clause to divide the rows in a table into groups. You can then use group functions to return summary information for each group. Group functions can appear in select lists and in ORDER BY and HAVING clauses. The Oracle server applies the group functions to each group of rows and returns a single result row for each group.

Types of group functions: Each of the group functions—AVG, SUM, MAX, MIN, COUNT, STDDEV, and VARIANCE—accepts one argument. The AVG, SUM, STDDEV, and VARIANCE functions operate only on numeric values. MAX and MIN can operate on numeric, character, or date data values. COUNT returns the number of non-NULL rows for the given expression. The example in the slide calculates the average salary, standard deviation on the salary, number of employees earning a commission, and the maximum hire date for those employees whose JOB_ID begins with SA.

Guidelines for Using Group Functions

- The data types for the arguments can be CHAR, VARCHAR2, NUMBER, or DATE.
- All group functions except COUNT (*) ignore null values. To substitute a value for null values, use the NVL function. COUNT returns a number or zero.
- The Oracle server implicitly sorts the result set in ascending order, when you use an ORDER BY clause. To override this default ordering, you can use the DESC option.

Note: The code examples in this lesson use the HR schema.

GROUP BY Clause

```
SELECT [column,] group_function(column)
FROM table
[WHERE condition]
[GROUP BY group_by_expression]
[ORDER BY column];
```

```
SELECT department_id, job_id, SUM(salary),
       COUNT(employee_id)
  FROM hr.employees
 GROUP BY department_id, job_id
Order by department_id;
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The example illustrated in the slide is evaluated by the Oracle server as follows:

- The SELECT clause specifies that the following columns be retrieved:
 - Department ID and job ID columns from the EMPLOYEES table
 - The sum of all the salaries and the number of employees in each group that you specified in the GROUP BY clause
- The GROUP BY clause specifies how the rows should be grouped in the table. The total salary and the number of employees are calculated for each job ID within each department. The rows are grouped by department ID and then grouped by job within each department. The results of the example in the slide are as follows:

	DEPARTMENT_ID	JOB_ID	SUM(SALARY)	COUNT(EMPLOYEE_ID)
1		10 AD_ASST	4400	1
2		20 MK_MAN	13000	1
3		20 MK_REP	6000	1
...				
16		100 FI_ACCOUNT	39600	5
17		100 FI_MGR	12008	1
18		110 AC_ACCOUNT	8300	1
19		110 AC_MGR	12008	1
20		SA_REP	7000	1

	DEPARTMENT_ID	JOB_ID	SUM(SALARY)	COUNT(EMPLOYEE_ID)
16		100 FI_ACCOUNT	39600	5
17		100 FI_MGR	12008	1
18		110 AC_ACCOUNT	8300	1
19		110 AC_MGR	12008	1
20		SA_REP	7000	1

HAVING Clause

- Use the HAVING clause to specify which groups are to be displayed.
- You further restrict the groups on the basis of a limiting condition.

```
SELECT [column,] group_function(column)...
FROM table
[WHERE condition]
[GROUP BY group_by_expression]
[HAVING having_expression]
[ORDER BY column];
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Groups are formed and group functions are calculated before the HAVING clause is applied to the groups. The HAVING clause can precede the GROUP BY clause, but it is recommended that you place the GROUP BY clause first because it is more logical.

The Oracle server performs the following steps when you use the HAVING clause:

1. It groups rows.
2. It applies group functions to the groups and displays the groups that match the criteria in the HAVING clause.

GROUP BY with ROLLUP and CUBE Operators

- Use ROLLUP or CUBE with GROUP BY to produce superaggregate rows by cross-referencing columns.
- ROLLUP grouping produces a result set containing the regular grouped rows and the subtotal and grand total values.
- CUBE grouping produces a result set containing the rows from ROLLUP and cross-tabulation rows.



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You specify ROLLUP and CUBE operators in the GROUP BY clause of a query. ROLLUP grouping produces a result set containing the regular grouped rows and subtotal rows. The ROLLUP operator also calculates a grand total. The CUBE operator in the GROUP BY clause groups the selected rows based on the values of all possible combinations of expressions in the specification and returns a single row of summary information for each group. You can use the CUBE operator to produce cross-tabulation rows.

Note: When working with ROLLUP and CUBE, make sure that the columns following the GROUP BY clause have meaningful, real-life relationships with each other; otherwise, the operators return irrelevant information.

Using the ROLLUP Operator

- ROLLUP is an extension of the GROUP BY clause.
- Use the ROLLUP operation to produce cumulative aggregates, such as subtotals.

```
SELECT [column,] group_function(column) . . .
FROM table
[WHERE condition]
[GROUP BY [ROLLUP] group_by_expression]
[HAVING having_expression];
[ORDER BY column];
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The ROLLUP operator delivers aggregates and superaggregates for expressions within a GROUP BY statement. The ROLLUP operator can be used by report writers to extract statistics and summary information from result sets. The cumulative aggregates can be used in reports, charts, and graphs.

The ROLLUP operator creates groupings by moving in one direction, from right to left, along the list of columns specified in the GROUP BY clause. It then applies the aggregate function to these groupings.

Notes:

- To produce subtotals in n dimensions (that is, n columns in the GROUP BY clause) without a ROLLUP operator, $n+1$ SELECT statements must be linked with UNION ALL. This makes the query execution inefficient because each of the SELECT statements causes table access. The ROLLUP operator gathers its results with just one table access. The ROLLUP operator is useful when there are many columns involved in producing the subtotals.
- Subtotals and totals are produced with ROLLUP. The CUBE operator produces totals as well, but effectively rolls up in each possible direction, producing cross-tabular data.

Using the ROLLUP Operator: Example

```
SELECT      department_id, job_id, SUM(salary)
FROM        hr.employees
WHERE       department_id < 60
GROUP BY   ROLLUP(department_id, job_id);
```

	DEPARTMENT_ID	JOB_ID	SUM(SALARY)
1	10	AD_ASST	4400
2	10		4400
3	20	MK_MAN	13000
4	20	MK_REP	6000
5	20		19000
6	30	PU_MAN	11000
7	30	PU_CLERK	13900
8	30		24900
9	40	HR REP	6500
10	40		6500
11	50	ST_MAN	36400
12	50	SH_CLERK	64300
13	50	ST_CLERK	55700
14	50		156400
15			211200

1 Total by DEPARTMENT_ID and JOB_ID

2 Total by DEPARTMENT_ID

3 Grand total

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In the example in the slide:

- Total salaries for every job ID within a department—for those departments whose department ID is less than 60—are displayed by the GROUP BY clause
- The ROLLUP operator displays:
 - Total salary for each department whose department ID is less than 60
 - Total salary for all departments whose department ID is less than 60, irrespective of the job IDs

In this example, 1 indicates a group totaled by both DEPARTMENT_ID and JOB_ID, 2 indicates a group totaled only by DEPARTMENT_ID, and 3 indicates the grand total.

The ROLLUP operator creates subtotals that roll up from the most detailed level to a grand total, following the grouping list specified in the GROUP BY clause. First, it calculates the standard aggregate values for the groups specified in the GROUP BY clause (in the example, the sum of salaries grouped on each job within a department). Then it creates progressively higher-level subtotals, moving from right to left through the list of grouping columns. (In the example, the sum of salaries for each department is calculated, followed by the sum of salaries for all departments.)

- Given n expressions in the ROLLUP operator of the GROUP BY clause, the operation results in $n + 1$ (in this case, $2 + 1 = 3$) groupings.
- Rows based on the values of the first n expressions are called “rows” or “regular rows,” and the others are called “superaggregate rows.”

Using the CUBE Operator

- CUBE is an extension of the GROUP BY clause.
- You can use the CUBE operator to produce cross-tabulation values with a single SELECT statement.

```
SELECT [column,] group_function(column) ...
FROM table
[WHERE condition]
[GROUP BY [CUBE] group_by_expression]
[HAVING having_expression]
[ORDER BY column];
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The CUBE operator is an additional switch in the GROUP BY clause in a SELECT statement. The CUBE operator can be applied to all aggregate functions, including AVG, SUM, MAX, MIN, and COUNT. It is used to produce result sets that are typically used for cross-tabular reports. ROLLUP produces only a fraction of possible subtotal combinations, whereas CUBE produces subtotals for all possible combinations of groupings specified in the GROUP BY clause, and a grand total.

The CUBE operator is used with an aggregate function to generate additional rows in a result set. Columns included in the GROUP BY clause are cross-referenced to produce a superset of groups. The aggregate function specified in the select list is applied to these groups to produce summary values for the additional superaggregate rows. The number of extra groups in the result set is determined by the number of columns included in the GROUP BY clause.

In fact, every possible combination of the columns or expressions in the GROUP BY clause is used to produce superaggregates. If you have n columns or expressions in the GROUP BY clause, there will be 2^n possible superaggregate combinations. Mathematically, these combinations form an n -dimensional cube, which is how the operator gets its name.

By using application or programming tools, these superaggregate values can then be fed into charts and graphs that convey results and relationships visually and effectively.

Using the CUBE Operator: Example

```
SELECT      department_id, job_id, SUM(salary)
FROM        hr.employees
WHERE       department_id < 60
GROUP BY    CUBE (department_id, job_id) ;
```

The diagram shows the output of the provided SQL query. The data is presented in a table with columns: DEPARTMENT_ID, JOB_ID, and SUM(SALARY). The rows are numbered from 1 to 18. Annotations 1 through 4 are placed next to specific rows to indicate different types of totals generated by the CUBE operator:

- Annotation 1 points to row 1, which contains the value 211200 in the SUM(SALARY) column. This is labeled as "Grand total".
- Annotation 2 points to rows 10 and 12, which both contain the value 4400 in the SUM(SALARY) column. This is labeled as "Total by JOB_ID".
- Annotation 3 points to rows 11, 12, and 13, which represent subtotal rows for department 10. These rows have department_id 10 and job_id AD_ASST, with values 4400, 4400, and 19000 respectively. This is labeled as "Total by DEPARTMENT_ID and JOB_ID".
- Annotation 4 points to rows 16 and 18, which represent subtotal rows for department 30. These rows have department_id 30 and job_id PU_MAN and PU_CLERK, with values 24900 and 13900 respectively. This is labeled as "Total by DEPARTMENT_ID".

	DEPARTMENT_ID	JOB_ID	SUM(SALARY)
1			211200
2		HR_REP	6500
3		MK_MAN	13000
4		MK_REP	6000
5		PU_MAN	11000
6		ST_MAN	36400
7		AD_ASST	4400
8		PU_CLERK	13900
9		SH_CLERK	64300
10		ST_CLERK	55700
11	10		4400
12	10	AD_ASST	4400
13	20		19000
14	20	MK_MAN	13000
15	20	MK_REP	6000
16	30		24900
17	30	PU_MAN	11000
18	30	PU_CLERK	13900
.	.	.	.

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The output of the `SELECT` statement in the example can be interpreted as follows:

- The total salary for every job within a department (for those departments whose department ID is less than 60)
- The total salary for each department whose department ID is less than 60
- The total salary for each job irrespective of department
- The total salary for those departments whose department ID is less than 60, irrespective of the job titles

In this example, 1 indicates the grand total, 2 indicates the rows totaled by `JOB_ID` alone, 3 indicates some of the rows totaled by `DEPARTMENT_ID` and `JOB_ID`, and 4 indicates some of the rows totaled by `DEPARTMENT_ID` alone.

The `CUBE` operator has also performed the `ROLLUP` operation to display the subtotals for those departments whose department ID is less than 60 and the total salary for those departments whose department ID is less than 60, irrespective of the job titles. Further, the `CUBE` operator displays the total salary for every job irrespective of department.

Note: Similar to the `ROLLUP` operator, producing subtotals in n dimensions (that is, n columns in the `GROUP BY` clause) without a `CUBE` operator requires that 2^n `SELECT` statements be linked with `UNION ALL`. Thus, a report with three dimensions requires $2^3 = 8$ `SELECT` statements to be linked with `UNION ALL`.

Lesson Agenda

- Using SQL for aggregation in data warehouses:
 - CUBE and ROLLUP extensions to the GROUP BY clause
 - GROUPING function
 - GROUPING SETS expression
- Working with composite columns
- Using concatenated groupings



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Working with the GROUPING Function

The GROUPING function:

- Is used with the CUBE or ROLLUP operator
- Is used to find the groups forming the subtotal in a row
- Is used to differentiate stored NULL values from NULL values created by ROLLUP or CUBE
- Returns 0 or 1

```
SELECT      [column,] group_function(column) . . ,
            GROUPING(expr)
FROM        table
[WHERE      condition]
[GROUP BY  [ROLLUP] [CUBE] group_by_expression]
[HAVING    having_expression]
[ORDER BY  column];
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The GROUPING function can be used with the CUBE or ROLLUP operator to help determine how a summary value has been obtained. The GROUPING function uses a single column as its argument. The *expr* in the GROUPING function must match one of the expressions in the GROUP BY clause. The function returns a value of 0 or 1.

The values returned by the GROUPING function are useful to:

- Determine the level of aggregation of a given subtotal (that is, the group or groups on which the subtotal is based)
- Identify whether a NULL value in the expression column of a row in the result set indicates:
 - A NULL value from the base table (stored NULL value)
 - A NULL value created by ROLLUP or CUBE (as a result of a group function on that expression)

A value of 0 returned by the GROUPING function based on an expression indicates one of the following:

- The expression has been used to calculate the aggregate value.
- The NULL value in the expression column is a stored NULL value.

A value of 1 returned by the GROUPING function based on an expression indicates one of the following:

- The expression has not been used to calculate the aggregate value.
- The NULL value in the expression column is created by ROLLUP or CUBE as a result of grouping.

Working with the GROUPING Function: Example

```

SELECT      department_id DEPTID, job_id JOB,
            SUM(salary),
            GROUPING(department_id) GRP_DEPT,
            GROUPING(job_id) GRP_JOB
FROM        hr.employees
WHERE       department_id < 50
GROUP BY   ROLLUP(department_id, job_id);
    
```

The diagram shows the output of the SQL query as a table. The columns are labeled DEPTID, JOB, SUM(SALARY), GRP_DEPT, and GRP_JOB. The table has 11 rows, indexed from 1 to 11. Row 11 is a summary row for all departments. Callout 1 points to row 1, callout 2 points to row 2, and callout 3 points to row 11.

	DEPTID	JOB	SUM(SALARY)	GRP_DEPT	GRP_JOB
1	10	AD_ASST	4400	0	0
2	10		4400	0	1
3	20	MK_MAN	13000	0	0
4	20	MK_REP	6000	0	0
5	20		19000	0	1
6	30	PU_MAN	11000	0	0
7	30	PU_CLERK	13900	0	0
8	30		24900	0	1
9	40	HR_REP	6500	0	0
10	40		6500	0	1
11			54800	1	1

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

- In the example in the slide, consider the summary value 4400 in the first row (labeled 1). This summary value is the total salary for the job ID of AD_ASST within department 10. To calculate this summary value, both the DEPARTMENT_ID and JOB_ID columns have been taken into account. Thus, a value of 0 is returned for both the GROUPING(department_id) and GROUPING(job_id) expressions.
- Consider the summary value 4400 in the second row (labeled 2). This value is the total salary for department 10 and has been calculated by taking into account the DEPARTMENT_ID column; thus, a value of 0 has been returned by GROUPING(department_id). Because the JOB_ID column has not been taken into account to calculate this value, a value of 1 has been returned for GROUPING(job_id). You can observe similar output in the fifth row.
- In the last row, consider the summary value 54800 (labeled 3). This is the total salary for those departments whose department ID is less than 50 and all job titles. Both DEPARTMENT_ID and JOB_ID columns have not been taken into account to calculate this summary value. Thus, a value of 1 is returned for both the GROUPING(department_id) and GROUPING(job_id) expressions.

Working with the GROUPING_ID Function

- The GROUPING_ID function returns a number corresponding to the GROUPING bit vector associated with a row.
- The bit vector is treated as a binary number with 1s and 0s.

```
SELECT department_id, job_id, SUM(salary),
       GROUPING_ID(department_id, job_id) gdept_job,
       GROUPING_ID(job_id, department_id) gjob_dept
  FROM hr.employees
 WHERE department_id < 50
 GROUP BY CUBE(department_id, job_id)
 ORDER BY gjob_dept;
```



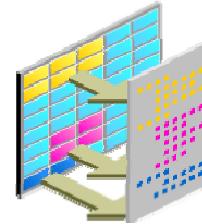
Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The GROUPING_ID function can be used in a SQL query, (which contains many GROUP_BY expressions) to determine the GROUP_BY level of a particular row. GROUPING_ID is functionally equivalent to taking the results of multiple GROUPING functions and concatenating them into a bit vector (a string of ones and zeroes). By using this function, you can avoid the need for multiple GROUPING functions. Also, you can make row filtering conditions easier to express because the desired rows can be identified with a single condition of GROUPING_ID = n. This function is widely used when storing multiple levels of aggregation in a single table. The results of the example in the slide are as follows:

	DEPARTMENT_ID	JOB_ID	SUM(SALARY)	GDEPT_JOB	GJOB_DEPT
1	20	MK_MAN	13000	0	0
...					
12		HR_REP	6500	2	1
13		30	24900	1	2
14		10	4400	1	2
15		40	6500	1	2
16		20	19000	1	2
17			54800	3	3

GROUPING SETS

- The GROUPING SETS syntax is used to define multiple groupings in the same query.
- Grouping set efficiency:
 - Only one pass over the base table is required.
 - There is no need to write complex UNION statements.
 - The more elements GROUPING SETS has, the greater the performance benefit.



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

GROUPING SETS is a further extension of the GROUP BY clause that you can use to specify multiple groupings of data. Doing so facilitates efficient aggregation and, therefore, facilitates analysis of data across multiple dimensions. All groupings specified in the GROUPING SETS clause are computed and the results of individual groupings are combined with a UNION ALL operation. A single SELECT statement can now be written by using GROUPING SETS to specify various groupings, (which can also include ROLLUP or CUBE operators), rather than multiple SELECT statements combined by UNION ALL operators. Example:

```
SELECT department_id, job_id, manager_id, AVG(salary)
FROM employees
GROUP BY
GROUPING SETS
    ((department_id, job_id, manager_id),
     (department_id, manager_id), (job_id, manager_id));
```

This statement calculates aggregates over three groupings:

(department_id, job_id, manager_id), (department_id, manager_id), and
(job_id, manager_id)

Without this feature, multiple queries combined together with UNION ALL must obtain the output of the preceding SELECT statement. A multiquery approach is inefficient because it requires multiple scans of the same data.

The results of the GROUPING SETS example displayed on the previous page are as follows:

	DEPARTMENT_ID	JOB_ID	MANAGER_ID	Avg(Salary)
1		90 AD_VP	100	17000
2		110 AC_MGR	101	12008
3		100 FI_MGR	101	12008
4		40 HR REP	101	6500
5		20 MK_MAN	100	13000
6		20 MK_REP	201	6000
7		70 PR_REP	101	10000
8		30 PU_MAN	100	11000
9		80 SA_MAN	100	12200
10		80 SA_REP	145	8500
<hr/>				
62		ST_CLERK	123	3000
63		ST_CLERK	124	2925
64		AC_ACCOUNT	205	8300
65		FT ACCOUNT	108	7920
66		40	101	6500
67		80	146	8500
68		60	103	4950
69		70	101	10000
70		80	145	8500
71		80	149	8600
72			149	7000
73		50	122	2950
74		80	147	7766.667
75		50	100	7280
76		50	121	3175
77		50	123	3237.5
78		110	205	8300
79		90	100	17000
80		50	120	2762.5
81		30	100	11000
82		10	101	4400
83		90		24000
84		100	101	12008
85		80	148	8650
86		60	102	9000
87		50	124	2875
88		110	101	12008
89		20	100	13000
90		20	201	6000
91		80	100	12200
92		30	114	2780

1

3

2

Compare the GROUPING SETS expression example on the previous page with the following alternative:

```
SELECT department_id, job_id, manager_id, AVG(salary)
FROM hr.employees
GROUP BY CUBE(department_id, job_id, manager_id);
```

This statement computes all the 8 (2^3) groupings, though only the (department_id, job_id, manager_id), (department_id, manager_id), and (job_id, manager_id) groups interest you.

Another alternative is the following statement:

```
SELECT department_id, job_id, manager_id, AVG(salary)
FROM hr.employees
GROUP BY department_id, job_id, manager_id
UNION ALL
SELECT department_id, NULL, manager_id, AVG(salary)
FROM hr.employees
GROUP BY department_id, manager_id
UNION ALL
SELECT NULL, job_id, manager_id, AVG(salary)
FROM hr.employees
GROUP BY job_id, manager_id;
```

This statement requires three scans of the base table. Note that you do not have native support for GROUPING SETS. The optimizer uses the UNION ALL operator to handle intermediate query blocks as illustrated previously. Therefore, using the ROLLUP and/or CUBE operators in place of GROUPING SETS is recommended. CUBE and ROLLUP can be thought of as grouping sets with very specific semantics and results. The following equivalencies show this fact:

CUBE(a, b, c) is equivalent to	GROUPING SETS ((a, b, c), (a, b), (a, c), (b, c), (a), (b), (c), ())
ROLLUP(a, b, c) is equivalent to	GROUPING SETS ((a, b, c), (a, b), (a), ())

GROUPING SETS: Example 2

```
SELECT      department_id, job_id,
            manager_id, AVG(salary)
FROM        hr.employees
GROUP BY   GROUPING SETS
            ((department_id,job_id), (job_id,manager_id), ());
```

The diagram illustrates the output of the GROUPING SETS query. It shows three distinct result sets, each highlighted with a red border:

- Set 1:** Groups employees by department and job. This corresponds to the grouping `((department_id,job_id))`. The output is a table with columns: DEPARTMENT_ID, JOB_ID, MANAGER_ID, and AVG(SALARY). The data is as follows:

DEPARTMENT_ID	JOB_ID	MANAGER_ID	AVG(SALARY)
1	SH_CLERK	122	3200
2	AC_MGR	101	12000
3	ST_MAN	100	7280
4	ST_CLERK	121	2675
5	SA_REP	148	8650

- Set 2:** Groups employees by department and manager. This corresponds to the grouping `(job_id,manager_id)`. The output is a table with columns: DEPARTMENT_ID, JOB_ID, MANAGER_ID, and AVG(SALARY). The data is as follows:

DEPARTMENT_ID	JOB_ID	MANAGER_ID	AVG(SALARY)
33	110 AC_ACCOUNT		8300
34	90 AD_VP		17000
35	50 ST_CLERK		2785

- Set 3:** A single row representing the grand total average salary. This corresponds to the grouping `()`. The output is a table with one row containing the value 6461.831775700934579439252336448598130841.

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The query in the slide calculates aggregates over two groupings. The table is divided into the following groups:

- Department ID, job ID
- Job ID, manager ID
- Grant total

The average salaries for each of these groups, with the average grand total, is calculated. The result set displays the average salary for each of the two groups.

In the output, the group marked as 1 can be interpreted as follows:

- The average salary of all employees with the `SH_CLERK` job ID under manager 122 is 3,200.
- The average salary of all employees with the `AC_MGR` job ID under manager 101 is 12,000, and so on.

The group marked as 2 in the output is interpreted as follows:

- The average salary of all employees with the `AC_ACCOUNT` job ID in department 110 is 8,300.
- The average salary of all employees with the `AD_VP` job ID in department 90 is 17,000, and so on.

The group marked as 3 in the output is interpreted as the average of all the values listed under the AVG(SALARY) column.

The example in the slide can also be written as:

```
SELECT department_id, job_id, NULL as manager_id,  
      AVG(salary) as AVGSAL  
FROM hr.employees  
GROUP BY department_id, job_id  
UNION ALL  
SELECT NULL, job_id, manager_id, avg(salary) as AVGSAL  
FROM hr.employees  
GROUP BY job_id, manager_id;
```

The following is the output of this code example:

NR	DEPARTMENT_ID	JOB_ID	MANAGER_ID	AVGSAL
1		110 AC_ACCOUNT		8300
2		90 AD_VP		17000
3		50 ST_CLERK		2785
4		80 SA_REP		8396.551724137931034482758620689655172414
5		50 ST_MAN		7280
6		80 SA_MAN		12200

33	AD_ASST	101	4400
34	IT_PROG	102	9000
35	IT_PROG	103	4950
36	FI_ACCOUNT	108	7920
37	PU_MAN	100	11000
38	ST_CLERK	122	2700
39	SA_REP	145	8500
40	AC_ACCOUNT	205	8300
41	AD_VP	100	17000
42	ST_CLERK	120	2625
43	ST_CLERK	124	2925
44	SA_REP	147	7766.666666666666666666666666666666666666667
45	SA_REP	149	8333.33333333333333333333333333333333333333
46	HR REP	101	6500
47	PR REP	101	10000
48	ST_CLERK	123	3000
49	SH_CLERK	121	3675
50	PU_CLERK	114	2780
51	SA_MAN	100	12200
52	MK REP	201	6000

In the absence of an optimizer that looks across query blocks to generate the execution plan, the preceding query needs two scans of the EMPLOYEES base table. This could be very inefficient. Therefore, the usage of the GROUPING SETS statement is recommended.

Lesson Agenda

- Using SQL for aggregation in data warehouses
 - CUBE and ROLLUP extensions to the GROUP BY clause
 - GROUPING function
 - GROUPING SETS expression
- Working with composite columns
- Using concatenated groupings



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Composite Columns

- A composite column is a collection of columns that are treated as a unit.
`ROLLUP (a, (b, c))`
- Use parentheses within the GROUP BY clause to group columns, so that they are treated as a unit while computing ROLLUP or CUBE operators.
- When used with ROLLUP or CUBE, composite columns require skipping aggregation across certain levels and/or attributes.



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

A composite column is a collection of columns that are treated as a unit during the computation of groupings. You specify the columns in parentheses as in the following statement: `ROLLUP (a, (b, c))`

Here `(b, c)` forms a composite column and is treated as a unit. In general, composite columns are useful in ROLLUP, CUBE, and GROUPING SETS. For example, in CUBE or ROLLUP, composite columns require skipping aggregation across certain levels and/or attributes.

That is, GROUP BY `ROLLUP(a, (b, c))` is equivalent to:

```
GROUP BY a, b, c UNION ALL  
GROUP BY a UNION ALL  
GROUP BY ()
```

Here `(b, c)` is treated as a unit and ROLLUP is not applied across `(b, c)`. It is as though you have an alias—for example, `z` as an alias for `(b, c)`, and the GROUP BY expression reduces to: GROUP BY `ROLLUP(a, z)`.

Note: GROUP BY `()` is typically a SELECT statement with NULL values for the columns `a` and `b` and only the aggregate function. It is generally used for generating grand totals.

```
SELECT    NULL, NULL, aggregate_col  
FROM      <table_name>  
GROUP BY  () ;
```

Compare this with the normal ROLLUP as in:

```
GROUP BY ROLLUP(a, b, c)
```

This would be:

```
GROUP BY a, b, c UNION ALL
GROUP BY a, b UNION ALL
GROUP BY a UNION ALL
GROUP BY ()
```

Similarly:

```
GROUP BY CUBE((a, b), c)
```

This would be equivalent to:

```
GROUP BY a, b, c UNION ALL
GROUP BY a, b UNION ALL
GROUP BY c UNION ALL
GROUP BY ()
```

The following table shows the GROUPING SETS specification and the equivalent GROUP BY specification.

GROUPING SETS Statements	Equivalent GROUP BY Statements
GROUP BY GROUPING SETS(a, b, c)	GROUP BY a UNION ALL GROUP BY b UNION ALL GROUP BY c
GROUP BY GROUPING SETS(a, b, (b, c)) <i>(The GROUPING SETS expression has a composite column.)</i>	GROUP BY a UNION ALL GROUP BY b UNION ALL GROUP BY b, c
GROUP BY GROUPING SETS((a, b, c))	GROUP BY a, b, c
GROUP BY GROUPING SETS(a, (b), ())	GROUP BY a UNION ALL GROUP BY b UNION ALL GROUP BY ()
GROUP BY GROUPING SETS (a,ROLLUP(b, c)) <i>(The GROUPING SETS expression has a composite column.)</i>	GROUP BY a UNION ALL GROUP BY ROLLUP(b, c)

Composite Columns: Example

```
SELECT department_id, job_id, manager_id,
       SUM(salary)
  FROM hr.employees
 GROUP BY ROLLUP( department_id, (job_id, manager_id));
```

DEPARTMENT_ID	JOB_ID	MANAGER_ID	SUM(SALARY)
SA_REP		149	7000
			7000
10	AD_ASST	101	4400
		10	4400
20	MK_MAN	100	13000
20	MK_REP	201	6000
		20	19000
40	FI_MGR	101	12008
41	FT_ACCOUNT	108	39600
42	100		51608
43	AC_MGR	101	12008
44	AC_ACCOUNT	205	8300
45	110		20308
46			691416

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The query in the slide example results in the Oracle server computing the following groupings:

- (department_id, job_id, manager_id)
- (department_id)
- Grand total

If you are interested only in specific groups, you cannot limit the calculation to those groupings without using composite columns. With composite columns, this is possible by treating JOB_ID and MANAGER_ID columns as a single unit while rolling up. Columns enclosed in parentheses are treated as a unit while computing ROLLUP and CUBE. This is illustrated in the example in the slide. By enclosing the JOB_ID and MANAGER_ID columns in parentheses, you instruct the Oracle server to treat JOB_ID and MANAGER_ID as a single unit—that is, a composite column.

The example in the slide computes the following groupings:

- (department_id, job_id, manager_id)
- (department_id)
- ()

The example in the slide displays the following:

- Total salary for every department, job, and manager (labeled 1)
- Total salary for every department (labeled 2)
- Grand total (labeled 3)

The example in the slide can also be written as:

```
SELECT      department_id, job_id, manager_id, SUM(salary)
FROM        hr.employees
GROUP BY    department_id, job_id, manager_id
UNION      ALL
SELECT      department_id, TO_CHAR(NULL), TO_NUMBER(NULL),  SUM(salary)
FROM hr.employees
GROUP BY    department_id
UNION ALL
SELECT      TO_NUMBER(NULL),  TO_CHAR(NULL), TO_NUMBER(NULL),  SUM(salary)
FROM hr.employees
GROUP BY  () ;
```

In the absence of an optimizer that looks across query blocks to generate the execution plan, the preceding query would need three scans of the base table, HR.EMPLOYEES. This could be very inefficient. Therefore, the use of composite columns is recommended.

Concatenated Groupings

- Concatenated groupings offer a concise way to generate useful combinations of groupings.
- To specify concatenated grouping sets, you separate multiple grouping sets, ROLLUP, and CUBE operations with commas so that the Oracle server combines them into a single GROUP BY clause.
- The result is a cross-product of groupings from each GROUPING SET.

```
GROUP BY GROUPING SETS(a, b), GROUPING SETS(c, d)
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Concatenated groupings offer a concise way to generate useful combinations of groupings. The concatenated groupings are specified by listing multiple grouping sets, CUBEs, and ROLLUPs, and separating them with commas. The following is an example of concatenated grouping sets:

```
GROUP BY GROUPING SETS(a, b), GROUPING SETS(c, d)
```

This SQL example defines the following groupings:

```
(a, c), (a, d), (b, c), (b, d)
```

Concatenation of grouping sets is very helpful for these reasons:

- **Ease of query development:** You need not manually enumerate all groupings.
- **Use by applications:** SQL generated by online analytical processing (OLAP) applications often involves concatenation of grouping sets, with each GROUPING SET defining groupings needed for a dimension.

Concatenated Groupings: Example

```
SELECT      department_id, job_id, manager_id,
            SUM(salary)
FROM        hr.employees
GROUP BY    department_id,
            ROLLUP (job_id),
            CUBE (manager_id);
```

DEPARTMENT_ID	JOB_ID	MANAGER_ID	SUM(SALARY)
	SA_REP	149	7000
10	AD_ASST	101	4400
		50	22100
		90	24000
		20	13000
			7000

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In the slide example, the total salary for each of the following groups is calculated as shown in the partial result set in the slide example:

- Grouping 1: (department_id, job_id, manager_id)
- Grouping 2: (department_id, manager_id)
- Grouping 3: (department_id)
- Grouping 4: (department_id, job_id)

Note: The following row is for employee_id 178 who is not assigned a department_id and is displayed because you are also grouping by department_id:

			7000
--	--	--	------

Quiz

Which of the following group function returns a number of non-NULL rows for the given expression?

- a. COUNT
- b. SUM
- c. AVG
- d. STDDEV



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Answer: a

Summary

In this lesson, you should have learned how to:

- Use the ROLLUP operation to produce subtotal values
- Use the CUBE operation to produce cross-tabulation values
- Use the GROUPING function to identify the row values created by ROLLUP or CUBE
- Use the GROUPING_ID function to identify the GROUP BY level of a row in a query
- Use GROUPING SETS to produce a single result set
- Work with composite columns
- Use concatenated groupings to generate useful combinations of groupings



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

- ROLLUP and CUBE are extensions of the GROUP BY clause. ROLLUP is used to display subtotal and grand total values. CUBE is used to display cross-tabulation values.
- The GROUPING function enables you to determine whether a row is an aggregate produced by a CUBE or ROLLUP operator.
- The GROUPING_ID function enables you to determine the GROUP BY level of a particular row in a query.
- With the GROUPING SETS syntax, you can define multiple groupings in the same query. GROUP BY computes all the groupings specified and combines them with UNION ALL.
- Within the GROUP BY clause, you can combine expressions in various ways:
 - To specify composite columns, you group columns within parentheses so that the Oracle server treats them as a unit while computing ROLLUP or CUBE operations.
 - To specify concatenated grouping sets, you separate multiple grouping sets, ROLLUP, and CUBE operations with commas so that the Oracle server combines them into a single GROUP BY clause. The result is a cross-product of groupings from each grouping set.

Practice 2 Overview

This practice covers using:

- ROLLUP operators
- CUBE operators
- GROUPING functions
- GROUPING SETS



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In this practice, you use the ROLLUP and CUBE operators as extensions of the GROUP BY clause. You also use GROUPING SETS. In this practice, you use the HR schema.

Hierarchical Retrieval

ORACLE®

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Objectives

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

- Interpret the concept of a hierarchical query
- Create a tree-structured report
- Format hierarchical data
- Exclude branches from the tree structure

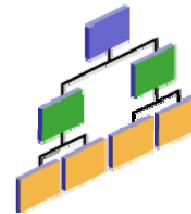


Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In this lesson, you learn how to use hierarchical queries to create tree-structured reports.

Using Hierarchical Queries

- You can use hierarchical queries to retrieve data based on a natural hierarchical relationship between rows in a table.
- A hierarchical query is possible only when a relationship exists between rows in a table.
- A process called “tree walking” enables you to construct hierarchy between the rows in a single table.



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You can use hierarchical queries to retrieve data based on a natural hierarchical relationship between rows in a table. A relational database does not store records in a hierarchical way. However, where a hierarchical relationship exists between the rows of a single table, a process called “tree walking” enables the hierarchy to be constructed. A hierarchical query is a method of reporting, with the branches of a tree in a specific order.

Imagine a family tree with the oldest members of the family found close to the base or trunk of the tree and the youngest members representing leaves of the tree. Branches can have their own branches, and so on.

See the example in the next slide.

Using Hierarchical Queries: Example

Sample Data from the EMPLOYEES Table

In the EMPLOYEES table in the HR schema, Kochhar, De Haan, and Hartstein report to the MANAGER_ID 100, which is King's EMPLOYEE_ID.

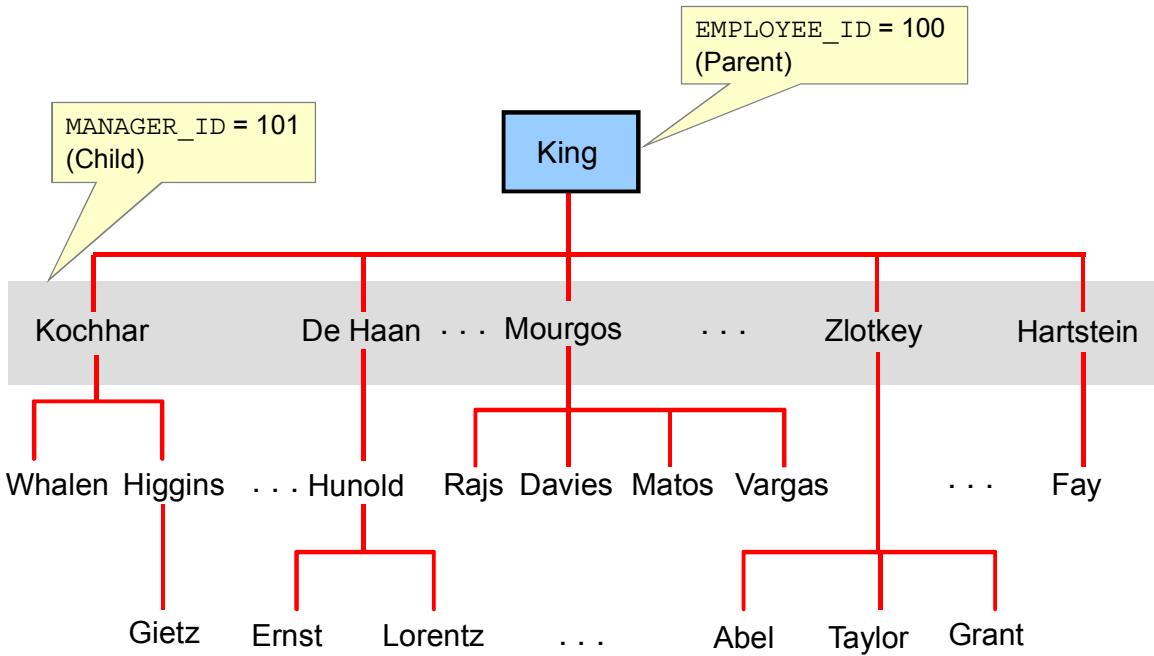
EMPLOYEE_ID	LAST_NAME	JOB_ID	MANAGER_ID
1	100 King	AD_PRES	(null)
2	101 Kochhar	AD_VP	100
3	102 De Haan	AD_VP	100
4	103 Hunold	IT_PROG	102
5	104 Ernst	IT_PROG	103
6	107 Lorentz	IT_PROG	103
...			
16	200 Whalen	AD_ASST	101
17	201 Hartstein	MK_MAN	100
18	202 Fay	MK_REP	201
19	205 Higgins	AC_MGR	101
20	206 Gietz	AC_ACCOUNT	205



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

A hierarchical query is possible when a parent-child relationship exists between rows in a table. In the slide example using the EMPLOYEES table, you see that Kochhar, De Haan, and Hartstein report to the MANAGER_ID 100, which is King's EMPLOYEE_ID.

Natural Tree Structure



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The EMPLOYEES table has a tree structure representing the management reporting line. The hierarchy can be created by looking at the relationship between equivalent values in the EMPLOYEE_ID and MANAGER_ID columns. This relationship can be exploited by joining the table to itself. The MANAGER_ID column contains the employee number of the employee's manager.

The CONNECT BY and START WITH clauses of a SQL SELECT statement enable you to control:

- The direction in which the hierarchy is walked
- The starting point inside the hierarchy

Note: The slide shows an inverted tree structure of a sample of the management hierarchy of the employees in the HR.EMPLOYEES table.

Hierarchical Queries: Syntax

```
SELECT [LEVEL], column, expr...
FROM table
[WHERE condition(s)]
[START WITH condition(s)]
[CONNECT BY PRIOR condition(s)] ;
```

condition:

```
expr comparison_operator expr
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Hierarchical queries can be identified by the presence of the CONNECT BY and START WITH clauses.

In the syntax:

SELECT	Is the standard SELECT clause
LEVEL	For each row returned by a hierarchical query, the LEVEL pseudocolumn returns 1 for a root row, 2 for a child of a root, and so on.
FROM <i>table</i>	Specifies the table, view, or snapshot containing the columns.
WHERE	Restricts the rows returned by the query without affecting other rows of the hierarchy
<i>condition</i>	Is a comparison with expressions
START WITH	Specifies the root rows of the hierarchy (where to start). This clause is required for a hierarchical query.
CONNECT BY	Specifies the relationship between parent rows and child rows of the hierarchy. This clause is required for a hierarchical query.

Walking the Tree: Specifying the Starting Point

- Use the START WITH clause to specify the starting point, that is, the row or rows to be used as the root of the tree:
 - Specifies the condition that must be met
 - Accepts any condition that is valid in a WHERE clause

```
START WITH column1 = value
```

- For example, using the HR.EMPLOYEES table, start with the employee whose last name is Kochhar.

```
...
START WITH last_name = 'Kochhar'
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The row or rows to be used as the root of the tree are determined by the START WITH clause. The START WITH clause accepts any condition that is valid in a WHERE clause.

Examples

- Using the EMPLOYEES table, start with King, the president of the company.
- ... START WITH manager_id IS NULL
- Using the EMPLOYEES table, start with employee Kochhar. A START WITH condition can contain a subquery.

```
... START WITH employee_id = (SELECT employee_id
                             FROM   employees
                             WHERE  last_name = 'Kochhar')
```

If the START WITH clause is omitted, the tree walk is started with all the rows in the table as root rows.

Note: The CONNECT BY and START WITH clauses are not American National Standards Institute (ANSI) SQL standard.

Walking the Tree: Specifying the Direction of the Query

- The direction of the query is determined by the CONNECT BY PRIOR column placement.
- The PRIOR operator refers to the parent row.

```
CONNECT BY PRIOR column1 = column2
```

- For example, you can walk the tree from top down using the EMPLOYEES table as follows:

```
...  
CONNECT BY PRIOR employee_id = manager_id  
...
```

Parent key Child key

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The direction of the query (from an employee to the employee's direct reports or from an employee to the employee's manager) is determined by the CONNECT BY PRIOR column placement. The PRIOR operator refers to the parent row. To find the rows in a given level, the Oracle server evaluates the PRIOR expression for the previous level. Rows for which the condition is true are returned. Rows are automatically sorted so that the connecting PRIOR row will always be the most recent row from the previous level.

Examples

- Walk from the top down using the EMPLOYEES table. Define a hierarchical relationship in which the EMPLOYEE_ID value of the parent row is equal to the MANAGER_ID value of the child row:


```
... CONNECT BY PRIOR employee_id = manager_id
```
- Walk from the bottom up using the EMPLOYEES table:


```
... CONNECT BY PRIOR manager_id = employee_id
```
- The PRIOR operator does not necessarily need to be coded immediately following CONNECT BY. Thus, the following CONNECT BY clause gives the same result as the one in example 2:


```
... CONNECT BY employee_id = PRIOR manager_id.
```

Hierarchical Query Example: Using the CONNECT BY Clause

```
SELECT employee_id, last_name, manager_id  
FROM hr.employees  
CONNECT BY PRIOR employee_id = manager_id;
```

EMPLOYEE_ID	LAST_NAME	MANAGER_ID
101	Kochhar	100
108	Greenberg	101
109	Faviet	108
110	Chen	108
111	Sciarrra	108
112	Urman	108
113	Popp	108
200	Whalen	101
203	Mavris	101
204	Baer	101
205	Higgins	101
206	Gietz	205
102	De Haan	100
103	Hunold	102
104	Ernst	103
105	Austin	103

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.



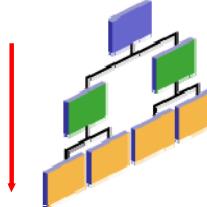
The hierarchical query example in the slide uses the CONNECT BY clause to define the relationship between employees and managers.

Specifying the Direction of the Query: From the Top Down

```
SELECT  last_name||' reports to '||  
PRIOR  last_name "Walk Top Down"  
FROM    hr.employees  
START   WITH manager_id is null  
CONNECT BY PRIOR employee_id = manager_id ;
```

Walk Top Down
1 King reports to
2 Kochhar reports to King
3 Greenberg reports to Kochhar

105 Johnson reports to Zlotkey
106 Hartstein reports to King
107 Fay reports to Hartstein



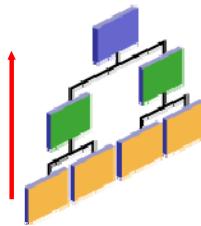
Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Walking from the top down, display the names of the employees and their manager. Use employee with manager_id as NULL as the starting point of the search. In the slide example, the PRIOR keyword is used in the SELECT clause to display the last_name from the parent record.

Specifying the Direction of the Query: From the Bottom Up

```
SELECT employee_id, last_name, job_id, manager_id  
FROM hr.employees  
START WITH employee_id = 101  
CONNECT BY PRIOR manager_id = employee_id ;
```

	EMPLOYEE_ID	LAST_NAME	JOB_ID	MANAGER_ID
1	101	Kochhar	AD_VP	100
2	100	King	AD_PRES	



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The example in the slide displays a list of managers starting with the employee whose employee ID is 101.

Computation Using the WITH Clause

- The WITH clause enables you to reuse a query block in a SELECT statement.
- Oracle stores the results of a query block in the user's temporary tablespace.

```
WITH org_chart(employee_last_name, employee_id,
manager_last_name)
AS
  (SELECT last_name, employee_id, null
   FROM hr.employees
  WHERE last_name = 'King'
  UNION ALL
    SELECT new.last_name, new.employee_id,
           previous.employee_last_name
      FROM org_chart previous, hr.employees new
     WHERE previous.employee_id = new.manager_id)
SEARCH DEPTH FIRST BY employee_id set ord1
SELECT employee_last_name || ' reports to ' || manager_last_name
  "Walk Top Down"
  FROM org_chart;
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The WITH clause (formally known as `subquery_factoring_clause`) enables you to reuse the same query block in a SELECT statement, when it occurs more than once within a complex query. The WITH clause is part of the SQL-99 standard. It is useful when a query has multiple references to the same query block. Using the WITH clause, Oracle retrieves the results of a query block and stores them in the user's temporary tablespace.

The example in the slide provides an alternative to the top-down query discussed in an earlier slide. The query uses the WITH clause to display the names of the employees and their manager in the top-down direction, starting with employee King.

Using the CONNECT_BY_ISLEAF Pseudocolumn

- The CONNECT_BY_ISLEAF pseudocolumn returns 1 if the current row is a leaf of the tree defined by the CONNECT BY condition.
- It indicates whether a given row can be further expanded to show more of the hierarchy.

```
SELECT last_name "Employee",
       CONNECT_BY_ISLEAF "IsLeaf",
       LEVEL, SYS_CONNECT_BY_PATH(last_name, '/') "Path"
  FROM employees
 WHERE LEVEL <= 3 AND department_id = 80
   START WITH employee_id = 100
 CONNECT BY PRIOR employee_id = manager_id
        AND LEVEL <= 4;
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

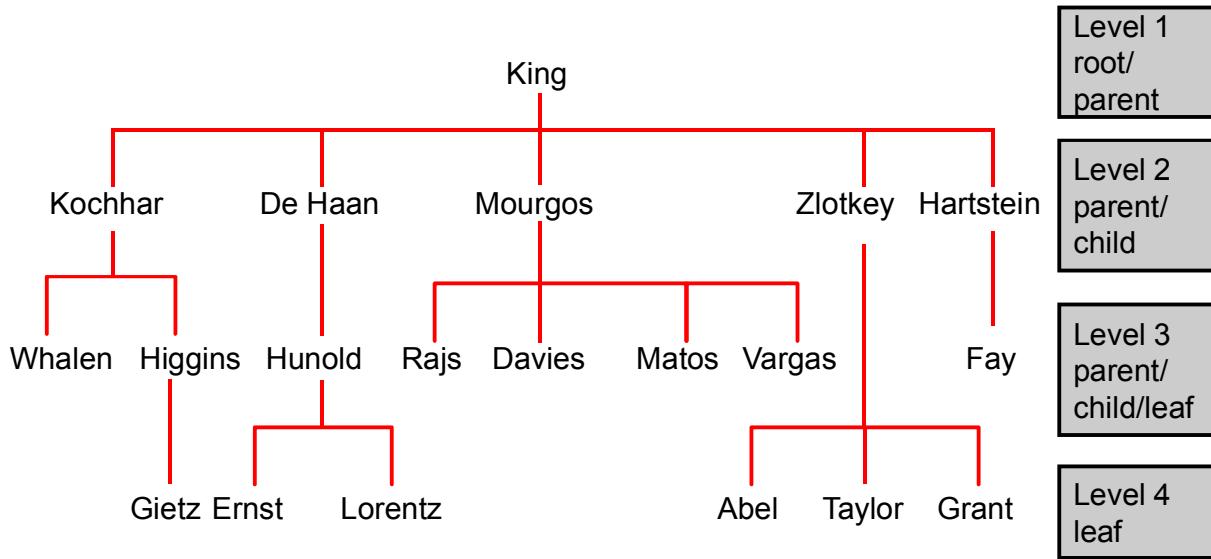
The CONNECT_BY_ISLEAF pseudocolumn is used to identify leaf nodes in a hierarchical tree. It returns 1 if the current row is a leaf node. Otherwise, it returns 0. Note that CONNECT_BY_ISLEAF considers only the tree defined by the CONNECT_BY condition, and not that of the underlying table data.

The example, in the slide, shows the first three levels of the employees table (LEVEL <= 4). It indicates if each row is a leaf row (indicated by 1 in the IsLeaf column) or whether it has child rows (indicated by 0 in the IsLeaf column).

The result of the example is as follows:

	Employee	IsLeaf	LEVEL	Path
1	Russell	0	2	/King/Russell
2	Tucker	1	3	/King/Russell/Tucker
3	Bernstein	1	3	/King/Russell/Bernstein
4	Hall	1	3	/King/Russell/Hall
5	Olsen	1	3	/King/Russell/Olsen
6	Cambrault	1	3	/King/Russell/Cambrault
.

Using the LEVEL Pseudocolumn



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You can explicitly show the level of a row in the hierarchy by using the `LEVEL` pseudocolumn. This makes your report more readable. The forks, where one or more branches split away from a larger branch are called “nodes,” and the end of a branch is called a “leaf” or a “leaf node.” The diagram in the slide shows the nodes of the inverted tree with their `LEVEL` values. For example, employee Higgins is a parent and a child, whereas employee Davies is a child and a leaf.

LEVEL Pseudocolumn

Value	Level
1	A root node
2	A child of a root node
3	A child of a child, and so on

In the slide, King is the root or parent (`LEVEL = 1`). Kochhar, De Haan, Mourgos, Zlotkey, Hartstein, Higgins, and Hunold are children and also parents (`LEVEL = 2` and `Level = 3`). Whalen, Rajas, Davies, Matos, Vargas, Gietz, Ernst, Lorentz, Abel, Taylor, Grant, and Fay are children and leaves (`LEVEL = 3` and `LEVEL = 4`).

Note: A root node is the highest node within an inverted tree. A child node is any nonroot node. A parent node is any node that has children. A leaf node is any node without children. The number of levels returned by a hierarchical query may be limited by available user memory.

Using the LEVEL Pseudocolumn: Example

```
SELECT employee_id, last_name, manager_id, LEVEL
FROM hr.employees
START WITH employee_id = 100
CONNECT BY PRIOR employee_id = manager_id
ORDER siblings BY last_name;
```

EMPLOYEE_ID	LAST_NAME	MANAGER_ID	LEVEL
100	King		1
148	Cambrault	100	2
172	Bates	148	3
169	Bloom	148	3
170	Fox	148	3
173	Kumar	148	3
168	Ozer	148	3
171	Smith	148	3
102	De Haan	100	2
103	Hunold	102	3
105	Austin	103	4
104	Ernst	103	4
107	Lorentz	103	4
106	Pataballa	103	4



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The hierarchical query example in the slide uses the CONNECT BY clause to define the relationship between employees and managers. In addition, the example uses the LEVEL pseudocolumn to show parent and child rows.

Using the ORDER SIBLINGS BY Clause

In a hierarchical query, do not specify either ORDER BY or GROUP BY, because they will override the hierarchical order of the CONNECT BY results. If you want to order rows of siblings of the same parent, then use the ORDER SIBLINGS BY clause.

The SIBLINGS keyword is valid only if you also specify the hierarchical_query_clause (CONNECT BY). ORDER SIBLINGS BY preserves any ordering specified in the hierarchical query clause and then applies the order_by_clause to the siblings of the hierarchy.

The example in the previous slide can be rewritten by using the SYS_CONNECT_BY_PATH function to determine the whole path for each row:

```
SELECT employee_id, last_name,
SUBSTR(SYS_CONNECT_BY_PATH(employee_id, '<='), 3) nodes,
SUBSTR(SYS_CONNECT_BY_PATH(last_name, '>='), 3) names,
PRIOR last_name "Manager"
FROM employees
CONNECT BY PRIOR employee_id=manager_id
START WITH employee_id=100
ORDER BY 1
/
```

The following is the output of this code example:

	EMPLOYEE_ID	LAST_NAME	NODES	NAME	Manager
1	100 King	100		King	
2	101 Kochhar	100<=101		King=>Kochhar	King
3	102 De Haan	100<=102		King=>De Haan	King
4	103 Hunold	100<=102<=103		King=>De Haan=>Hunold	De Haan
5	104 Ernst	100<=102<=103<=104		King=>De Haan=>Hunold=>Ernst	Hunold
6	105 Austin	100<=102<=103<=105		King=>De Haan=>Hunold=>Austin	Hunold
7	106 Pataballa	100<=102<=103<=106		King=>De Haan=>Hunold=>Pataballa	Hunold
8	107 Lorentz	100<=102<=103<=107		King=>De Haan=>Hunold=>Lorentz	Hunold
...					

Formatting Hierarchical Reports by Using LEVEL and LPAD

Create a report displaying company management levels beginning with the highest level and indenting each of the subsequent levels.

```
SELECT LPAD(last_name, LENGTH(last_name)+(LEVEL*2)-2, '_') AS org_chart
FROM hr.employees
START WITH first_name = 'Steven' AND last_name = 'King'
CONNECT BY PRIOR employee_id = manager_id
ORDER SIBLINGS BY last_name;
```



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The nodes in a tree are assigned level numbers from the root. Use the LPAD function in conjunction with the LEVEL pseudocolumn to display a hierarchical report as an indented tree.

In the example in the slide, note the following:

- LPAD(*char1, n [, char2]*) returns *char1*, left-padded to length *n* with the sequence of characters in *char2*. The argument *n* is the total length of the string returned by the database.
- LPAD(*last_name, LENGTH(last_name)+(LEVEL*2)-2, '_'*) defines the display format.
- *char1* is the LAST_NAME. *n* is the length of the LAST_NAME+ (LEVEL*2)-2. *char2* is '_'.

That is, this tells SQL to take the LAST_NAME and left-pad it with the '_' character until the padding itself has a length of (LEVEL*2) - 2.

For King, LEVEL = 1. Therefore, $(2 * 1) - 2 = 2 - 2 = 0$. So King does not get padded with any '_' character and is not indented.

For Kochhar, LEVEL = 2. Therefore, $(2 * 2) - 2 = 4 - 2 = 2$. So Kochhar gets padded with two '_' characters and is displayed indented.

The rest of the records in the EMPLOYEES table are displayed similarly.

Note: The ORDER SIBLINGS BY clause displays the employees' last names in alphabetical order at each level.

ORG_CHART	
1	King
2	Cambrault
3	Bates
4	Bloom
5	Fox
6	Kumar
7	Ozer
8	Smith
9	De Haan
10	Hunold
11	Austin
12	Ernst
13	Lorentz
14	Pataballa
15	Errazuriz
16	Ande
17	Banda
18	Greene
19	Lee
20	Marvins
21	Vishney
22	Fripp
23	Atkinson
24	Bissot
25	Bull
26	Cabrio
27	Dellinger
28	Marlow
29	Olson
30	Sarchand
31	Hartstein
32	Fay
33	Kaufling

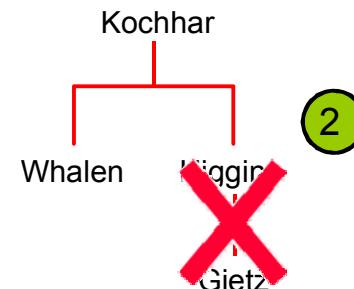
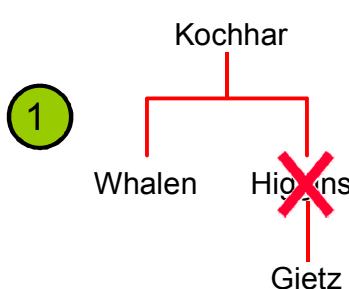
Pruning Nodes and Branches

- Use the WHERE clause to eliminate a node.

```
...  
WHERE last_name != 'Higgins'
```

- Use the CONNECT BY clause to eliminate a branch.

```
...  
CONNECT BY PRIOR employee_id = manager_id  
AND last_name != 'Higgins'
```



ORACLE

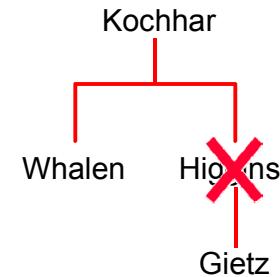
Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You can use the WHERE and CONNECT BY clauses to prune the tree (that is, to control which nodes or rows are displayed).

Pruning Branches: Example 1: Eliminating a Node

```
SELECT department_id, employee_id, last_name, job_id,
       salary
  FROM hr.employees
 WHERE last_name != 'Higgins'
 START WITH manager_id IS NULL
CONNECT BY PRIOR employee_id = manager_id;
```

	DEPARTMENT_ID	EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
31	50	181	Fleaur	SH_CLERK	3100
37	110	206	Gietz	AC_ACCOUNT	8300
38	50	199	Grant	SH_CLERK	2600
39		178	Grant	SA_REP	7000
40	100	108	Greenberg	FL_MGR	12000
41	80	163	Greene	SA_REP	9500
42	80	152	Hall	SA_REP	9000
43	20	201	Hartstein	MK_MAN	13000
44	30	118	Himuro	PU_CLERK	2600
45	60	103	Hunold	IT_PROG	9000
46	80	175	Hutton	SA_REP	8800
47	80	179	Johnson	SA_REP	6200
...					
105	10	200	Whalen	AD_ASST	4400
106	80	149	Zlotkey	SA_MAN	10500



ORACLE

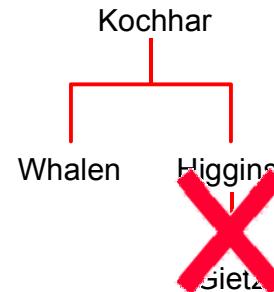
Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Starting at the root, walk from the top down, eliminate employee Higgins in the result, but process the child rows. Note that the slide example returns 106 rows out of 107 because employee Higgins is eliminated from the results set.

Pruning Branches: Example 2: Eliminating a Branch

```
SELECT department_id, employee_id, last_name, job_id,
       salary
  FROM hr.employees
 START WITH manager_id IS NULL
CONNECT BY PRIOR employee_id = manager_id
AND      last_name != 'Higgins';
```

#	DEPARTMENT_ID	EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
32	60	170	Fox	SA_REP	9600
33	50	121	Fripp	ST_MAN	8200
34	50	190	Gates	SH_CLERK	2900
35	50	135	Gee	ST_CLERK	2400
36	50	183	Geoni	SH_CLERK	2800
37	50	199	Grant	SH_CLERK	2600
38		178	Grant	SA REP	7000
39	100	108	Greenberg	FI_MGR	12000
40	80	163	Greene	SA REP	9500
41	80	152	Hall	SA REP	9000
42	20	201	Hartstein	MK_MAN	13000
43	30	118	Himuro	PLU_CLERK	2600
44	60	103	Hunold	IT_PROG	9000
45	80	175	Hutton	SA REP	6800
46	80	179	Johnson	SA REP	6200
...					
	105	80	149 Zlotkey	SA MAN	10500



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Starting at the root, walk from the top down, and eliminate employee Higgins and all child rows.

To find out the employees that report to employee Higgins, issue the following command:

```
SELECT last_name, employee_id
  FROM employees
 WHERE manager_id = (SELECT employee_id
                      FROM employees
                     WHERE last_name = 'Higgins');
```

LAST_NAME	EMPLOYEE_ID
Gietz	206
1 rows selected	

The results in the slide display the partial results where the entire branch of Higgins is eliminated. Note that the slide example returns 105 rows out of 107 because employee Higgins and the child, Gietz, are both eliminated from the results set.

Quiz

The CONNECT_BY_ISLEAF pseudocolumn returns 0 if the current row is a leaf node.

- a. True
- b. False



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Answer: b

The CONNECT_BY_ISLEAF pseudocolumn returns 0 if the current row is not a leaf node.

Summary

In this lesson, you should have learned how to:

- Interpret the concept of a hierarchical query
- Create a tree-structured report
- Format hierarchical data
- Exclude branches from the tree structure



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You can use hierarchical queries to retrieve data based on a natural hierarchical relationship between rows in a table. The `LEVEL` pseudocolumn counts how far down a hierarchical tree you have traveled. You can specify the direction of the query by using the `CONNECT BY` clause. You can specify the starting point by using the `START WITH` clause. You can use the `WHERE` and `CONNECT BY` clauses to prune the tree branches.

Practice 3 Overview

This practice covers the following topics:

- Walking through a tree
- Producing an indented report by using the LEVEL pseudocolumn
- Pruning the tree structure



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In this practice, you gain practice in producing hierarchical reports using the HR schema.

Working with Regular Expressions

4

ORACLE®

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Objectives

After completing this lesson, you should be able to use regular expressions to search for, match, and replace strings.



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In this lesson, you learn to use the regular expressions support feature that was introduced in Oracle Database 10g. In addition, you learn about the new regular expressions features in the later versions of Oracle Database.

Lesson Agenda

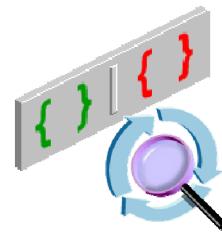
- Introduction to regular expressions and using metacharacters with regular expressions
- Using the regular expressions functions:
 - REGEXP_LIKE
 - REGEXP_REPLACE
 - REGEXP_INSTR
 - REGEXP_SUBSTR
 - REGEXP_COUNT
- Accessing subexpressions
- Regular expressions and check constraints



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

What Are Regular Expressions?

- You can use regular expressions to search for (and manipulate) simple and complex patterns in string data.
- You can use a set of SQL functions and conditions to search for and manipulate strings in SQL and PL/SQL.
- You specify a regular expression by using:
 - Metacharacters, which are operators that specify the search algorithms
 - Literals, which are the characters that you search for



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Oracle Database 10g introduced support for regular expressions. The implementation complies with the Portable Operating System for UNIX (POSIX) standard, controlled by the Institute of Electrical and Electronics Engineers (IEEE), for ASCII data-matching semantics and syntax. Oracle's multilingual capabilities extend the matching capabilities of the operators beyond the POSIX standard. Regular expressions describe both simple and complex patterns for searching and manipulating.

String manipulation and searching contribute to a large percentage of the logic within a web-based application. Usage ranges from the simple (such as finding the phrase "San Francisco" in a specified text), to the complex task of extracting all URLs from the text, and the more complex task of finding all words whose every second character is a vowel.

When coupled with native SQL, the use of regular expressions allows for very powerful search and manipulation operations on any data stored in an Oracle database. You can use this feature to easily solve problems that would otherwise involve complex programming.

Note: You can also use Oracle Text, which enables you to build text query applications and document classification applications. Oracle Text provides indexing, word and theme searching, and viewing capabilities for text. See the *Oracle Text Application Developer's Guide*.

Benefits of Using Regular Expressions

Regular expressions enable you to implement complex match logic in the database with the following benefits:

- Avoid intensive string processing of SQL results by middle-tier applications
- Eliminate the need to code data validation logic on the client
- Make string manipulations more powerful and easier



ORACLE®

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Regular expressions are a powerful text-processing component of programming languages such as PERL and Java. For example, a PERL script can process each HTML file in a directory, read its contents into a scalar variable as a single string, and then use regular expressions to search for URLs in the string. One reason for many developers writing in PERL is that it has a robust pattern-matching functionality. Oracle's support of regular expressions enables developers to implement complex match logic in the database. This technique is useful for the following reasons:

- By centralizing match logic in Oracle Database, you avoid intensive string processing of SQL results sets by middle-tier applications. The SQL regular expression functions move the processing logic closer to the data, thereby providing a more efficient solution.
- Before Oracle Database 10g, developers often coded data validation logic on the client, requiring the same validation logic to be duplicated for multiple clients. Using server-side regular expressions to enforce constraints solves this problem.
- The built-in SQL and PL/SQL regular expression functions and conditions make string manipulations more powerful and less cumbersome.

Using the Regular Expressions Functions and Conditions in SQL and PL/SQL

Function or Condition Name	Description
REGEXP_LIKE condition	Is similar to the <code>LIKE</code> operator, but performs regular expression matching instead of simple pattern matching
REGEXP_REPLACE function	Searches for a regular expression pattern and replaces it with a replacement string
REGEXP_INSTR function	Searches a string for a regular expression pattern and returns the position where the match is found
REGEXP_SUBSTR function	Searches for a regular expression pattern within a given string and extracts the matched substring
REGEXP_COUNT function	Returns an integer indicating the number of occurrences of the pattern



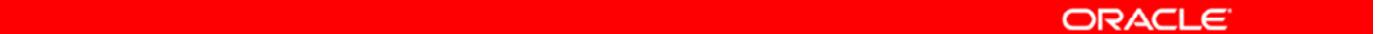
Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You can use the regular expressions SQL functions and conditions to search and manipulate strings using regular expressions. You can use these functions on a text literal, bind variable, or any column that holds character data such as CHAR, NCHAR, CLOB, NCLOB, NVARCHAR2, and VARCHAR2 (but not LONG).

- **REGEXP_LIKE:** This condition searches a character column for a pattern. Use this condition in the `WHERE` clause of a query to return rows matching the regular expression you specify.
- **REGEXP_REPLACE:** This function searches for a pattern in a character column and replaces each occurrence of that pattern with the pattern you specify.
- **REGEXP_INSTR:** This function searches a string for a given occurrence of a regular expression pattern. You specify which occurrence you want to find and the start position to search from. This function returns an integer indicating the position in the string where the match is found.
- **REGEXP_SUBSTR:** This function returns the actual substring matching the regular expression pattern you specify.
- **REGEXP_COUNT:** This function returns an integer indicating the number of occurrences of a pattern.

What Are Metacharacters?

- Metacharacters are characters that have a special meaning, such as a wildcard, a repeating character, a non-matching character, or a range of characters.
- For example, the `'^ (f | ht) tps? :$'` regular expression searches for the following from the beginning of the string:
 - The literals `f` or `ht` at the beginning of the string
 - The `t` literal
 - The `p` literal, optionally followed by the `s` literal
 - The colon “`:`” literal at the end of the string

**ORACLE**

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You can use several predefined metacharacter symbols in the pattern matching. A regular expression must be enclosed or wrapped within single quotation marks. This ensures that the entire expression is interpreted by the SQL function and can improve the readability of your code.

Note: For a complete list of the regular expressions' metacharacters, see the *Oracle Database Advanced Application Developer's Guide*.

Using Metacharacters with Regular Expressions

Syntax	Description
.	Matches any character in the supported character set, except NULL
+	Matches one or more occurrences
?	Matches zero or one occurrence
*	Matches zero or more occurrences of preceding subexpression
{m}	Matches exactly m occurrences of the preceding expression
{m, }	Matches at least m occurrences of the preceding subexpression
{m, n}	Matches at least m , but not more than n occurrences of the preceding subexpression
[...]	Matches any single character in the list within the brackets
	Matches one of the alternatives
(...)	Treats the enclosed expression within the parentheses as a unit. The subexpression can be a string of literals or a complex expression containing operators.

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

- Any character, “.”**: `a.b` matches the strings `abb`, `acb`, and `adb`, but not `acc`.
- One or more, “+”**: `a+` matches the strings `a`, `aa`, and `aaa`, but not `bbb`.
- Zero or one, “?”**: `ab?c` matches the strings `abc` and `ac`, but not `abbc`.
- Zero or more, “*”**: `ab*c` matches the strings `ac`, `abc`, and `abbc`, but not `abb`.
- Exact count, “[m]”**: `a{3}` matches the strings `aaa`, but not `aa`.
- At least count, “[m,]”**: `a{3, }` matches the strings `aaa` and `aaaa`, but not `aa`.
- Between count, “[m, n]”**: `a{3, 5}` matches the strings `aaa`, `aaaa`, and `aaaaa`, but not `aa`.
- Matching character list, “[...]"**: `[abc]` matches the first character in the strings `all`, `bill`, and `cold`, but does not match any characters in `doll`.
- Or, “|”**: `a | b` matches character `a` or `b`.
- Subexpression, “(...)"**: `(abc) ?def` matches the optional string `abc`, followed by `def`. The expression matches `abcdefghi` and `def`, but not `ghi`. The subexpression can be a string of literals or a complex expression containing operators.

Using Metacharacters with Regular Expressions

Syntax	Description
<code>^</code>	Matches the beginning of a string
<code>\$</code>	Matches the end of a string
<code>\</code>	Treats the subsequent metacharacter in the expression as a literal
<code>\n</code>	Matches the <i>n</i> th (1–9) preceding subexpression of whatever is grouped within parentheses. The parentheses cause an expression to be remembered; a backreference refers to it.
<code>\d</code>	A digit character
<code>[:class:]</code>	Matches any character belonging to the specified POSIX character class
<code>[^ :class:]</code>	Matches any single character <i>not</i> in the list within the brackets



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

- Beginning/end-of-line anchor, “^” and “\$”:** `^def` matches `def` in the string `defghi`, but does not match `def` in `abcdef`. `def$` matches `def` in the string `abcdef`, but does not match `def` in the string `defghi`.
- Escape character “\”:** `\+` searches for a `+`. It matches the plus character in the string `abc+def`, but does not match `Abcdef`.
- Backreference, “\n”:** `(abc|def)xy\1` matches the strings `abcyxabc` and `defxydef`, but does not match `abcxydef` or `abcyx`. A backreference enables you to search for a repeated string without knowing the actual string ahead of time. For example, the expression `^(.*)\1$` matches a line consisting of two adjacent instances of the same string.
- Digit character, “\d”:** The expression `^\[\d{3}\]\ \d{3}-\d{4}$` matches `[650] 555-1212`, but does not match `650-555-1212`.
- Character class, “[:upper:]”:** `[[:upper:]]+` searches for one or more consecutive uppercase characters. This matches `DEF` in the string `abcDEFghi`, but does not match the string `abcdefghi`.
- Nonmatching character list (or class), “[^ . . .]”:** `[^abc]` matches the character `d` in the string `abcdef`, but not the characters `a`, `b`, or `c`.

Lesson Agenda

- Introduction to regular expressions and using metacharacters with regular expressions
- Using the regular expressions functions:
 - REGEXP_LIKE
 - REGEXP_REPLACE
 - REGEXP_INSTR
 - REGEXP_SUBSTR
 - REGEXP_COUNT
- Accessing subexpressions
- Regular expressions and check constraints



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Regular Expressions Functions and Conditions: Syntax

```
REGEXP_LIKE      (source_char, pattern [,match_option])
```

```
REGEXP_INSTR (source_char, pattern [, position  
[, occurrence [, return_option  
[, match_option [, subexpr]]]])
```

```
REGEXP_SUBSTR (source_char, pattern [, position  
[, occurrence [, match_option  
[, subexpr]]]])
```

```
REGEXP_REPLACE(source_char, pattern [,replacestr  
[, position [, occurrence [, match_option]]]])
```

```
REGEXP_COUNT (source_char, pattern [, position  
[, occurrence [, match_option]]])
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The syntax for the regular expressions functions and conditions is as follows:

source_char: A character expression that serves as the search value

pattern: A regular expression, a text literal

occurrence: A positive integer indicating which occurrence of a pattern in `source_char` Oracle Server should search for. The default is 1.

position: A positive integer indicating the character of `source_char` where Oracle Server should begin the search. The default is 1.

return_option:

- 0: Returns the position of the first character of the occurrence (default)
- 1: Returns the position of the character following the occurrence

Replacestr: Character string replacing pattern

match_option:

- “c”: Uses case-sensitive matching (default)
- “i”: Uses non-case-sensitive matching
- “n”: Allows match-any-character operator
- “m”: Treats source string as multiple lines

Performing a Basic Search Using the REGEXP_LIKE Condition

```
REGEXP_LIKE(source_char, pattern [, match_parameter ])
```

```
SELECT first_name, last_name
FROM hr.employees
WHERE REGEXP_LIKE (first_name, '^Ste(v|ph)en$');
```

FIRST_NAME	LAST_NAME
Steven	King
Steven	Minkle
Stephen	Stiles
3 rows selected	



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

REGEXP_LIKE is similar to the LIKE condition, except that REGEXP_LIKE performs regular-expression matching instead of the simple pattern matching performed by LIKE. This condition evaluates strings by using characters as defined by the input character set.

Example of REGEXP_LIKE

In this query, against the EMPLOYEES table, all employees with first names containing Steven or Stephen are displayed. In the '^Ste(v|ph)en\$' expression used:

- ^ indicates the beginning of the sentence
- \$ indicates the end of the sentence
- | indicates either/or

Note: The code examples in this lesson use the HR schema.

Finding Patterns by Using the REGEXP_INSTR Function

```
REGEXP_INSTR (source_char, pattern [, position  
[, occurrence [, return_option  
[, match_option [, subexpr]]]])
```

```
SELECT street_address,  
REGEXP_INSTR(street_address,'[[[:alpha:]]') AS  
First_Alpha_Position  
FROM hr.locations;
```

STREET_ADDRESS	FIRST_ALPHA_POSITION
1297 Via Cola di Rie	6
93091 Calle della Testa	7
2017 Shinjuku-ku	6
9450 Kamiya-cho	6
<hr/>	
Rua Frei Caneca 1360	1
20 Rue des Corps-Saints	4
Murtenstrasse 921	1
Pieter Breughelstraat 837	1
Mariano Escobedo 9991	1
<hr/>	
23 rows selected	



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In this example, the REGEXP_INSTR function is used to search the street address to find the location of the first non-alphabetic character, regardless of whether it is in uppercase or lowercase. Note that [:<class>:] implies a character class and matches with any character from within that class; [:alpha:] matches with any alphabetic character. The results are displayed.

In the expression used in the query '[[[:alpha:]]':

- [starts the expression
- [:alpha:] indicates alpha character class
-] ends the expression

Note: The POSIX character class operator enables you to search for an expression within a character list that is a member of a specific POSIX character class. You can use this operator to search for specific formatting—such as uppercase characters—or you can search for special characters such as digits or punctuation characters. The full set of POSIX character classes is supported. Use the syntax [:class:], where class is the name of the POSIX character class to search for. The '[:upper:]+' regular expression searches for one or more consecutive uppercase characters.

Extracting Substrings by Using the REGEXP_SUBSTR Function

```
REGEXP_SUBSTR (source_char, pattern [, position  
[, occurrence [, match_option]])
```

```
SELECT REGEXP_SUBSTR(street_address , ' [^ ]+ ') "Road"  
FROM hr.locations;
```

ROAD
Via
Calle
Jabberwocky
Interiors
Zagora
Charade
Spadina
Boxwood
.
.
.
Rue
Breughelstraat
Escobedo

23 rows selected



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In this example, the road names are extracted from the LOCATIONS table. To do this, the contents in the STREET_ADDRESS column that are after the first space are returned by using the REGEXP_SUBSTR function. In the expression used in the query '`[^]+`':

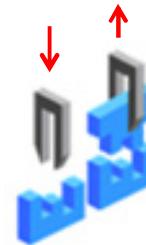
- [starts the expression
- ^ indicates NOT
- indicates space
-] ends the expression
- + indicates 1 or more
- indicates space

Replacing Patterns by Using the REGEXP_REPLACE Function

```
REGEXP_REPLACE(source_char, pattern [,replacestr  
[, position [, occurrence [, match_option]]]])
```

```
SELECT last_name,  
       REGEXP_REPLACE(phone_number, '\.', '-') AS phone  
  FROM hr.employees;
```

LAST_NAME	PHONE
King	515-123-4567
Kochhar	515-123-4568
De Haan	515-123-4569
Hunold	590-423-4567
Ernst	590-423-4568
Austin	590-423-4569
Pataballa	590-423-4560
Lorentz	590-423-5567
Greenberg	515-124-4569
...	
Higgins	515-123-8080
Gietz	515-123-8181
107 rows selected	



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Using the REGEXP_REPLACE function, you reformat the phone number to replace the period (.) delimiter with a dash (-) delimiter. Here is an explanation of each of the elements used in the regular expression example:

- phone_number is the source column.
- '\.' is the search pattern.
 - Use single quotation marks (' ') to search for the literal character period (.).
 - Use a backslash (\) to search for a character that is normally treated as a metacharacter.
- '-' is the replace string.

The partial results of the employees' last names and phone numbers before applying the REGEXP_REPLACE function is as follows:

```
SELECT last_name, phone_number  
  FROM hr.employees;
```

LAST_NAME	PHONE_NUMBER
King	515.123.4567
Kochhar	515.123.4568
De Haan	515.123.4569

...

Using the REGEXP_COUNT Function

```
REGEXP_COUNT (source_char, pattern [, position  
[, occurrence [, match_option]])
```

```
SELECT REGEXP_COUNT(  
    'ccacccctccactcctcacgttctcacctgtaaagcgtccctccatccccca  
    tgcccccttaccctgcagggtagagtaggctagaaaccagagagctccaagctccatct  
    gtggagaggtgccatcctgggctgcagagagaggagaattgccccaaagctgcctgc  
    agagcttcaccacccttagtctcacaaagccttgagttcatgcattcttgagtttc  
    accctgcccagcaggacactgcagcacccaaaggctcccaggagtagggttgcctc  
    aagaggcttgggtctgatggccacatcctggaattgtttcaagttgatggtcacag  
    ccctgaggcatgttagggcgatgcgctctgctctcctcctgaacc  
    tgaaccctctggctaccccagagcacccag', '[gtc]' ) AS Count  
  
FROM dual;
```

COUNT

4

1 rows selected

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The REGEXP_COUNT function evaluates strings using characters as defined by the input character set. It returns an integer indicating the number of occurrences of the pattern. If no match is found, then the function returns 0.

In the example in the slide, the number of occurrences for a DNA substring is determined using the REGEXP_COUNT function. The following example shows that the number of times the pattern 123 occurs in the string 123123123123 is three. The search starts from the second position of the string.

```
SELECT REGEXP_COUNT  
( '123123123123', -- source char or search value  
  
    '123',          -- regular expression pattern  
    2,              -- position where the search should start  
    'i')            -- match option (case insensitive)  
  
As Count  
FROM dual;
```

COUNT

3

1 rows selected

Lesson Agenda

- Introduction to regular expressions and using metacharacters with regular expressions
- Using the regular expressions functions:
 - REGEXP_LIKE
 - REGEXP_REPLACE
 - REGEXP_INSTR
 - REGEXP_SUBSTR
 - REGEXP_COUNT
- Accessing subexpressions
- Regular expressions and check constraints



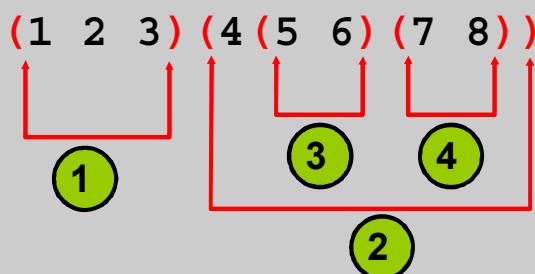
Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Working with Subexpressions

Examine this expression:

```
(1 2 3) (4 (5 6) (7 8))
```

The subexpressions are:



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Oracle Database provides a regular expressions support parameter to access a subexpression. In the example in the slide, a string of digits is shown. The parentheses identify the subexpressions within the string of digits. Reading from left to right and from the inner parentheses to the outer, the subexpressions in the string of digits are:

1. 123
2. 45678
3. 56
4. 78

You can search for any of these subexpressions with the `REGEXP_INSTR` and `REGEXP_SUBSTR` functions.

Using Subexpressions with Regular Expression Support: Example

```

SELECT
  REGEXP_INSTR
    ('0123456789',          2           -- source char or search value
     '(123)(4(56)(78))',   -- regular expression patterns
     1,                      -- position to start searching
     1,                      -- occurrence
     0,                      -- return option
     'i',                   -- case-insensitive
     1)                     -- sub-expression on which to search
  "Position"
  FROM dual;

```

Position
2
1 rows selected



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

REGEXP_INSTR and REGEXP_SUBSTR have an optional SUBEXPR parameter that lets you target a particular substring of the regular expression being evaluated.

In the example shown in the slide, you may want to search for the first subexpression pattern in your list of subexpressions. The example shown identifies several parameters for the REGEXP_INSTR function.

1. The string you are searching is identified.
2. The subexpressions are identified. The first subexpression is 123. The second subexpression is 45678, the third is 56, and the fourth is 78.
3. The third parameter identifies from which position to start searching.
4. The fourth parameter identifies the occurrence of the pattern you want to find. 1 means find the first occurrence.
5. The fifth parameter is the return option. This is the position of the first character of the occurrence. (If you specify 1, the position of the character following the occurrence is returned.)
6. The sixth parameter identifies whether your search should be case-sensitive or not.
7. The last parameter is the parameter added in Oracle Database 11g. This parameter specifies which subexpression you want to find. In the example in the slide, you search for the first subexpression, which is 123.

Why Access the *n*th Subexpression?

- DNA sequencing is a more realistic use.
- You may need to find a specific subpattern that identifies a protein needed for immunity in mouse DNA.

```
SELECT
    REGEXP_INSTR('ccaccttcctccactcctcacgttctcacctgtaaagcgccccccat
ccccatgcccccttaccctgcagggtagagtaggctagaaaccagagagctccaagctccatctgt
gagagggtccatccttggtgcagagagaggagaattgccccaaagctgcctgcagagcttcacc
acccttagtctcacaaaggccttgagttcatagcatttcttgagtttaccctgcccagcaggacac
tgcaagcacccaaaggcctccaggagttagggtgcctcaagaggctttgggtctatggccaca
tccttggaaatgtttcaagttcatggtcacagccctgaggcatgttagggcgtgggatgcgtctg
ctctgtctcccttcctgaaccctctggctacccagagcacttagagccag',
    '(gtc(tcac)(aaag))',
    1, 1, 0, 'i',
    1) "Position"
FROM dual;
```

Position
195

1 rows selected



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In life sciences, you may need to extract the offsets of subexpression matches from a DNA sequence for further processing. For example, you may need to find a specific protein sequence, such as the begin offset for the DNA sequence preceded by `gtc` and followed by `tcac` followed by `aaag`. To accomplish this goal, you can use the `REGEXP_INSTR` function, which returns the position where a match is found.

In the example in the slide, the position of the first subexpression (`gtc`) is returned. `gtc` appears starting in position 195 of the DNA string.

If you modify the slide example to search for the second subexpression (`tcac`), the query results in the following output. `tcac` appears starting in position 198 of the DNA string.

Position
198

If you modify the slide example to search for the third subexpression (`aaag`), the query results in the following output. `aaag` appears starting in position 202 of the DNA string.

Position
202

REGEXP_SUBSTR: Example

```

SELECT
  REGEXP_SUBSTR
  ('acgctgcactgca',      -- source char or search value
   'acg(.*)gca',          -- regular expression pattern
   1,                      -- position to start searching
   1,                      -- occurrence
   'i',                   -- match option (case insensitive)
   1)                     -- sub-expression
  "Value"
FROM dual;

```

Value

ctgcact
1 rows selected

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In the example shown in the slide:

1. acgctgcactgca is the source to be searched.
2. acg(.*)gca is the pattern to be searched. Find acg followed by gca with potential characters between the acg and the gca.
3. Start searching at the first character of the source.
4. Search for the first occurrence of the pattern.
5. Use non-case-sensitive matching on the source.
6. Use a nonnegative integer value that identifies the *n*th subexpression to be targeted. This is the subexpression parameter. In this example, 1 indicates the first subexpression. You can use a value from 0–9. A zero means that no subexpression is targeted. The default value for this parameter is 0.

Regular Expressions and Check Constraints: Examples

```
ALTER TABLE emp8
ADD CONSTRAINT email_addr
CHECK (REGEXP_LIKE(email,'@')) NOVALIDATE;
```

```
INSERT INTO emp8 VALUES
(500,'Christian','Patel','ChrisP2creme.com',
1234567890,'12-Jan-2004','HR_REP',2000,null,102,40);
```

```
Error starting at line 2 in command:
INSERT INTO emp8 VALUES
(500,'Christian','Patel',
'ChrisP2creme.com', 1234567890,
'12-Jan-2004', 'HR_REP', 2000, null, 102, 40)
Error report:
SQL Error: ORA-02290: check constraint (HR.EMAIL_ADDR) violated
02290. 00000 - "check constraint (%s.%s) violated"
*Cause: The values being inserted do not satisfy the named check
*Action: do not insert values that violate the constraint.
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Regular expressions can also be used in CHECK constraints. In this example, a CHECK constraint is added on the EMAIL column of the EMPLOYEES table. This ensures that only strings containing an "@" symbol are accepted. The constraint is tested. The CHECK constraint is violated because the email address does not contain the required symbol. The NOVALIDATE clause ensures that existing data is not checked.

For the example in the slide, the emp8 table is created by using the following code:

```
CREATE TABLE emp8 AS SELECT * FROM employees;
```

Note:

- Ignore the error message, if any, when you execute the CREATE TABLE emp8 statement.
- The example in the slide is executed by using the “Execute Statement” option in SQL Developer. The output format differs if you use the “Run Script” option.

Quiz

Which meta character treats the subsequent metacharacter in the expression as a literal?

- a. *
- b. +
- c. ?
- d. \



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Answer: d

Summary

In this lesson, you should have learned how to use regular expressions to search for, match, and replace strings.



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In this lesson you have learned to use the regular expressions support features in Oracle Database. Regular expression support is available in both SQL and PL/SQL.

Practice 4 Overview

This practice covers using regular expressions functions to do the following:

- Searching for, replacing, and manipulating data
- Creating a new CONTACTS table and adding a CHECK constraint to the p_number column to ensure that phone numbers are entered into the database in a specific standard format
- Testing the adding of some phone numbers into the p_number column by using various formats



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In this practice, you use regular expressions functions to search for, replace, and manipulate data. You also create a new CONTACTS table and add a CHECK constraint to the p_number column to ensure that phone numbers are entered into the database in a specific standard format.

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

Analyzing and Reporting Data by Using SQL



ORACLE®

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Objectives

After completing this lesson, you should be able to:

- List the SQL for analysis and reporting functions
- Use some of the ranking and percentile SQL analytic functions
- Compute ratio by using the `RATIO_TO_REPORT` function
- Perform analytical operations by using the `LAG/LEAD` and `NTH_VALUE` functions
- Perform aggregate operations by using the `LISTAGG` function
- Limit the rows returned by a query



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Lesson Agenda

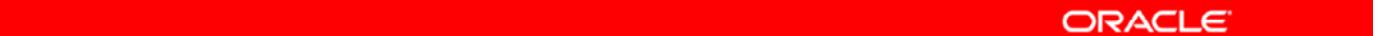
- Overview of SQL analytic functions
- Using the RANK, PERCENT_RANK, CUME_DIST, and DENSE_RANK functions
- Using the RATIO_TO_REPORT reporting function
- Using the LAG/LEAD and NTH_VALUE functions
- Performing aggregate operations by using the LISTAGG function
- Using the row_limiting_clause to limit the rows returned in the result set



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

SQL for Analysis and Reporting: Overview

- Oracle has enhanced SQL's analytical processing capabilities by including a family of analytic SQL functions.
- These analytic functions enable you to calculate and perform:
 - Rankings and percentiles
 - Pivoting operations
 - Moving window calculations
 - LAG/LEAD analysis
 - FIRST/LAST analysis
 - Linear regression statistics

The Oracle logo, which consists of the word "ORACLE" in white capital letters on a red rectangular background.

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Ranking functions include cumulative distributions, percent rank, and n-tiles. Moving window calculations allows you to find moving and cumulative aggregations, such as sums and averages. LAG/LEAD analysis enables direct inter-row references so you can calculate period-to-period changes. FIRST/LAST analysis enables you to find the first or last value in an ordered group.

Other enhancements to SQL include the CASE expression and partitioned outer join. CASE expressions provide if-then logic that is useful in many situations. Partitioned outer join is an extension to the American National Standards Institute (ANSI) outer join syntax that allows users to selectively densify certain dimensions while keeping others sparse. This allows reporting tools to selectively densify dimensions—for example, the ones that appear in their cross-tabular reports while keeping others sparse.

To enhance performance, analytic functions can be parallelized: Multiple processes can simultaneously execute all these statements. These capabilities make calculations easier and more efficient, thereby enhancing database performance, scalability, and simplicity.

Using the Analytic Functions

Function type	Used for
Ranking	Calculating ranks, percentiles, and n-tiles of the values in a result set
Windowing	Calculating cumulative and moving aggregates; works with functions such as SUM, AVG, MIN, and so on
Reporting	Calculating shares such as market share; works with functions such as SUM, AVG, MIN, MAX, COUNT, VARIANCE, STDDEV, RATIO_TO_REPORT, and so on
LAG/LEAD	Finding a value in a row or a specified number of rows from a current row
FIRST/LAST	First or last value in an ordered group
Linear Regression	Calculating linear regression and other statistics



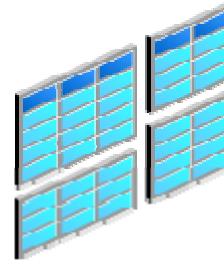
Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

To perform these operations, the analytic functions add several new elements to SQL processing. These elements build on existing SQL to allow flexible and powerful calculation expressions. With just a few exceptions, the analytic functions have these elements.

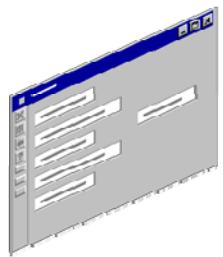
Concepts Used in Analytic Functions



Processing Order



Result Set Partitions



Window



Current Row

ORACLE®

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Processing Order

Query processing using analytic functions takes place in three stages.

- First, all joins, WHERE, GROUP BY, and HAVING clauses are performed.
- Second, the result set is made available to the analytic functions and all their calculations take place.
- Third, if the query has an ORDER BY clause at its end, the ORDER BY is processed to allow for precise output ordering.

Result Set Partitions

Analytic functions allow users to divide query result sets into groups of rows called “partitions.” Note that the term “partitions” used with analytic functions is unrelated to the table partitions feature. Throughout this lesson, this term refers only to analytic functions. Partitions are created after the groups are defined with GROUP BY clauses, so they are available to any aggregate results such as sums and averages. Partition divisions may be based upon any desired columns or expressions. A query result set may be partitioned into just one partition holding all the rows, a few large partitions, or many small partitions holding just a few rows each.

Window

For each row in a partition, you can define a sliding window of data. This window determines the range of rows used to perform the calculations for the current row. Window sizes can be based on a physical number of rows or a logical interval such as time. The window has a starting row and an ending row. Depending on its definition, the window may move at one or both ends. For instance, a window defined for a cumulative sum function would have its starting row fixed at the first row of its partition, and its ending row would slide from the starting point all the way to the last row of the partition. In contrast, a window defined for a moving average would have both its starting and end points slide so that they maintain a constant physical or logical range.

A window can be set as large as all the rows in a partition, or as a sliding window of one row within a partition. When a window is near a border, the function returns results for only the available rows, rather than warning you that the results are not what you want.

When you use window functions, the current row is included during calculations, so you should specify $(n-1)$ only when you are dealing with n items.

Current Row

Each calculation performed with an analytic function is based on a current row within a partition. The current row serves as the reference point determining the start and end of the window. For instance, a centered moving average calculation could be defined with a window that holds the current row, the six preceding rows, and the following six rows. This would create a sliding window of 13 rows.

Lesson Agenda

- Overview of SQL analytic functions
- Using the RANK, PERCENT_RANK, CUME_DIST, and DENSE_RANK functions
- Using the RATIO_TO_REPORT reporting function
- Using the LAG/LEAD and NTH_VALUE functions
- Performing aggregate operations by using the LISTAGG function
- Using the row_limiting_clause to limit the rows returned in the result set

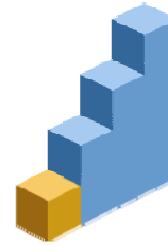


Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Using the Ranking Functions

A ranking function computes the rank of a record compared to other records in the data set based on the values of a set of measures. The types of ranking function are:

- RANK and DENSE_RANK functions
- CUME_DIST function
- PERCENT_RANK function
- NTILE function
- ROW_NUMBER function



ORACLE®

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Working with the RANK Function

```
RANK ( ) OVER ( [query_partition_clause] order_by_clause )
```

- The RANK function calculates the rank of a value in a group of values, which is useful for top-N and bottom-N reporting.
- When using the RANK function, ascending is the default sort order, which you can change to descending.
- Rows with equal values for the ranking criteria receive the same rank.



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The expressions in the optional PARTITION BY clause divide the query result set into groups within which the RANK function operates. That is, RANK gets reset whenever the group changes. In effect, the value expressions of the PARTITION BY clause define the reset boundaries. If the PARTITION BY clause is missing, ranks are computed over the entire query result set. The ORDER BY clause specifies the measures (<value expression>) based on which ranking is done, and defines the order in which rows are sorted in each group (or partition). After the data is sorted within each partition, ranks are given to each row starting from 1. The NULLS FIRST | NULLS LAST clause indicates the position of NULLs in the ordered sequence, which is first or last in the sequence. The order of the sequence would make NULLs compare high or low with respect to non-NULL values. If the sequence was in ascending order, NULLS FIRST implies that NULLs are smaller than all other non-NULL values and NULLS LAST implies they are larger than non-NULL values. It is the opposite for descending order.

If the NULLS FIRST | NULLS LAST clause is omitted, the ordering of the NULL values depends on the ASC or DESC arguments. NULLs are considered larger than other values. If the ordering sequence is ASC, NULLs appear last; otherwise, they appear first. NULLs are considered equal to other NULLs and, therefore, the order in which NULLs are presented is nondeterministic.

Using the RANK Function: Example

```
SELECT department_id, last_name, salary,  
       RANK() OVER (PARTITION BY department_id  
                     ORDER BY salary DESC) "Rank"  
FROM employees  
WHERE department_id = 60  
ORDER BY department_id, "Rank", salary;
```

DEPARTMENT_ID	LAST_NAME	SALARY	Rank
60	Hunold	9000	1
60	Ernst	6000	2
60	Austin	4800	3
60	Pataballa	4800	3
60	Lorentz	4200	5



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The query in the slide example ranks the employees in the sample HR schema in department 60 based on their salary. Note how identical salary values such as 4800 receive the same rank and cause nonconsecutive ranks. Compare this example with the example for DENSE_RANK.

Note: The slide example uses the HR schema. Most of the code examples in this lesson use the SH schema unless otherwise indicated.

Per-Group Ranking

- The RANK function can be made to operate within groups—that is, the rank gets reset whenever the group changes.
- The group expressions in the PARTITION BY subclause divide the data set into groups within which RANK operates.



ORACLE®

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The RANK function can be made to operate within groups by using the PARTITION BY clause. For example, to rank products within each channel by their dollar sales, you could issue a statement similar to the one in the next slide.

Per-Group Ranking: Example

```
SELECT channel_desc, calendar_month_desc, TO_CHAR(SUM(amount_sold),
      '9,999,999,999') SALES$, RANK() OVER (PARTITION BY channel_desc
      ORDER BY SUM(amount_sold) DESC) AS RANK_BY_CHANNEL
FROM sales, products, customers, times, channels
WHERE sales.prod_id = products.prod_id
      AND sales.cust_id = customers.cust_id
      AND sales.time_id = times.time_id
      AND sales.channel_id = channels.channel_id
      AND times.calendar_month_desc IN ('2000-08', '2000-09', '2000-
      10', '2000-11')
      AND channels.channel_desc IN ('Direct Sales', 'Internet')
GROUP BY channel_desc, calendar_month_desc;
```

CHANNEL_DESC	CALENDAR_MONTH_DESC	SALES\$	RANK_BY_CHANNEL
Direct Sales	2000-08	1,236,104	1
Direct Sales	2000-10	1,225,584	2
Direct Sales	2000-09	1,217,808	3
Direct Sales	2000-11	1,115,239	4
Internet	2000-11	284,742	1
Internet	2000-10	239,236	2
Internet	2000-09	228,241	3
Internet	2000-08	215,107	4



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

A single query block can contain more than one ranking function, each partitioning the data into different groups (that is, reset on different boundaries). The groups can be mutually exclusive. The following queries rank products based on their dollar sales:

Within each month (`rank_of_product_per_region`)

Within each channel (`rank_of_product_total`)

RANK and DENSE_RANK Functions: Example

```
SELECT last_name, salary,
       RANK() OVER (PARTITION BY
                     department_id
                     ORDER BY salary DESC) "Rank"
  FROM employees
 WHERE department_id = 60
 ORDER BY salary DESC, "Rank" DESC;
```

	LAST_NAME	SALARY	Rank
1	Hunold	9000	1
2	Ernst	6000	2
3	Austin	4800	3
4	Pataballa	4800	3
5	Lorentz	4200	5

DENSE_RANK function leaves no gaps in ranking sequence when there are ties.

	LAST_NAME	SALARY	Drank
1	Hunold	9000	1
2	Ernst	6000	2
3	Austin	4800	3
4	Pataballa	4800	3
5	Lorentz	4200	4

```
SELECT last_name, salary,
       DENSE_RANK() over (PARTITION BY
                           department_id
                           ORDER BY salary DESC) "Drank"
  FROM employees
 WHERE department_id = 60
 ORDER BY salary DESC, "Drank" DESC;
```

RANK function leaves gaps in ranking sequence when there are ties.

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The difference between RANK and DENSE_RANK is that DENSE_RANK leaves no gaps in ranking sequence when there are ties. That is, if you were ranking a competition using DENSE_RANK and had three people tie for the second place, you would say that all three were in the second place and that the next person came in third. The RANK function would also give three people in the second place, but the next person would be in the fifth place.

Per-Cube and Rollup Group Ranking

```
SELECT channel_desc, country_iso_code,
       TO_CHAR(SUM(amount_sold), '9,999,999,999') SALES$,
       RANK() OVER (PARTITION BY GROUPING_ID(channel_desc, country_iso_code)
                     ORDER BY SUM(amount_sold) DESC) AS RANK_PER_GROUP
  FROM sales, customers, times, channels, countries
 WHERE sales.time_id = times.time_id AND
       sales.cust_id=customers.cust_id AND
       sales.channel_id = channels.channel_id AND
       channels.channel_desc IN ('Direct Sales', 'Internet') AND
       times.calendar_month_desc='2000-09' AND
       country_iso_code IN ('GB', 'US', 'JP')
 GROUP BY CUBE(channel_desc, country_iso_code);
```

CHANNEL_DESC	COUNTRY_ISO_CODE	SALES\$	RANK_PER_GROUP
Direct Sales	JP	1,217,808	1
Direct Sales	GB	1,217,808	1
Direct Sales	US	1,217,808	1
Internet	GB	228,241	4
Internet	US	228,241	4
Internet	JP	228,241	4
Direct Sales		3,653,423	1
Internet		684,724	2
	GB	1,446,049	1
	JP	1,446,049	1
	US	1,446,049	1
		4,338,147	1

12 rows selected



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Analytic functions, for example RANK, can be reset based on the groupings provided by a CUBE, ROLLUP, or GROUPING SETS operator. It is useful to assign ranks to the groups created by the CUBE, ROLLUP, and GROUPING SETS queries.

Using CUME_DIST and PERCENT_RANK Functions

- The CUME_DIST function computes the position of a specified value relative to a set of values.

```
CUME_DIST () OVER ([query_partition_clause]  
order_by_clause)
```

- The PERCENT_RANK function uses rank values in its numerator and returns the percent rank of a value relative to a group of values.

```
(rank of row in its partition - 1) / (number of rows in  
the partition - 1)
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

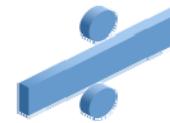
PERCENT_RANK is similar to the CUME_DIST (cumulative distribution) function, including the syntax. The range of values returned by PERCENT_RANK and CUME_DIST functions is 0 to 1 (both inclusive). The first row in any set has a value of 0. The return value is NUMBER.

As an analytic function, for a row r , the PERCENT_RANK function calculates the rank of $r-1$, divided by 1 less than the number of rows being evaluated.

Using the CUME_DIST Function: Example

```
SELECT job_id, last_name, salary,  
       CUME_DIST()  
    OVER (PARTITION BY job_id ORDER BY salary) AS CD  
FROM employees  
WHERE job_id LIKE 'PU%';
```

	JOB_ID	LAST_NAME	SALARY	CD
1	PU_CLERK	Colmenares	2500	0.2
2	PU_CLERK	Himuro	2600	0.4
3	PU_CLERK	Tobias	2800	0.6
4	PU_CLERK	Baida	2900	0.8
5	PU_CLERK	Khoo	3100	1
6	PU_MAN	Raphaely	11000	1



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

As an analytic function, CUME_DIST computes the relative position of a specified value in a group of values. For a row R, assuming ascending ordering, the CUME_DIST of R is the number of rows with values lower than or equal to the value of R, divided by the number of rows being evaluated (the entire query result set or a partition). The example in the slide calculates the salary percentile for each employee in the purchasing division.

- For row 1, there is only 1 row with job_id as PU_CLERK and with salary less than or equal to 2500 among the five rows fetched from the result set. Therefore CUME_DIST = 1/5 or 0.2.
- For row 2, there are 2 rows with job_id as PU_CLERK and with salary less than or equal to 2600 among the five rows fetched from the result set. Therefore CUME_DIST = 2/5 or 0.4.
- ...
- For row 6, there is only 1 row with job_id as PU_MAN and with salary less than or equal to 11000 in the result set. Therefore CUME_DIST = 1/1 or 1.

Note: Execute the examples in the slide by using the HR schema.

Using the PERCENT_RANK Function: Example

```
SELECT job_id, last_name, salary,
       RANK ()
    OVER (PARTITION BY job_id ORDER BY salary ) AS rank,
       PERCENT_RANK()
    OVER (PARTITION BY job_id ORDER BY salary ) AS pr
  FROM employees
 WHERE job_id LIKE 'PU%';
```

JOB_ID	LAST_NAME	SALARY	RANK	PR
1 PU_CLERK	Colmenares	2500	1	0
2 PU_CLERK	Himuro	2600	2	0.25
3 PU_CLERK	Tobias	2800	3	0.5
4 PU_CLERK	Baida	2900	4	0.75
5 PU_CLERK	Khoo	3100	5	1
6 PU_MAN	Raphaely	11000	1	0



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The slide example calculates the percent rank of each employee in the purchasing division, partitioning by the `job_id` column. For a row r , `PERCENT_RANK` function calculates the rank of $r-1$, divided by 1 less than the number of rows being evaluated.

For row 1, the `PERCENT_RANK` = $(1-1)/(5-1)$ OR 0/4 OR 0

For row 2, the `PERCENT_RANK` = $(2-1)/(5-1)$ OR 1/4 OR 0.25

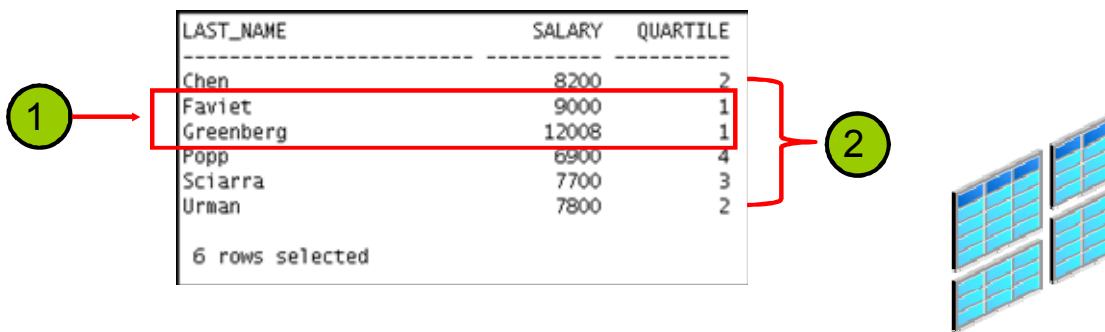
...

For row 6, there is only 1 row with `job_id` as `PU_MAN`. Therefore `PERCENT_RANK` = $(1-1)/(1-1)$ OR 0.

Using the NTILE Function: Example

```
NTILE (expr) OVER ([query_partition_clause] order_by_clause)
```

```
SELECT last_name, salary, NTILE(4)
      OVER (ORDER BY salary DESC) AS quartile
  FROM employees
 WHERE department_id = 100
 ORDER BY last_name, salary, quartile;
```



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The NTILE function is used for calculation of quartiles, deciles, and other common summary statistics. This function divides an ordered partition into a specified number of groups called buckets and assigns a bucket number to each row in the partition. This function also allows users to divide a data set into fourths, thirds, and other groupings. If the number of rows in the partition does not divide evenly into the number of buckets, then the number of rows assigned for each bucket will differ by one at most. The extra rows will be distributed one for each bucket starting from the lowest bucket number. For instance, if there are 62 rows in a partition that has an NTILE(3) function, the first 21 rows will be in the first bucket, the next 21 in the second bucket, and the final 20 rows in the third bucket.

The example in the slide divides the values in the salary column of department ID 100 into 4 buckets. The salary column has 6 values in this department, so the two extra values (the remainder of $6 / 4$) are allocated to buckets 1 and 2. Therefore, buckets 1 and 2 have more value than buckets 3 or 4.

Lesson Agenda

- Overview of SQL analytic functions
- Using the RANK, PERCENT_RANK, CUME_DIST, and DENSE_RANK functions
- Using the RATIO_TO_REPORT reporting function
- Using the LAG/LEAD and NTH_VALUE functions
- Performing aggregate operations by using the LISTAGG function
- Using the row_limiting_clause to limit the rows returned in the result set



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Using the RATIO_TO_REPORT Function

- The RATIO_TO_REPORT function computes the ratio of a value to the sum of a set of values.

```
RATIO_TO_REPORT ( expr ) OVER ( [query_partition_clause] )
```

- The `expr` argument is a valid expression involving column references or aggregates.
- The `PARTITION BY` clause defines the groups on which the function computes.

The Oracle logo, featuring the word "ORACLE" in a white sans-serif font inside a red horizontal bar.

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The RATIO_TO_REPORT is a reporting function. This function computes the ratio of a value to the sum of a set of values. In the syntax, `expr` is an expression that involves column references or aggregates. If `expr` evaluates to `NULL`, then the ratio-to-report value also evaluates to `NULL`. The `query_partition_clause` determines the set of values. If you omit this clause, then the ratio-to-report is computed over all rows returned by the query.

Note: You cannot nest analytic functions by using RATIO_TO_REPORT or any other analytic function for `expr`. However, you can use other built-in function expressions.

Using the RATIO_TO_REPORT Function: Example

```
SELECT last_name, salary, RATIO_TO_REPORT(salary)
      OVER () AS rr
  FROM employees
 WHERE job_id = 'PU_CLERK'
 ORDER BY last_name, salary, rr;
```

LAST_NAME	SALARY	RR
Baida	2900	0.2086330935
Colmenares	2500	0.1798561151
Himuro	2600	0.1870503597
Khoo	3100	0.2230215827
Tobias	2800	0.2014388489



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The slide example calculates the ratio-to-report value of each purchasing clerk's salary to the total of all purchasing clerks' salaries.

The following example illustrates the use of the RATIO_TO_REPORT function with the PARTITION BY clause:

```
SELECT department_id, last_name, salary,
       RATIO_TO_REPORT(salary)
      OVER (PARTITION BY department_id) AS ratio
  FROM employees
 ORDER BY department_id, salary DESC;
```

	DEPARTMENT_ID	LAST_NAME	SALARY	RATIO
1	10	Whalen	4400	1
2	20	Hartstein	13000	0.6842105263157894736842105263157894736842
3	20	Fay	6000	0.3157894736842105263157894736842105263158
4	30	Raphaely	11000	0.4417670682730923694779116465863453815261
	...			

Lesson Agenda

- Overview of SQL analytic functions
- Using the RANK, PERCENT_RANK, CUME_DIST, and DENSE_RANK functions
- Using the RATIO_TO_REPORT reporting function
- Using the LAG/LEAD and NTH_VALUE functions
- Performing aggregate operations by using the LISTAGG function
- Using the row_limiting_clause to limit the rows returned in the result set



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Using the LAG and LEAD Analytic Functions

```
{LAG | LEAD}(value_expr [, offset] [, default])
OVER ([query_partition_clause] order_by_clause)
```

- The LAG and LEAD functions are used to compare values when the relative positions of rows are known.
- They work by specifying the count of rows that separate the target row from the current row.



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The LAG and LEAD functions are used to compare values when the relative positions of rows are known. These functions work by specifying the count of rows that separate the target row from the current row. The LAG function provides access to a row at a given offset before the current position, and the LEAD function provides access to a row at a given offset after the current position. These functions enhance processing speed as they provide access to more than one row of a table at the same time without a self-join.

In the syntax:

offset	is an optional parameter and defaults to 1
default	is an optional parameter and is the value returned if offset falls outside the bounds of the table or partition
IGNORE NULLS	ignores rows with NULL values

Note: You cannot nest analytic functions using LAG or any other analytic function for value_expr. However, you can use other built-in function expressions for value_expr.

Using the LAG and LEAD Analytic Functions: Example

```

SELECT time_id, TO_CHAR(SUM(amount_sold), '9,999,999') AS
      SALES,
      TO_CHAR(LAG(SUM(amount_sold), 1) OVER (ORDER BY
          time_id), '9,999,999') AS LAG1,
      TO_CHAR(LEAD(SUM(amount_sold), 1) OVER (ORDER BY
          time_id), '9,999,999') AS LEAD1
FROM sales
WHERE time_id >= TO_DATE('10-OCT-2000') AND
      time_id <= TO_DATE('14-OCT-2000')
GROUP BY time_id;

```

TIME_ID	SALES	LAG1	LEAD1
10-OCT-00	238,479	23,183	
11-OCT-00	23,183	238,479	24,616
12-OCT-00	24,616	23,183	76,516
13-OCT-00	76,516	24,616	29,795
14-OCT-00	29,795	76,516	



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The example in the slide uses both the LAG and LEAD functions using the SH schema.

In the example, the offset is specified as 1 and there is no specified default value. This means that if the offset goes beyond the scope of the window, the default is NULL.

The following is an example of using the LAG and LEAD functions on more than one table:

```

SELECT t.calendar_month_desc,
       MAX(t.calendar_month_name) MONTH_NAME ,SUM(s.quantity_sold) AS
           quantity_sold,
       LAG(SUM(s.quantity_sold),12) OVER (ORDER BY t.calendar_month_desc )
           AS year_before,
       LEAD(SUM(s.quantity_sold),12) OVER (ORDER BY t.calendar_month_desc )
           AS year_after
  FROM sh.sales s, sh.times t
 WHERE s.time_id=t.time_id
 GROUP BY t.calendar_year, t.calendar_month_desc ;

```

Using the FIRST_VALUE and LAST_VALUE Functions

```
{FIRST_VALUE | LAST_VALUE} ( <expr> )
[RESPECT NULLS | IGNORE NULLS] OVER (analytic clause);
```

- Given a series of rows, the FIRST_VALUE and LAST_VALUE functions select the first and last rows returned from the query.
- If you specify the IGNORE NULLS option, then:
 - FIRST_VALUE returns the first non-null value in the set
 - LAST_VALUE returns the last non-null value in the set



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The FIRST_VALUE and LAST_VALUE functions return the first and last values in an ordered set of values. If the first value or last value in the set is NULL, then the function returns NULL unless you specify the IGNORE NULLS option in the query. This setting is useful for data densification. If all values in the set are NULL, then both the functions return a NULL value irrespective of using the IGNORE NULLS option.

To select the name of the employee with the lowest salary for each employee in department 60, issue the following command:

```
SELECT department_id, last_name, salary,
TO_CHAR(FIRST_VALUE(last_name) OVER (ORDER BY salary )) AS
FIRST_VALUE1
FROM (SELECT * FROM employees
WHERE department_id = 60
ORDER BY employee_id);
```

DEPARTMENT_ID	LAST_NAME	SALARY	FIRST_VALUE1
60	Lorentz	4200	Lorentz
60	Austin	4800	Lorentz
60	Pataballa	4800	Lorentz
60	Ernst	6000	Lorentz
60	Hunold	9000	Lorentz

Using the NTH_VALUE Function

```
{NTH_VALUE (<expr>, <n expr>) } [FROM FIRST | FROM LAST]
[RESPECT NULLS | IGNORE NULLS] OVER (<>window
specification>)
```

In the syntax:

- `expr` and `n` can be a column, constant, bind variable, or an expression.
- `FROM { FIRST | LAST }` determines whether the calculation begins at the first or last row of the window.
- `{ RESPECT | IGNORE } NULLS` determines whether null values are included in or eliminated from the calculation.



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The `NTH_VALUE` function enables you to find column values from an arbitrary row in the window. By default, this function considers null values in the expression and performs calculations from the first row of the window.

The `LAG` and `LEAD` functions are a simplification of the `NTH_VALUE` function. With `LAG` and `LEAD` functions, you can retrieve values from a row only at the specified physical offset. But with `NTH_VALUE` function, you can retrieve row-based values on what is called a logical offset or relative position.

Using the NTH_VALUE Function: Example

```
SELECT prod_id, channel_id, MIN(amount_sold),  
       NTH_VALUE(MIN(amount_sold), 2) OVER (PARTITION BY  
         prod_id ORDER BY channel_id) NV  
FROM sales  
WHERE prod_id BETWEEN 13 AND 16 GROUP BY prod_id,  
      channel_id;
```

PROD_ID	CHANNEL_ID	MIN(AMOUNT SOLD)	NV
13	2	907.34	
13	3	906.2	906.2
13	4	842.21	906.2
14	2	1015.94	
14	3	1036.72	1036.72
14	4	935.79	1036.72
15	2	871.19	
15	3	871.19	871.19
15	4	871.19	871.19
16	2	266.84	
16	3	266.84	266.84
16	4	266.84	266.84
16	9	11.99	266.84

13 rows selected



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The example in the slide returns the amount_sold value of the second channel_id in ascending order for each prod_id in the range between 13 and 16 in the SH schema.

Lesson Agenda

- Overview of SQL analytic functions
- Using the RANK, PERCENT_RANK, CUME_DIST, and DENSE_RANK functions
- Using the RATIO_TO_REPORT reporting function
- Using the LAG/LEAD and NTH_VALUE functions
- Performing aggregate operations by using the LISTAGG function
- Using the row_limiting_clause to limit the rows returned in the result set



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Using the LISTAGG Function

```
LISTAGG (<measure_expr> [, <delimiter>) WITHIN GROUP  
(ORDER BY <oby_expression_list>))
```

- The LISTAGG function orders data within each group based on the ORDER BY clause and concatenates the values within the measure column
- The delimiter_expr designates the string to separate the measure values
- The order_by_clause determines the order in which the concatenated values are returned



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The LISTAGG function orders data within each group based on the ORDER BY clause and then concatenates the values of the measure column.

- As a single-set aggregate function, the LISTAGG function operates on all rows and returns a single output row.
- As a group-set aggregate, the function operates on and returns an output row for each group defined by the GROUP BY clause.
- As an analytic function, LISTAGG partitions the query result set into groups based on one or more expressions in the query_partition_clause.

Note: The measure_expr argument can be any expression, and null values in this column are ignored.

Using the LISTAGG Function: Example

- LISTAGG as a single-set aggregate function:

```
SELECT LISTAGG(last_name, ', ')  
      WITHIN GROUP (ORDER BY hire_date, last_name)  
      "Emp_list", MIN(hire_date) "Earliest"  
FROM employees  
WHERE department_id = 30;
```

- LISTAGG as a group-set aggregate function:

```
SELECT department_id "Dept",  
      LISTAGG(last_name, ', ')  
      WITHIN GROUP (ORDER BY hire_date) "Employees"  
FROM employees  
WHERE department_id IN(10, 20, 30)  
GROUP BY department_id;
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The first example in the slide lists all of the employees in the department 30 ordered by the hire date and last name columns. The results of the example are as follows:

Emp_list

Raphaely; Khoo; Tobias; Baida; Himuro; Colmenares

The second example in the slide lists all the employees in departments 10, 20, and 30 in the order of their hire date. The results of the example are as follows:

Dept. Employees

10 Whalen
20 Hartstein; Fay
30 Raphaely; Khoo; Tobias; Baida; Himuro; Colmenares

Lesson Agenda

- Overview of SQL analytic functions
- Using the RANK, PERCENT_RANK, CUME_DIST, and DENSE_RANK functions
- Using the RATIO_TO_REPORT reporting function
- Using the LAG/LEAD and NTH_VALUE functions
- Performing aggregate operations by using the LISTAGG function
- Using the row_limiting_clause to limit the rows returned in the result set



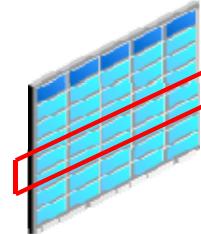
Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

SQL Row-Limiting Clause

- Enables you to limit the rows that are returned by the query

```
...
[OFFSET offset { ROW | ROWS }]
[FETCH { FIRST | NEXT } [{ row_count | percent
PERCENT }] { ROW | ROWS }
{ ONLY | WITH TIES }]
```

- Abides to the ANSI SQL international standard for enhanced compatibility and easier migration



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

A query that first sorts the rows and then limits the number of rows returned in the query is often called a Top-N query. The SQL SELECT syntax provides this functionality by using the `row_limiting_clause`. This clause limits the number of rows that are returned in the result set. It provides a straightforward way of creating reports in SQL.

In the syntax:

- `OFFSET` specifies the number of rows to skip before row limiting begins. This value must be a number.
- `FETCH_FIRST` specifies the number of rows or percentage of rows to return.
- `Row_count | percent PERCENT` specifies the number of rows or the percentage of the total number of selected rows to return.
- `WITH TIES` includes rows with the same ordering keys as the last row of the row-limited result set.

Note: The value for `offset` and `percent` must be a number.

SQL Row-Limiting Clause: Example

```
SELECT employee_id, first_name  
FROM employees  
ORDER BY employee_id  
FETCH FIRST 5 ROWS ONLY;
```

EMPLOYEE_ID	FIRST_NAME
100	Steven
101	Neena
102	Lex
103	Alexander
104	Bruce

```
SELECT employee_id, first_name  
FROM employees  
ORDER BY employee_id  
OFFSET 5 ROWS FETCH NEXT 5 ROWS ONLY;
```

EMPLOYEE_ID	FIRST_NAME
105	David
106	Valli
107	Diana
108	Nancy
109	Daniel

Returns the 5 employees with the next set of lowest employee_id

Returns the 5 employees with the lowest employee_id

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The first example in the slide returns the first set of five employees with the lowest employee_id. The second example in the slide returns the next set of five employees with the lowest employee_id.

Quiz

What is the difference between the RANK and DENSE_RANK functions?



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Answer: The difference between the RANK and DENSE_RANK function is that the DENSE_RANK function leaves no gaps in ranking sequence when there are ties.

Summary

In this lesson, you should have learned how to:

- List the SQL for analysis and reporting functions
- Use some of the ranking and percentile SQL analytic functions
- Compute ratio by using the `RATIO_TO_REPORT` function
- Perform analytical operations by using the `LAG/LEAD` and `NTH_VALUE` functions
- Perform aggregate operations by using the `LISTAGG` function
- Limit the rows returned by a query



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Practice 5 Overview

This practice covers the following topics:

- Using the `RANK` function to rank the employees in the sample `HR` schema in department 80 based on their salary and commission
- Using the `LISTAGG` function to list the employees in department 30 based on their salary and commission
- Using the `RATIO_TO_REPORT` function to calculate the ratio of the quantity of products sold for `product_id` 1797
- Using the `row_limiting_clause` to display the first five ranks and the next five ranks from a result set



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In this practice, you use the `RANK` function, the `LISTAGG` function, the `RATIO_TO_REPORT` function, and the `ROW_LIMITING_CLAUSE` clause. If you encounter any errors, you can use the Compiler-Log tab in SQL Developer.

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

Performing Pivoting and Unpivoting Operations

ORACLE®

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Objectives

After completing this lesson, you should be able to:

- Identify the benefits of pivoting and unpivoting operations
- Write cross-tabulation queries to pivot (rotate) column values into new columns and to unpivot (rotate) columns into column values
- Pivot and unpivot with multiple columns and multiple aggregates
- Use wildcards and aliases with pivoting operations



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Benefits of Using Pivoting Operations

- Data returned by business intelligence (BI) queries is more useful if presented in a cross-tabular format.
- Pivoting enables you to transform multiple rows of input into fewer rows, generally with more columns.
- Performing pivots on the server side can:
 - Enhance processing speed
 - Reduce network load



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Pivoting is a key technique in BI queries where you transform multiple rows of input into fewer rows, generally with more columns. When pivoting, an aggregation operator is applied, enabling the query to condense large data sets into smaller and more readable results.

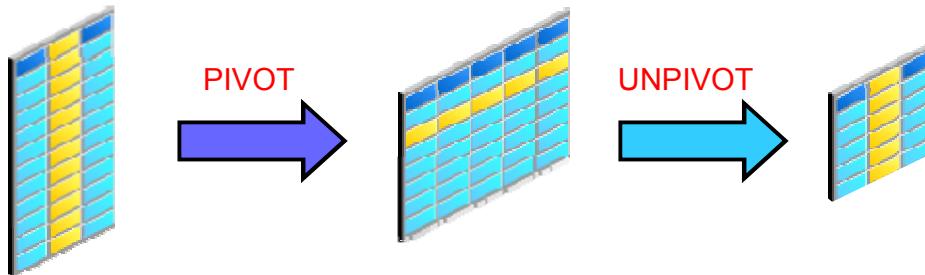
By performing pivots on the server side:

- Processing burden is removed from client applications, simplifying client-side development, and potentially enhancing processing speed
- Network load is reduced because only aggregated pivot results need to traverse the network and not the detail data

Data that was originally on multiple rows can be transformed into a single row of output, enabling intrarow calculations without a SQL join operation.

PIVOT and UNPIVOT Clauses of the SELECT Statement

- You can use the PIVOT operator to write cross-tabulation queries that rotate the column values into new columns, aggregating data in the process.
- You can use the UNPIVOT operator to rotate columns into values of a column.



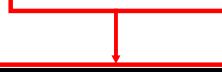
ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

An UNPIVOT does not reverse a PIVOT operation; instead, it rotates data from columns into rows.

Pivoting on the QUARTER Column: Conceptual Example

PRODUCT	COUNTRY	CHANNEL	QUARTER	AMOUNT_SOLD	QUANTITY_SOLD
Shorts	Germany	C	Q2	20,000	1,000
Shorts	Poland	P	Q2	40,000	2,500
Shorts	USA	S	Q3	30,000	2,000
Kids Jeans	Japan	C	Q2	60,000	2,000
Kids Jeans	USA	I	Q1	40,000	2,500
Kids Jeans	USA	I	Q4	30,000	1,500



PRODUCT	Q1	Q2	Q3	Q4
Shorts		3,500	2,000	
Kids Jeans	2,500	2,000		1,500

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The first slide example displays the columns of a table. The table in the second example displays the results of pivoting on the QUARTER column. The values of the QUARTER column, Q1, Q2, Q3, and Q4 are rotated into new columns. The quantity sold is grouped by products and quarters. To accomplish the desired result, you can use the PIVOT clause to pivot the QUARTER column, that is, turn the values of this column into separate columns and aggregate data by using a group function such as SUM on the QUANTITY_SOLD along the way for each PRODUCT.

The quantity sold is calculated for each quarter for *all* products, *all* channels, and *all* countries. For example, the total number of shorts sold for all channels and all countries for quarter 2 is 3500. The total number Kids Jeans sold for all channels and all countries for quarter 2 is 2000.

Note that the first example contains six rows before the pivoting operation. In the second example and after pivoting the QUARTER column, only two rows are displayed. Pivoting transforms multiple rows of input into fewer and generally wider rows.

PIVOT Clause Syntax

```
table_reference PIVOT [ XML ]
  ( aggregate_function ( expr ) [[AS] alias]
    [, aggregate_function ( expr ) [[AS] alias] ]...
    pivot_for_clause
    pivot_in_clause )
```

```
-- Specify the column(s) to pivot whose values are to
-- be pivoted into columns.
pivot_for_clause =
FOR { column |( column [, column]... ) }
```

```
-- Specify the pivot column values from the columns you
-- specified in the pivot_for_clause.
pivot_in_clause =
IN ( { { { expr | ( expr [, expr]... ) } [ [ AS] alias] }...
  | subquery | { ANY | ANY [, ANY]... } }
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You can use the `PIVOT` clause to write cross-tabulation queries that rotate rows into columns, aggregating the data in the process of rotation. The `XML` keyword is required when you use either a subquery or the wildcard `ANY` in `pivot_in_clause` to specify pivot values. You cannot specify `XML` when you specify explicit pivot values using expressions in `pivot_in_clause`. If the `XML` keyword is used, the output will include grouping columns and one column of `XMLType` rather than a series of pivoted columns.

Using the optional `AS` alias, you can specify an alias for each measure.

`aggregate_function` operates on the table's data, and the result of the computation appears in the cross-tabulation report. The `expr` argument for the aggregate function is the measure to be pivoted. It must be a column or expression of `query_table_expression` on which `pivot_clause` operates. Note that the aggregate function has an implicit `GROUP BY` based on the columns in the source data.

In `pivot_for_clause`, specify one or more columns whose values are to be pivoted into columns. In `pivot_in_clause`, specify the pivot column values from the column(s) you specified in `pivot_for_clause`.

For `expr`, specify a constant value of a pivot column. You can optionally provide an alias for each pivot column value. Use a subquery to extract the pivot column values by way of a nested subquery. If you specify `ANY`, all values of the pivot columns are pivoted into columns. Subqueries and wildcards are useful if you do not know the specific values in the pivot columns. However, you will need to do further processing to convert the XML output into a tabular format. The values evaluated by `pivot_in_clause` become the columns in the pivoted data.

Creating a New View: Example

```
CREATE OR REPLACE VIEW sales_view AS
SELECT
    prod_name AS product,
    country_name AS country,
    channel_id AS channel,
    SUBSTR(calendar_quarter_desc, 6,2) AS quarter,
    SUM(amount_sold) AS amount_sold,
    SUM(quantity_sold) AS quantity_sold
FROM sales, times, customers, countries, products
WHERE sales.time_id = times.time_id AND
      sales.prod_id = products.prod_id AND
      sales.cust_id = customers.cust_id AND
      customers.country_id = countries.country_id
GROUP BY prod_name, country_name, channel_id,
SUBSTR(calendar_quarter_desc, 6, 2);
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In this lesson, you use the newly created view, `sales_view`. The definition of `sales_view` is displayed as follows:

```
DESCRIBE sales_view
Name          Null?    Type
-----        -----
PRODUCT       NOT NULL VARCHAR2(50)
COUNTRY       NOT NULL VARCHAR2(40)
CHANNEL       NOT NULL NUMBER
QUARTER      VARCHAR2(8)
AMOUNT_SOLD   NUMBER
QUANTITY_SOLD NUMBER
```

The calendar_quarter_desc column values in the SH schema are as follows:

```
SELECT DISTINCT calendar_quarter_desc  
FROM times;
```

```
CALENDAR_QUARTER_DESC
```

```
-----  
1999-02  
2000-04  
2002-03  
2000-03  
2001-04  
1998-01  
1999-04  
2002-02  
2002-04  
2000-02  
2001-01  
1998-02  
1998-04  
1999-01  
2001-02  
2001-03  
2002-01  
1999-03  
1998-03  
2000-01
```

20 rows selected.

Note: The two-character quarter value is extracted from the calendar_quarter_desc column starting at position 6 as shown in the example.

Selecting the SALES_VIEW Data

```
SELECT product, country, channel, quarter, quantity_sold  
FROM sales_view;
```

PRODUCT	COUNTRY	CHANNEL	QUARTER	QUANTITY SOLD
Y Box	Italy	4	01	21
Y Box	Italy	4	02	17
Y Box	Italy	4	03	20
. . .				
Y Box	Japan	2	01	35
Y Box	Japan	2	02	39
Y Box	Japan	2	03	36
Y Box	Japan	2	04	46
Y Box	Japan	3	01	65
. . .				
Bounce	Italy	2	01	34
Bounce	Italy	2	02	43
. . .				
9502 rows selected.				



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The two boxes around QUARTER and “9502 rows selected” highlight the two key changes performed by the PIVOT operator:

- The QUARTER column becomes multiple columns, each holding one quarter.
- The row count will drop to just 71 from 9502, representing the distinct products in the schema.

The following example displays the distinct channel_id and channel_desc column values from the CHANNELS table, and the distinct QUARTER column values from the QUARTER table in the SH schema.

```
SELECT DISTINCT channel_id, channel_desc
FROM channels
ORDER BY channel_id;
```

CHANNEL_ID	CHANNEL_DESC
2	Partners
3	Direct Sales
4	Internet
5	Catalog
9	Tele Sales

```
SELECT DISTINCT quarter
FROM sales_view;
```

QUARTER
04
01
02
03

Pivoting the QUARTER Column in the SH Schema: Example

```
SELECT *
FROM
  (SELECT product, quarter, quantity_sold
   FROM sales_view) PIVOT (sum(quantity_sold)
    FOR quarter IN ('01', '02', '03', '04'))
ORDER BY product DESC;
```

PRODUCT	'01'	'02'	'03'	'04'
Y Box	1455	1766	1716	1992
Xtend Memory	3146	4121	4122	
Unix/Windows 1-user	4259	3887	4601	4049
Standard Mouse	3376	1699	2654	2427
Smash up Boxing	1608	2127	1999	2110
...				
71 rows selected.				

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The slide example uses a subquery (inline view) in the `FROM` clause. This is needed because if you issue `SELECT *` directly from the sales view, the query output rows for each row of `sales_view` will be displayed.

The result set shows the product column followed by a column for each value of the quarter specified in the `IN` clause. The numbers shown in the pivoted output are the sum of `quantity_sold` for each product at each quarter. If you also specify an alias for each measure, then the column name is a concatenation of the pivot column value or alias, an underscore (`_`), and the measure alias. You can use the aliases in the slide example shown as follows. The results are shown on the next notes page.

```
SELECT *
FROM (SELECT product, quarter, quantity_sold
      FROM sales_view) PIVOT (sum(quantity_sold)
       FOR quarter in ('01' AS Q1, '02' AS Q2, '03' AS Q3,
       '04' AS Q4))
ORDER by PRODUCT DESC;
```

The results of the code example on the previous notes page are as follows:

PRODUCT	Q1	Q2	Q3	Q4
Y Box	1455	1766	1716	1992
Xtend Memory	3146	4121	4122	
Unix/Windows	4259	3887	4601	4049
Standard Mouse	3376	1699	2654	2427
...				
71 rows selected.				

Before Oracle Database 11g, this example could be accomplished by using the following syntax:

```
SELECT product,
       SUM(CASE when quarter = '01' THEN quantity_sold ELSE NULL END) Q1,
       SUM(CASE when quarter = '02' THEN quantity_sold ELSE NULL END) Q2,
       SUM(CASE when quarter = '03' THEN quantity_sold ELSE NULL END) Q3,
       SUM(CASE when quarter = '04' THEN quantity_sold ELSE NULL END) Q4
  FROM (
    SELECT product, quarter, quantity_sold
      FROM SALES_VIEW )
 GROUP BY product;
```

The advantage of the new syntax over the syntax used before Oracle Database 11g is that it enables greater query optimization by Oracle. The query optimizer recognizes the `PIVOT` keyword and, as a result, uses algorithms optimized to process it efficiently.

Pivoting on Multiple Columns

- To pivot on more than one column:
 - A pivoting column must be a column of the table reference on which the pivot is operating.
 - The pivoting column cannot be an arbitrary expression.
- An aggregation operator is applied for each item in the pivot column value list.



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Pivoting on Multiple Columns

```
SELECT *
FROM
  (SELECT product, channel, quarter, quantity_sold
   FROM sales_view) PIVOT (sum(quantity_sold) FOR (channel,
quarter) IN ((3, '01') AS Direct_Sales_Q1,
              (4, '01') AS Internet_Sales_Q1))
ORDER BY product DESC;
```

PRODUCT	DIRECT_SALES_Q1	INTERNET_SALES_Q1
Y Box	771	253
Xtend Memory	1935	350
Unix/Windows 1-user pack	2544	397
Standard Mouse	2326	256
Smash up Boxing	1114	129
...		
71 rows selected.		



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The example in the slide pivots on both the CHANNEL and QUARTER columns. The example uses only the CHANNEL values 3 (Direct Sales) and 4 (Internet) and only the Q1 value for the QUARTER column. The following example specifies more values for the QUARTER column:

```
SELECT *
FROM
  (SELECT product, channel, quarter, quantity_sold
   FROM sales_view
  ) PIVOT (sum(quantity_sold) FOR (channel, quarter) IN
              ((3, '01') AS Direct_Sales_Q1,
               (3, '02') AS Direct_Sales_Q2,
               (3, '03') AS Direct_Sales_Q3,
               (3, '04') AS Direct_Sales_Q4,
               (4, '01') AS Internet_Sales_Q1,
               (4, '02') AS Internet_Sales_Q2,
               (4, '03') AS Internet_Sales_Q3,
               (4, '04') AS Internet_Sales_Q4))
ORDER BY product DESC;
```

Pivoting Using Multiple Aggregations

```
SELECT *
FROM
  (SELECT product, channel, amount_sold, quantity_sold
   FROM sales_view) PIVOT (SUM(amount_sold) AS sums,
                           SUM(quantity_sold) AS sumq
                          FOR channel IN (3 AS Dir_Sales, 4 AS Int_Sales))
ORDER BY product DESC;
```

PRODUCT	DIR_SALES_SUMS	DIR_SALES_SUMQ	INT_SALES_SUMS	INT_SALES_SUMQ
Y Box	1081050.96	3552	382767.45	1339
Xtend Memory	217011.38	8562	40553.93	1878
Unix/Windows	1999882.17	9313	376071.62	1872
Standard Mouse	153199.63	6140	28768.04	1195
Smash up Boxing	174592.24	5106	27858.84	904
...				
71 rows selected.				



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The query in the slide pivots SALES_VIEW on the CHANNEL column. The amount_sold and quantity_sold measures get pivoted.

Note that the query creates column headings by concatenating the pivot columns with the aliases of the aggregate functions, plus an underscore.

When you use multiple aggregation, you can omit the alias for only one aggregation. If you omit an alias, the corresponding result column name is the pivot value (or the alias for the pivot value).

Distinguishing PIVOT-Generated NULLS from NULLs in the Source Data

```
CREATE TABLE sales2
(prod_id      NUMBER,
 qtr         VARCHAR2(5),
 amount_sold NUMBER);
INSERT INTO sales2 VALUES(100, 'Q1', 10);
INSERT INTO sales2 VALUES(100, 'Q1', 20);
INSERT INTO sales2 VALUES(100, 'Q2', null);
INSERT INTO sales2 VALUES(200, 'Q1', 50);
```

```
SELECT * FROM sales2;
```

PROD_ID	QTR	AMOUNT_SOLD
100	Q1	10
100	Q1	20
100	Q2	
200	Q1	50



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You can distinguish between NULL values that are generated from the use of PIVOT and those that exist in the source data. The examples in this slide and the next illustrate NULLs that PIVOT generates.

The first slide example displays the definition of the `sales2` table structure and the commands used to insert the data into `sales2`.

Distinguishing PIVOT-Generated NULLS from NULLs in the Source Data

```
SELECT *
FROM
(SELECT prod_id, qtr, amount_sold
  FROM sales2) PIVOT (SUM(amount_sold),
                      COUNT(*) AS count_total
                     FOR qtr IN ('Q1', 'Q2') )
ORDER BY prod_id;
```

PROD_ID	Q1	Q1_COUNT_TOTAL	Q2	Q2_COUNT_TOTAL
100	30	2		1
200	50	1		0



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The query in the slide returns the `prod_id` rows and the resulting pivot columns: `Q1`, `Q1_COUNT_TOTAL`, and `Q2`, `Q2_COUNT_TOTAL`.

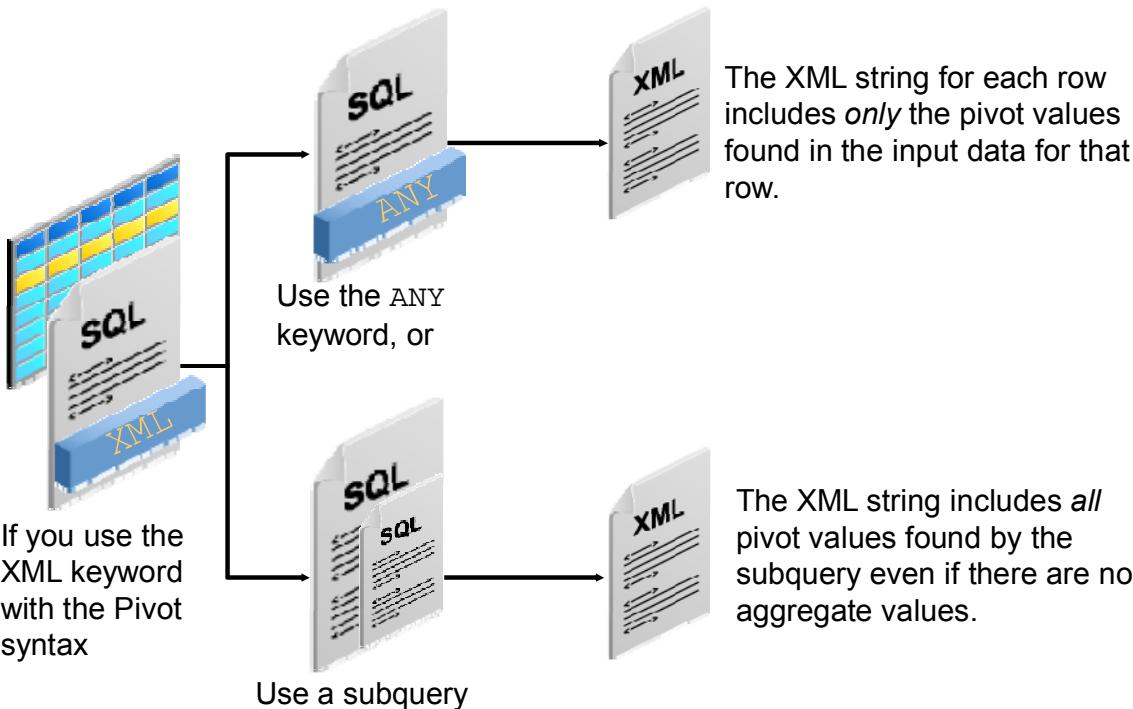
For each unique value of `prod_id`, `Q1_COUNT_TOTAL`, the query returns the total number of rows whose QTR value is `Q1`. The unique value of `Q2_COUNT_TOTAL` returns the total number of rows whose QTR value is `Q2`.

From the result set shown in the slide, there are two sales rows for `prod_id` 100 for the quarter `Q1`, and one sales row for `prod_id` 100 and the quarter `Q2`.

For `prod_id` 200, there is one sales row for the quarter `Q1` and no sales row for the quarter `Q2`.

Using `Q2_COUNT_TOTAL`, you can identify that the `NULL` for `PROD_ID` 100 in `Q2` is a result of a row in the original table whose measure is of `NULL` value. The `NULL` for `PROD_ID` 200 in `Q2` is due to no row being present in the original table for `prod_id` 200 in `Q2`.

Using the XML Keyword to Specify Pivot Values: Two Methods



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Using the XML keyword in the Pivot syntax requires including either the ANY keyword or a subquery.

Each output row will include the following:

- The implicit Group By columns
- A single column of XML Type containing an XML string for all value and measure pairs

The XML string for each row will hold aggregated data corresponding to the row's implicit GROUP BY value. The values of the pivot columns are evaluated at execution time.

Specifying Pivot Values: Using the ANY Keyword

```
SET LONG 1024;
SELECT *
FROM
  (SELECT product, channel, quantity_sold
   FROM sales_view
    ) PIVOT XML (SUM(quantity_sold) FOR channel IN (ANY))
ORDER BY product DESC;
```

```
PRODUCT
-----
CHANNEL_XML
-----
.
.
.
1.44MB External 3.5" Diskette
<PivotSet>
<item><column name = "CHANNEL">3</column><column name =
"SUM(QUANTITY_SOLD)">14189</column></item>
<item><column name = "CHANNEL">2</column><column name =
"SUM(QUANTITY_SOLD)">6455</column></item>
<item><column name = "CHANNEL">4</column><column name =
"SUM(QUANTITY_SOLD)">2464</column></item></PivotSet>
71 rows selected.
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The ANY keyword acts as a wildcard. If you specify ANY, all values found in the pivot column are used for pivoting. Note that when using the ANY keyword, the ANY string for each output row includes only the pivot values found in the input data corresponding to that row. The example in the slide uses the ANY wildcard keyword. The XML output includes all channel values in the sales_view view created earlier in this lesson. The ANY keyword is available only in PIVOT operations as part of an XML operation. This output includes data for cases where the channel exists in the data set. You can use wildcards or subqueries to specify the pivot IN list members when the values of the pivot column are not known.

Specifying Pivot Values: Using Subqueries

```
SELECT *
  FROM
    (SELECT product, channel, quantity_sold
      FROM sales_view
     ) PIVOT XML(SUM(quantity_sold)
      FOR channel IN (SELECT distinct channel_id
                      FROM channels));
```

```
PRODUCT
-----
CHANNEL_XML
-----
.
.
.
Y Box
<PivotSet>
<item><column name = "CHANNEL">9</column><column name =
"SUM(QUANTITY_SOLD)">1</column></item>
<item><column name = "CHANNEL">2</column><column name =
"SUM(QUANTITY SOLD)">2037</column></item>
<item><column name = "CHANNEL">5</column><column name =
"SUM(QUANTITY SOLD)"></column></item>
<item><column name = "CHANNEL">3</column><column name =
"SUM(QUANTITY SOLD)">3552</column></item>
.
.
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The example in the slide illustrates using a subquery. The XML output includes all channel values and the sales data corresponding to each channel and for each product.

Subquery-based pivots give results different from those of the ANY wildcard. In the example in the slide, when you use a subquery, the XMLtype column shows value and measure pairs for all channels for each product even if the input data has no such product/channel combination. For example, the XML string in the slide shows the highlighted Channel 5 although it has no value for the SUM (QUANTITY SOLD) column. Pivots that use a subquery, therefore, often have longer output than queries based on the ANY keyword. Depending on how you process the query results, subquery-style output may be more convenient to work with than the results derived from ANY.

Note: The results in the slide do not show a complete row of output due to space limitations.

Unpivoting the QUARTER Column: Conceptual Example

PRODUCT	Q1	Q2	Q3	Q4
Shorts		3,500	2,000	
Kids Jeans	2,500	2,000		1,500

PRODUCT	QUARTER	SUM_OF_QUANTITY
Shorts	Q2	3,500
Shorts	Q3	2,000
Kids Jeans	Q1	2,500
Kids Jeans	Q2	2,000
Kids Jeans	Q4	1,500

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

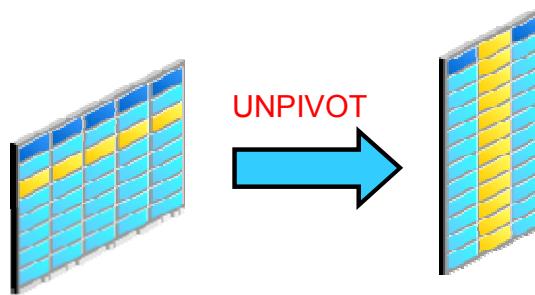
The slide example unpivots the QUARTER column, which turns the quarter columns into the values of the QUARTER column.

An UNPIVOT operation does not reverse a PIVOT operation; instead, it rotates data found in multiple columns of a single row into multiple rows of a single column. If you are working with pivoted data, an UNPIVOT operation cannot reverse any aggregations that have been made by PIVOT or any other means.

Note that the first example contains two rows before the unpivoting operation. In the second example, the unpivoting operation on the QUARTER column displays five rows. Unpivoting transforms fewer rows of input into generally more rows.

Using the UNPIVOT Operator

- UNPIVOT does not reverse a PIVOT operation; instead, it rotates data from columns into rows.
- If you are working with pivoted data, an UNPIVOT operation cannot reverse any aggregations that have been made by PIVOT or any other means.



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Data from sources such as spreadsheets and flat files is often in pivoted form. For instance, sales data will often be stored in a separate column for each time period. UNPIVOT can normalize such data, transforming multiple columns into a single column.

When the data is normalized with UNPIVOT, it is much more accessible to relational database processing with SQL. By placing data in a normalized layout, queries can readily apply SQL aggregate and analytic functions, enabling powerful analysis. Similarly, it is more efficient to specify WHERE clause predicates on normalized data.

Using the UNPIVOT Clause

- The UNPIVOT clause rotates columns from a previously pivoted table or a regular table into rows. You specify:
 - The measure column(s) to be unpivoted
 - The name or names for the columns that result from the unpivot operation
 - The columns that will be unpivoted back into values of the column specified in the `pivot_for_clause`
- You can use an alias to map the column name to another value.



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Data Types of the Value Columns in an UNPIVOT Operation

Data Type for the Value Columns	Resulting Unpivoted Column Data Type
If ALL the value columns are CHAR	CHAR
If ANY value column is VARCHAR2	VARCHAR2
If ALL the value columns are NUMBER	NUMBER
If ANY value column is BINARY_DOUBLE	BINARY_DOUBLE
If NO value column is BINARY_DOUBLE but ANY value column is BINARY_FLOAT	BINARY_FLOAT



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The UNPIVOT operation turns a set of value columns into one column. Therefore, the data types of all the value columns must be of the same data type, such as numeric or character.

- If all the value columns are CHAR, the unpivoted column is CHAR.
- If any value column is VARCHAR2, the unpivoted column is VARCHAR2.
- If all the value columns are NUMBER, the unpivoted column is NUMBER.
- If any value column is BINARY_DOUBLE, the unpivoted column is BINARY_DOUBLE.
- If no value column is BINARY_DOUBLE but any value column is BINARY_FLOAT, the unpivoted column is BINARY_FLOAT.

UNPIVOT Clause Syntax

```
table_reference UNPIVOT [{INCLUDE|EXCLUDE} NULLS]
-- specify the measure column(s) to be unpivoted.
( { column | ( column [, column]... ) }
  unpivot_for_clause
  unpivot_in_clause )
```

```
-- Specify one or more names for the columns that will
-- result from the unpivot operation.

unpivot_for_clause =
FOR { column | ( column [, column]... ) }
```

```
-- Specify the columns that will be unpivoted into values of
-- the column specified in the pivot_for_clause.

unpivot_in_clause =
( { column | ( column [, column]... ) }
  [ AS { constant | ( constant [, constant]... ) } ]
  [ , { column | ( column [, column]... ) }
    [ AS { constant | ( constant [, constant]... ) } ] ]...)
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The UNPIVOT clause rotates columns into rows.

The [INCLUDE] | [EXCLUDE] [NULLS] clause gives you the option of including or excluding null-valued rows. [INCLUDE] [NULLS] causes the UNPIVOT operation to include null-valued rows. [EXCLUDE] [NULLS] eliminates null-valued rows from the return set. If you omit this clause, the UNPIVOT operation excludes nulls.

For `column`, specify a name for each output column that holds measure values, such as `sales_quantity`.

In `pivot_for_clause`, specify a name for each output column that holds descriptor values, such as `quarter` or `product`.

In `unpivot_in_clause`, specify the input data columns whose names become values in the output columns of `pivot_for_clause`. These input data columns have names specifying a category value, such as `Q1`, `Q2`, `Q3`, and `Q4`. The optional alias enables you to map the column name to any desired value.

Creating a New Pivot Table: Example

```
CREATE TABLE pivotedtable AS
SELECT *
FROM
  (SELECT product, quarter, quantity_sold
   FROM sales_view) PIVOT (sum(quantity_sold)
    FOR quarter IN ('01' AS Q1, '02' AS Q2,
                     '03' AS Q3, '04' AS Q4));
```

Table created.

```
SELECT * FROM pivotedtable
ORDER BY product DESC;
```

PRODUCT	Q1	Q2	Q3	Q4
Y Box	1455	1766	1716	1992
Xtend Memory	3146	4121	4122	3802
...				

71 rows selected.

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The slide example creates a new table named `pivotedtable`.

Unpivoting the QUARTER Column in the SH Schema: Example

```
SELECT *
FROM pivotedtable
UNPIVOT (quantity_sold For Quarter IN (Q1, Q2, Q3, Q4))
ORDER BY product DESC, quarter;
```

PRODUCT	QUARTER	QUANTITY_SOLD
Y Box	Q1	1455
Y Box	Q2	1766
Y Box	Q3	1716
Y Box	Q4	1992
Xtend Memory	Q1	3146
Xtend Memory	Q2	4121
Xtend Memory	Q3	4122
Xtend Memory	Q4	3802
Unix/Windows 1-user pack	Q1	4259
...		
283 rows selected.		



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Before Oracle Database 11g, you could simulate the UNPIVOT syntax in the example in the slide by using existing SQL as follows:

```
SELECT product, 'Q1' as quarter, Q1 as quantity_sold
FROM pivotedTable
WHERE Q1 is not NULL
union all
SELECT product, 'Q2' as quarter, Q2 as quantity_sold
FROM pivotedTable
WHERE Q2 is not NULL
union all
SELECT product, 'Q3' as quarter, Q3 as quantity_sold
FROM pivotedTable
WHERE Q3 is not NULL
union all
SELECT product, 'Q4' as quarter, Q4 as quantity_sold
FROM pivotedTable
WHERE Q4 is not NULL;
```

The UNPIVOT syntax enables more efficient query processing. UNPIVOT alerts the query optimizer to the desired behavior. As a result, the optimizer calls highly efficient algorithms.

Unpivoting Multiple Columns in the SH Schema: Example

```
CREATE TABLE multi_col_pivot AS
SELECT *
FROM
  (SELECT product, channel, quarter, quantity_sold
   FROM sales_view) PIVOT (sum(quantity_sold) FOR (channel,
   quarter) IN ((3, '01') AS Direct_Sales_Q1,
                  (4, '01') AS Internet_Sales_Q1))
ORDER BY product DESC;
Table created.
```

```
SELECT *
FROM multi_col_pivot;
```

PRODUCT	DIRECT_SALES_Q1	INTERNET_SALES_Q1
Y Box	771	253
Xtend Memory	1935	350
...		
71 rows selected.		

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The slide example creates a pivot table using the CHANNEL and QUARTER columns in the SH schema. This is similar to “Pivoting on Multiple Columns,” earlier in this lesson. The example uses only the CHANNEL values 3 (Direct Sales) and 4 (Internet), and only the Q1 value for the QUARTER column. You can use other values for both columns. The structure of the newly created table is as follows:

```
DESCRIBE multi_col_pivot
Name          Null?    Type
-----        -----
PRODUCT      NOT NULL VARCHAR2 (50)
DIRECT_SALES_Q1      NUMBER
INTERNET_SALES_Q1      NUMBER
```

The query in the example here returns 71 rows.

Unpivoting Multiple Columns in the SH Schema: Example

```
-- Provide explicit values for the unpivot columns

SELECT *
FROM multi_col_pivot
UNPIVOT (quantity_sold For (channel, quarter) IN
    ( Direct_Sales_Q1 AS ('Direct', 'Q1'),
      Internet_Sales_Q1 AS ('Internet', 'Q1') ) )
ORDER BY product DESC, quarter;
```

PRODUCT	CHANNEL	QUARTER	QUANTITY SOLD
Y Box	Internet	Q1	253
Y Box	Direct	Q1	771
Xtend Memory	Internet	Q1	350
Xtend Memory	Direct	Q1	1935
...			
142 rows selected.			

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The example in the slide unpivots the CHANNEL and QUARTER columns using the multi_col_pivot table created in the previous slide. Note that the example uses explicit values for the unpivoted CHANNEL and QUARTER columns. The following example demonstrates unpivoting on the CHANNEL and QUARTER columns without using aliases as explicit values for the unpivoted columns. In this case, each unpivoted column uses the column name as its value.

The query in the example in the slide returns 142 rows.

```
SELECT *
FROM multi_col_pivot
UNPIVOT (quantity_sold For (channel, quarter) IN
    (Direct_Sales_Q1, Internet_Sales_Q1 ) );
```

PRODUCT	CHANNEL	QUARTER	QUANTITY SOLD
Y Box	DIRECT_SALES_Q1	DIRECT_SALES_Q1	771
Y Box	INTERNET_SALES_Q1	INTERNET_SALES_Q1	253
Xtend Memory	DIRECT_SALES_Q1	DIRECT_SALES_Q1	1935
...			

142 rows selected.

Unpivoting Multiple Aggregations in the SH Schema: Example

```
CREATE TABLE multi_agg_pivot AS
SELECT *
FROM
(SELECT product, channel, quarter, quantity_sold, amount_sold
FROM sales_view) PIVOT
(sum(quantity_sold) sumq, sum(amount_sold) suma
FOR channel IN (3 AS Direct, 4 AS Internet) )
ORDER BY product DESC;
Table created.
```

```
SELECT * FROM multi_agg_pivot;

PRODUCT      QUARTER      DIRECT_SUMQ      DIRECT_SUMA      INTERNET_SUMQ      INTERNET_SUMA
-----      -----
.
.
.
Bounce        01          1000          21738.97          347          6948.76
Bounce        02          1212          26417.37          453          9173.59
Bounce        03          1746          37781.27          528          10029.99
Bounce        04          1741          38838.63          632          12592.07
.
.
.
283 row selected.
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The example shown in the slide creates the `multi_agg_pivot` table using the `CHANNEL` column and the `amount_sold` and `quantity_sold` measures in the `SH` schema. This is similar to “Pivoting Using Multiple Aggregates,” earlier in this lesson. The example uses only the `CHANNEL` values 3 (Direct Sales) and 4 (Internet). You can use other values for the `CHANNEL` column. Note that the query creates column headings by concatenating the pivot columns with the aliases of the aggregate functions, plus an underscore.

When you use multiple aggregation, you can omit the alias for only one aggregation. If you omit an alias, the corresponding result column name is the pivot value (or the alias for the pivot value). The structure of the newly created table is as follows:

```
DESCRIBE multi_agg_pivot
Name           Null?    Type
-----
PRODUCT        NOT NULL  VARCHAR2(50)
QUARTER        NOT NULL  VARCHAR2(8)
DIRECT_SUMQ
DIRECT_SUMA
INTERNET_SUMQ
INTERNET_SUMA
```

Unpivoting Multiple Aggregations in the SH Schema: Example

```
SELECT *
FROM multi_agg_pivot
UNPIVOT ((total_amount_sold, total_quantity_sold)
For channel IN ((Direct_sumq, Direct_suma) AS 3,
                 (Internet_sumq, Internet_suma) AS 4 ))
ORDER BY product DESC, quarter, channel;
```

PRODUCT	QUARTER	CHANNEL	TOTAL_AMOUNT SOLD	TOTAL_QUANTITY SOLD
Bounce	01	3	1000	21738.97
Bounce	01	4	347	6948.76
Bounce	02	3	1212	26417.37
Bounce	02	4	453	9173.59
Bounce	03	3	1746	37781.27
Bounce	03	4	528	10029.99
Bounce	04	3	1741	38838.63
Bounce	04	4	632	12592.07
. . .				
566 rows selected.				



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The example in the slide uses the newly created `multi_agg_pivot` table. The measures `amount_sold` and `quantity_sold` are unpivoted. Channels are mapped to the value 3 for `Direct_sumq` and `Direct_suma` and to the value 4 for `Internet_sumq` and `Internet_suma`. The channel mapping is consistent with the values used in the pivot operation that created the `multi_agg_pivot` table. However, any values could have been used for the channel mappings.

Quiz

Using the XML keyword in the PIVOT clause requires the presence of _____ or _____?



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Answer: The ANY keyword or a subquery

Summary

In this lesson, you should have learned how to:

- Identify the benefits of pivoting and unpivoting operations
- Write cross-tabulation queries to pivot (rotate) column values into new columns and to unpivot (rotate) columns into column values
- Pivot and unpivot with multiple columns and multiple aggregates
- Use wildcards and aliases with pivoting operations



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Practice 6 Overview

This practice covers the following topics:

- Using the PIVOT clause to pivot `order_mode` values into columns, aggregating `order_total` data in the process, to get yearly totals by order mode
- Using the UNPIVOT clause in a query to restore the `pivot_table` table so that selected columns are pivoted into values in a single column



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

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

Pattern Matching by Using SQL

ORACLE®

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Objectives

After completing this lesson, you should be able to:

- Identify the benefits of pattern matching
- Identify tasks and keywords in pattern matching
- Use scalar expressions in pattern matching
- Handle empty matches or unmatched rows in the output
- Exclude portions of the pattern from the output



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Lesson Agenda

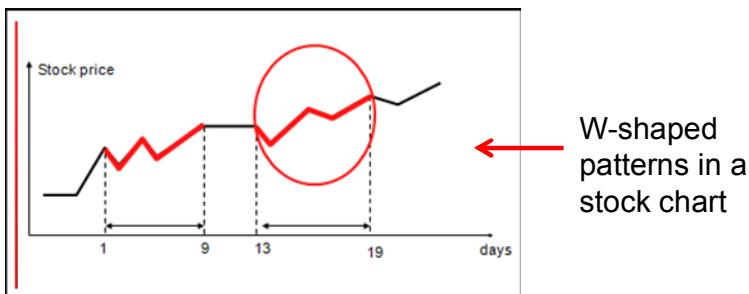
- Keywords and tasks in pattern matching
- Scalar expressions in pattern matching
- Row pattern column references
- Running versus final semantics and keywords
- Empty matches and unmatched rows
 - Excluding patterns from the output
- Rules and restrictions in pattern matching



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Benefits of Pattern Matching

- Pattern matching identifies price patterns, such as V-shapes and W-shapes in stock charts, along with performing many types of calculations.
- The ability to recognize patterns found across multiple rows is essential for many kinds of work:
 - In security applications to detect unusual behavior
 - In financial applications to seek patterns of pricing, trading volume, and other behavior



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Pattern Matching: Overview

- Pattern matching in SQL is performed using the MATCH_RECOGNIZE clause.
- The MATCH_RECOGNIZE clause performs the following tasks:



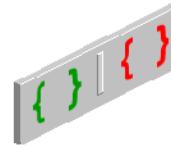
Logically partition and order the data.



Specify the logical conditions to map rows to a row pattern.



Define patterns of rows to seek.



Define expressions usable in other parts of the SQL query.

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The MATCH_RECOGNIZE clause enables you to perform the following tasks:

- Logically partition and order data that is used in the MATCH_RECOGNIZE clause with its PARTITION BY and ORDER BY clauses.
- Define patterns of rows to seek by using the PATTERN clause of the MATCH_RECOGNIZE clause. These patterns use regular expression syntax, a powerful and expressive feature, applied to the pattern variables you define.
- Specify the logical conditions required to map a row to a row pattern variable in the DEFINE clause.
- Define measures, which are expressions usable in other parts of the SQL query, in the MEASURES clause.

Keywords in Pattern Matching

- PARTITION BY: Logically divides rows into groups
- [ONE ROW | ALL ROWS] PER MATCH: For each row in the match, displays one output row or all output rows
- MEASURES: Defines calculations for export from the pattern matching
- PATTERN: Defines the row pattern that will be matched
- DEFINE: Defines primary pattern variables
- AFTER MATCH SKIP: Restarts the matching process after a match is found
- MATCH_NUMBER: Finds which rows are members of which match
- CLASSIFIER: Finds which pattern variable applies to which rows



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The keywords discussed in the slide are explained in detail in the later slides.

Pattern Matching: Example

```
CREATE TABLE Ticker (SYMBOL VARCHAR2(10), tstamp DATE, price NUMBER);
```

```
INSERT INTO Ticker VALUES('ACME', '01-Apr-11', 12);
```

...

```
SELECT *
FROM Ticker
MATCH_RECOGNIZE (
    PARTITION BY symbol
    ORDER BY tstamp
    MEASURES STRT.tstamp AS start_tstamp,
              LAST(DOWN.tstamp) AS bottom_tstamp,
              LAST(UP.tstamp) AS end_tstamp
    ONE ROW PER MATCH
    AFTER MATCH SKIP TO LAST UP
    PATTERN (STRT DOWN+ UP+)
    DEFINE
        DOWN AS DOWN.price < PREV(DOWN.price),
        UP AS UP.price > PREV(UP.price)      ) MR
ORDER BY MR.symbol, MR.start_tstamp;
```

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The example in the slide calculates simple V-shaped patterns for all cases in the ticker table where stock prices dipped to a bottom price and then rose. This query returns a single row output for each match.

The `MATCH_RECOGNIZE` clause is evaluated by the Oracle server as follows:

- `PARTITION BY` divides the data from the ticker table into logical groups where each group contains one stock symbol.
- `ORDER BY` orders the data within each logical group by `tstamp`.
- `MEASURES` defines the following three measures:
 - The time stamp at the beginning of a V-shape (`start_tstamp`)
 - The time stamp at the bottom of a V-shape (`bottom_tstamp`)
 - The time stamp at the end of the a V-shape (`end_tstamp`).
- `ONE ROW PER MATCH` indicates that the query returns only one row per match.
- `AFTER MATCH SKIP TO LAST UP` indicates that whenever you find a match, you restart your search at the row that is the last row of the `UP` pattern variable. A pattern variable is a variable used in a `MATCH_RECOGNIZE` statement, and is defined in the `DEFINE` clause.

Note: The `bottom_tstamp` and `end_tstamp` measures use the `LAST()` function to ensure that the values retrieved are the final value of the time stamp within each pattern match.

- PATTERN (STRT DOWN+ UP+) indicates the three pattern variables: STRT, DOWN, and UP. The plus sign (+) after DOWN and UP indicates that at least one row must be mapped to each of them.
- DEFINE declares the conditions that must be met for a row to map to the row pattern variables STRT, DOWN, and UP. Because there is no condition for STRT, any row can be mapped to STRT.

The result of the example displayed on the previous page is as follows:

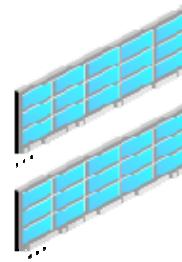
The screenshot shows the 'Script Output' window from Oracle SQL Developer. The window title is 'Script Output'. Below the title, there are several icons: a red play button, a yellow pencil, a blue folder, a green document, and a blue file. To the right of these icons, the text 'Task completed in 0.005 seconds' is displayed. The main area of the window contains a table with the following data:

SYMBOL	START_TSTAMP	BOTTOM_TSTAMP	END_TSTAMP
ACME	05-APR-11	06-APR-11	10-APR-11
ACME	10-APR-11	12-APR-11	13-APR-11
ACME	14-APR-11	16-APR-11	18-APR-11

Note: Both the DOWN and UP pattern variables take advantage of the PREV() function, which lets them compare the price in the current row to the price in the previous row. The DOWN variable is matched when a row has a lower price than the row that preceded it, so it defines the downward (left) leg of the V-shape. A row is mapped to the UP variable if the row has a higher price than the row that preceded it.

Using the PARTITION BY Clause

- You can use the PARTITION BY keyword within the SELECT statement to logically divide rows into groups.
- After you divide the input data into logical partitions, use the ORDER BY keyword to specify the order of rows within a row pattern partition.
- If you do not specify the PARTITION BY keyword during pattern matching, then all rows of the input table constitute a single row pattern partition.



ORACLE

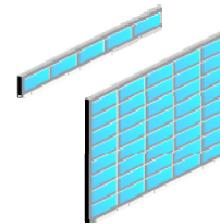
Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You can use the PARTITION BY keyword to specify one or more columns to partition the rows of a table into groups. You can then use the ORDER BY keyword to specify the order of rows within a row pattern partition.

In the pattern matching example, in the previous slide, you use the PARTITION BY keyword to restrict the query to access a single stock at a time.

Using [ONE ROW | ALL ROWS] PER MATCH Keywords

- ONE ROW PER MATCH indicates that for every pattern match found, there will be one row of output. This keyword gives the summary information of a particular match.
- ALL ROWS PER MATCH indicates that for every pattern match found, all rows are displayed in the output.



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The result of the `MATCH RECOGNIZE` clause is called the row pattern output table. The shape (row type) of the row pattern output table depends on the choice of ONE ROW PER MATCH or ALL ROWS PER MATCH keywords.

ONE ROW PER MATCH gives the summary information of a particular match. It is the default option. For an empty match, ONE ROW PER MATCH returns a summary row where the PARTITION BY columns take the values from the row where the empty match occurs, and the measure columns are evaluated over an empty set of rows.

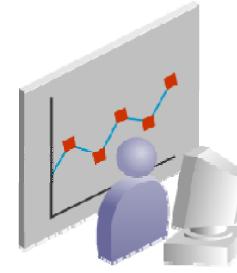
ALL ROWS PER MATCH gives the detailed information of a particular match. This keyword can be further extended to provide the following three options:

- ALL ROWS PER MATCH SHOW EMPTY MATCHES
- ALL ROWS PER MATCH OMIT EMPTY MATCHES
- ALL ROWS PER MATCH WITH UNMATCHED ROWS

The preceding three options are discussed in detail in the topic titled “Handling Empty Matches or Unmatched Rows”.

Defining Row Pattern Measure Columns

- The MEASURES clause defines a list of columns for the pattern output table.
- Each pattern measure column is defined with a column name whose value is specified by a corresponding pattern measure expression.



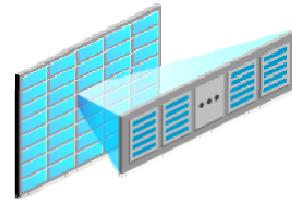
ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The pattern matching clause enables you to create expressions useful in a wide range of analyses. These are presented as columns in the output by using the MEASURES clause. The MEASURES clause defines row pattern measure columns, whose value is computed by evaluating an expression related to a particular match.

Defining the Row Pattern to be Matched

- The PATTERN clause defines the following:
 - Pattern variables that must be matched
 - Sequence in which the pattern variables must be matched
 - Number of rows that must be matched
- It also specifies a regular expression for the match search.



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The PATTERN keyword specifies the pattern to be recognized in the ordered sequence of rows in a partition. Each variable name in a pattern corresponds to a Boolean condition, which is specified later by using the DEFINE clause.

The PATTERN clause is used to specify a regular expression in a query by using the following operators.

- **Concatenation:** It is used to list two or more items in a pattern to be matched in that order. Items are concatenated when there is no operator sign between two successive items, for example, PATTERN (A B C).
- **Quantifiers:** They define the number of iterations accepted for a match. For example, the + operator indicates one or more iterations, whereas {n} indicates n iterations where n > 0.
- **Alternation:** It matches a single regular expression from a list of several possible regular expressions. The alternation list is created by placing a vertical bar (|) between each regular expression. Alternatives are preferred in the order they are specified. For example, PATTERN (A | B | C) attempts to match A first. If A is not matched, it attempts to match B. If B is not matched, it attempts to match C.

- **Grouping:** It treats a portion of the regular expression as a single unit, enabling you to apply regular expression operators such as quantifiers to that group. For example, PATTERN ((A B) {3} C) attempts to match the group (A B) three times and then seeks one occurrence of C.
- **PERMUTE:** This is used to express a pattern that is a permutation of simpler patterns. For example, PATTERN (PERMUTE (A, B, C)) is equivalent to PATTERN (A B C | A C B | B A C | B C A | C A B | C B A).
- **Exclusion:** This indicates parts of the pattern to be excluded from the output of ALL ROWS PER MATCH. The parts of the pattern to be excluded are enclosed between { - and - }.
- **Anchors:** These work in terms of positions rather than rows. They match a position either at the start or end of a partition. The ^ anchor matches the position before the first row in the partition and the \$ anchor matches the position after the last row in the partition.
- **Empty pattern ():** This matches an empty set of rows.

Defining Primary Pattern Variables

- The **DEFINE** clause is used to specify the conditions that define primary pattern variables.
- It is a mandatory clause in the **SELECT** statement for pattern matching.

```
DEFINE UP AS UP.Price > PREV(UP.Price),  
DOWN AS DOWN.Price < PREV(DOWN.Price)
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The **DEFINE** clause is used to specify the conditions that a row must meet to be mapped to a specific pattern variable. In the code snippet in the slide, **UP** is defined by the condition **UP.Price > PREV (UP.Price)**, and **DOWN** is defined by the condition **DOWN.Price < PREV (DOWN.Price)**. This implies that the value of **UP** for a particular row must be greater than that of a previous row and the value of **DOWN** for a particular row must be less than that of the previous row. **PREV** is a row pattern navigation operation that evaluates an expression by using the value stored in the previous row of a partition.

Note:

- The aggregate functions such as **COUNT**, **SUM**, **AVG**, **MAX**, and **MIN** can be used in both the **MEASURES** and **DEFINE** clauses. These functions operate on a set of rows that are mapped to a particular pattern variable.
- The **DISTINCT** keyword is not supported with aggregates.

Restarting the Matching Process

- The AFTER MATCH SKIP clause determines the point to resume row pattern matching after a non-empty match was found.
- The options include:
 - AFTER MATCH SKIP PAST LAST ROW
 - AFTER MATCH SKIP TO NEXT ROW
 - AFTER MATCH SKIP TO FIRST *pattern_variable*
 - AFTER MATCH SKIP TO LAST *pattern_variable*

```
AFTER MATCH SKIP TO X  
PATTERN (X Y+ Z) ,
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The AFTER MATCH SKIP clause determines the point to resume row pattern matching after a non-empty match was found. The default for the clause is AFTER MATCH SKIP PAST LAST ROW. This option resumes pattern matching at the next row after the last row of the current match. The other options are as follows:

- AFTER MATCH SKIP TO NEXT ROW: Resume pattern matching at the row after the first row of the current match.
- AFTER MATCH SKIP TO FIRST *pattern_variable*: Resume pattern matching at the first row that is mapped to the pattern variable.
- AFTER MATCH SKIP TO LAST *pattern_variable*: Resume pattern matching at the last row that is mapped to the pattern variable.

In the code snippet in the slide, the AFTER MATCH SKIP TO X option resumes pattern matching at the same row where the previous match was found.

Note: The AFTER MATCH SKIP TO *pattern_variable* option works similar to AFTER MATCH SKIP TO LAST *pattern_variable*.

Lesson Agenda

- Keywords and tasks in pattern matching
- Scalar expressions in pattern matching
- Row pattern column references
- Running versus final semantics and keywords
- Empty matches and unmatched rows
 - Excluding patterns from the output
- Rules and restrictions in pattern matching



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Using Scalar Expressions in Pattern Matching

Pattern matching provides the following scalar expressions that are unique to row pattern matching:

- Row pattern navigation operations
- MATCH_NUMBER function
- CLASSIFIER function



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

- Row pattern navigation operations enable navigation within the row pattern by either physical or logical offsets. The navigation operations are performed by using the following functions:
 - The PREV and NEXT functions can be used to evaluate an expression by using a previous or next row in a partition. These functions operate in terms of physical rows and are not limited to the rows mapped to a specific variable.
 - The FIRST and LAST functions return the value of an expression evaluated in the first or last row of the group of rows mapped to a pattern variable. These functions navigate only among the rows mapped to pattern variables.
- Matches within a row pattern partition are numbered sequentially, starting with 1, in the order they are found. MATCH_NUMBER() is a function that returns a numeric value with scale 0 (zero) whose value is the sequential number of the match within the row pattern partition.
- The CLASSIFIER function returns a character string whose value is the name of the pattern variable to which a row is mapped. This function is allowed in both the MEASURES and the DEFINE clauses.

Scalar Expressions in Pattern Matching: Example

```

SELECT *
FROM Ticker MATCH_RECOGNIZE (
    PARTITION BY symbol
    ORDER BY tstamp
    MEASURES
        MATCH_NUMBER() AS match_num,
        CLASSIFIER() AS var_match,
        STRT.tstamp AS start_tstamp,
        FINAL LAST(UP.tstamp) AS end_tstamp
    ALL ROWS PER MATCH
    AFTER MATCH SKIP TO LAST UP
    PATTERN (STRT DOWN+ UP+ DOWN+ UP+)
    DEFINE
        DOWN AS DOWN.price < PREV(DOWN.price),
        UP AS UP.price > PREV(UP.price) ) MR
ORDER BY MR.symbol, MR.match_num, MR.tstamp;

```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The example in the slide calculates simple W-shaped patterns for all cases in the ticker table where stock prices dipped to a bottom price and then rose two consecutive times. The result of the example is as follows:

Query Result x

SQL | All Rows Fetched: 9 in 0.007 seconds

	SYMBOL	TSTAMP	MATCH_NUM	VAR_MATCH	START_TSTAMP	END_TSTAMP	PRICE
1	ACME	05-APR-11	1	STRT	05-APR-11	13-APR-11	25
2	ACME	06-APR-11	1	DOWN	05-APR-11	13-APR-11	12
3	ACME	07-APR-11	1	UP	05-APR-11	13-APR-11	15
4	ACME	08-APR-11	1	UP	05-APR-11	13-APR-11	20
5	ACME	09-APR-11	1	UP	05-APR-11	13-APR-11	24
6	ACME	10-APR-11	1	UP	05-APR-11	13-APR-11	25
7	ACME	11-APR-11	1	DOWN	05-APR-11	13-APR-11	19
8	ACME	12-APR-11	1	DOWN	05-APR-11	13-APR-11	15
9	ACME	13-APR-11	1	UP	05-APR-11	13-APR-11	25

Lesson Agenda

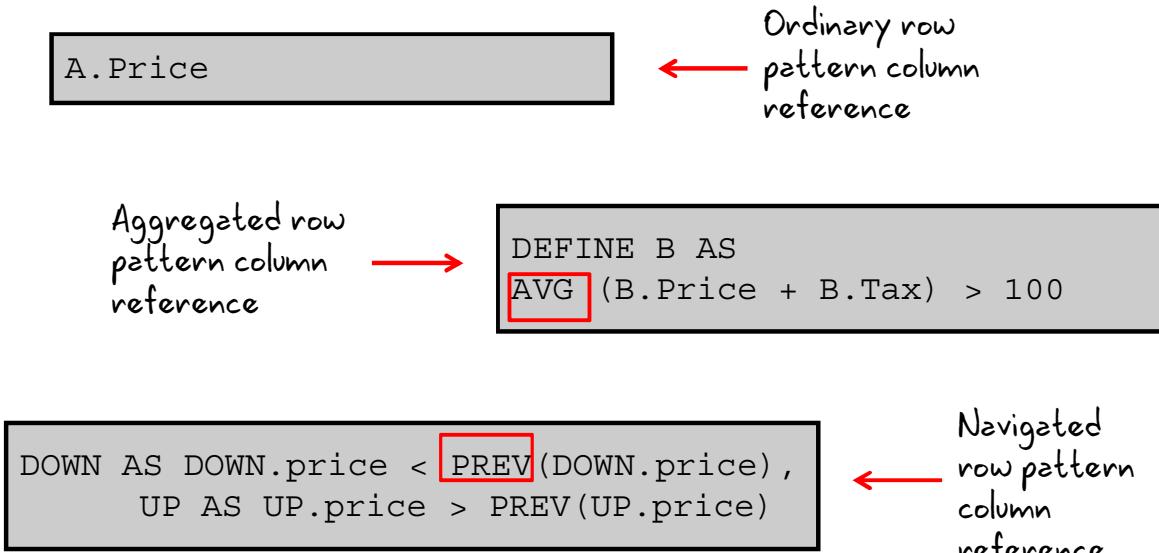
- Keywords and tasks in pattern matching
- Scalar expressions in pattern matching
- **Row pattern column references**
- Running versus final semantics and keywords
- Empty matches and unmatched rows
 - Excluding patterns from the output
- Rules and restrictions in pattern matching



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Row Pattern Column References

A row pattern column reference is a column name qualified by an explicit or implicit pattern variable.



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

A row pattern column reference is a column name qualified by an explicit or implicit pattern variable. It can be:

- Ordinary row pattern column reference
- Aggregated row pattern column reference, which can be nested within an aggregate function, such as SUM or AVG
- Navigated row pattern column reference, which can be nested within a row pattern navigation operation, such as PREV or NEXT

Note: All pattern column references in an aggregate or row pattern navigation operation must be qualified by the same pattern variable. For example, in the second example, pattern variable B is associated with both price and tax to calculate the average.

Lesson Agenda

- Keywords and tasks in pattern matching
- Scalar expressions in pattern matching
- Row pattern column references
- Running versus final semantics and keywords
- Empty matches and unmatched rows
 - Excluding patterns from the output
- Rules and restrictions in pattern matching



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Running Versus Final Semantics and Keywords

- The keywords `RUNNING` and `FINAL` are used to indicate running or final semantics, respectively.
- These keywords can be used with aggregate functions and the row pattern navigation operations `FIRST` and `LAST`.
- `RUNNING` is used in the `DEFINE` clause and `FINAL` is used in the `MEASURES` clause only



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

`RUNNING` and `FINAL` are keywords are used to indicate the running and final semantics, respectively.

Pattern matching in a sequence of rows is an incremental process where one row after another is examined to check for a matching pattern. With this incremental processing model, only a partial match is achieved at any step until all the rows are examined. Therefore, a row pattern column reference in the Boolean condition of a `DEFINE` clause has running semantics. This means that a pattern variable represents the set of rows that were already mapped to the pattern variable, up to and including the current row, but not any future rows. After the complete match is established, the final semantics is available.

Final semantics is the same as running semantics on the last row of a successful match. It is only used in the `MEASURES` clause as there is uncertainty about whether a complete match was achieved in the `DEFINE` clause.

Lesson Agenda

- Keywords and tasks in pattern matching
- Scalar expressions in pattern matching
- Row pattern column references
- Running versus final semantics and keywords
- Empty matches and unmatched rows
 - Excluding patterns from the output
- Rules and restrictions in pattern matching



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Handling Empty Matches and Unmatched Rows

- An empty match does not map any rows to pattern variables.
 - ALL ROWS PER MATCH SHOW EMPTY MATCHES option generates a single row in the row pattern output table for each empty match.
 - ALL ROWS PER MATCH OMIT EMPTY MATCHES option omits an empty match from the row pattern output table.
- Rows that are neither the starting row of an empty match, nor mapped by a non-empty match are called unmatched rows.



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

An empty match has a starting row, though it does not map any rows to pattern variables. It is assigned a sequential match number, the same as any other match.

When using ONE ROW PER MATCH, an empty match results in one row of the output table. With the ALL ROWS PER MATCH option, the question arises whether to generate a row of output for an empty match because there are no rows in the empty match. This option defaults to SHOW EMPTY MATCHES, where an empty match generates one row in the row pattern output table. In this row, the value of the CLASSIFIER() function is null and the value of the MATCH_NUMBER() function is the sequential match number of the empty match.

Some rows of the row pattern input table may be neither the starting row of an empty match, nor mapped by a non-empty match. Such rows are called unmatched rows. The option ALL ROWS PER MATCH WITH UNMATCHED ROWS shows both empty matches and unmatched rows. When displaying an unmatched row, all row pattern measures are assigned NULL values. The WITH UNMATCHED ROWS option is primarily intended for use with patterns that do not permit empty matches. However, you may specify this option if you are uncertain whether a pattern may have empty matches or unmatched rows. Then, use the COUNT and MATCH_NUMBER functions to distinguish an unmatched row from the starting row of an empty match.

Lesson Agenda

- Keywords and tasks in pattern matching
- Scalar expressions
- Row pattern column references
- Running versus final semantics and keywords
- Empty matches and unmatched rows
 - Excluding patterns from the output
- Rules and restrictions in pattern matching



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Input Table Requirements

- The row pattern input table can be a derived table, but not a joined table.

```
FROM (SELECT * FROM A LEFT OUTER JOIN B ON (A.X = B.Y))  
MATCH_RECOGNIZE (... ) M
```



- Column names in the pattern input table must be unambiguous when the row pattern input table is derived.

```
FROM (SELECT D.Name, E.Name, E.Empno, E.Salary  
      FROM Dept D, Emp E  
     WHERE D.Deptno = E.Deptno)  
MATCH_RECOGNIZE (  
    PARTITION BY D.Name  
    ...)
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The row pattern input table is the input argument to the `MATCH_RECOGNIZE` clause in the `SELECT` statement. This table can be a view, a named query (defined in a `WITH` clause), or a derived table. It cannot be a joined table.

Also, column names in the pattern input table must be unambiguous. For example, in the second code box, `Name` is ambiguous because there are two columns of the derived table with the same name. Therefore, you must disambiguate the column names within the derived table itself. The following code snippet abides by the rule because it provides aliases to the column name in the derived table itself.

```
FROM (SELECT D.Name AS Dname, E.Name AS Ename,  
        E.Empno, E.Salary  
      FROM Dept D, Emp E  
     WHERE D.Deptno = E.Deptno)  
MATCH_RECOGNIZE (  
    PARTITION BY Dname  
    ...)
```

Nesting in the MATCH_RECOGNIZE Clause

The following are prohibited in the MATCH_RECOGNIZE clause:

- Nesting one MATCH_RECOGNIZE clause within another
- Outer references in the MEASURES clause or the DEFINE subclause
- Correlated subqueries in MEASURES or DEFINE
- Recursive queries
- SELECT FOR UPDATE statement



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

A MATCH_RECOGNIZE clause cannot reference any table in an outer query block except the row pattern input table. Also, it is prohibited to feed the output of one MATCH_RECOGNIZE clause into the input of another.

Quiz

Which keyword restarts the matching process after a match is found?



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Answer: The AFTER MATCH SKIP keyword determines the point to resume the row pattern matching after a match is found.

Summary

In this lesson, you should have learned how to:

- Identify the benefits of pattern matching
- Identify tasks and keywords in pattern matching
- Use scalar expressions in pattern matching
- Handle empty matches or unmatched rows in the output
- Exclude portions of the pattern from the output



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Practice 7 Overview

In this practice, you will perform pattern matching on the Ticker3Wave table. You will:

- Display stocks where the current price is more than a specific percentage
- Find a stock with a price drop of more than eight percent



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

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

Modeling Data by Using SQL

8

ORACLE®

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Objectives

After completing this lesson, you should be able to:

- Perform interrow calculations with the SQL MODEL clause
- Use symbolic and positional references to identify cells
- Use UPDATE, UPSERT, and UPSERT ALL options
- Describe FOR loop enhancements
- Write SQL statements with the MODEL clause by using incremental loops
- Include the analytic functions within the MODEL clause



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In this lesson, you learn about the SQL MODEL clause. With the SQL MODEL clause, you can treat a query result as a multidimensional array and then apply formulas on the array to calculate new values. The formulas can be sophisticated interdependent calculations. By integrating advanced calculations into the database, performance, scalability, and manageability are enhanced significantly when compared to external solutions. You can use these computations on relational tables as well as Oracle online analytical processing (OLAP) workspaces. Rather than copying data into separate applications or PC spreadsheets, users can keep their data within the Oracle environment.

Lesson Agenda

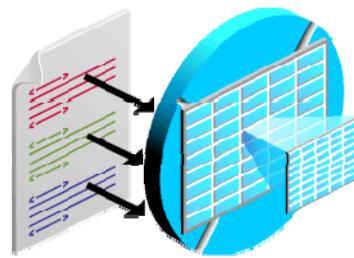
- Using the MODEL clause and cell references
- Using the CV function, the FOR construct, and distinguishing missing cells from NULLs
- Using the UPDATE, UPSERT, and UPSERT ALL options
- Reference models and cyclic rules in models



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Benefits of Integrating Interrow Calculations in SQL

- Offer analytical capabilities.
- Use the MODEL clause to integrate interrow functionality into the Oracle server.
- Enhance SQL by directly providing spreadsheet-like computations.



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Spreadsheets are a good personal productivity tool and can be used to build complex models. However, a significant scalability problem exists when the number of formulas and amount of data become large. Desktop spreadsheets have no access to the parallel processing abilities of advanced servers. In a collaborative enterprise setting with many spreadsheets, consolidation is difficult, and analyzing a business by querying multiple spreadsheets involves hard work. The Oracle server solves these problems by introducing spreadsheet-like array computations into SQL.

You use the MODEL clause to integrate interrow functionality into the Oracle server. The MODEL clause enhances SQL by directly providing spreadsheet-like computations.

Why Use SQL Modeling?

- The MODEL clause brings a new level of power and flexibility to SQL calculations.
- Use the MODEL clause to create a multidimensional array from a query results set and then apply formulas (called “rules”) to this array to calculate new values.
- Models in SQL leverage Oracle Database’s strengths in scalability, manageability, collaboration, and security.



ORACLE

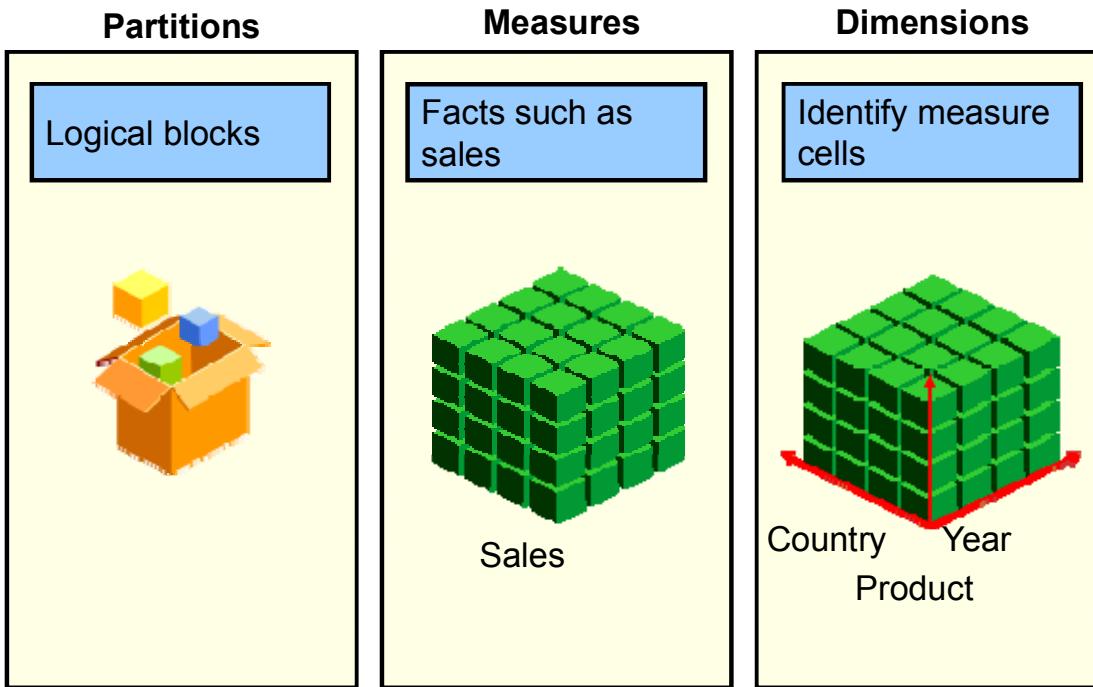
Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

SQL MODEL Clause: Features

With the SQL MODEL clause, you can perform interrow calculations within SQL. You can view the selected rows as a multidimensional array and randomly access cells within that array. Using the MODEL clause, you can specify a series of assignment statements (referred to as rules) that invoke calculations on individual cells and ranges of cells. These rules operate on the results of a query and do not update any database tables. You can use the SQL MODEL clause to perform spreadsheet computations directly in the SQL language.

The MODEL clause brings a new level of power and flexibility to SQL calculations. With the MODEL clause, you can create a multidimensional array from query results and then apply formulas (called “rules”) to this array to calculate new values. The rules can range from basic arithmetic to simultaneous equations using recursion. For some applications, the MODEL clause can replace PC-based spreadsheets. Models in SQL leverage Oracle Database’s strengths in scalability, manageability, collaboration, and security. The core query engine can work with unlimited quantities of data. By defining and executing models within the database, users avoid transferring large data sets to and from separate modeling environments. Models can be shared easily across workgroups, ensuring that calculations are consistent for all applications. Just as models can be shared, access can also be controlled precisely with Oracle’s security features. With its rich functionality, the MODEL clause can enhance all types of applications.

Partitions, Measures, and Dimensions



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

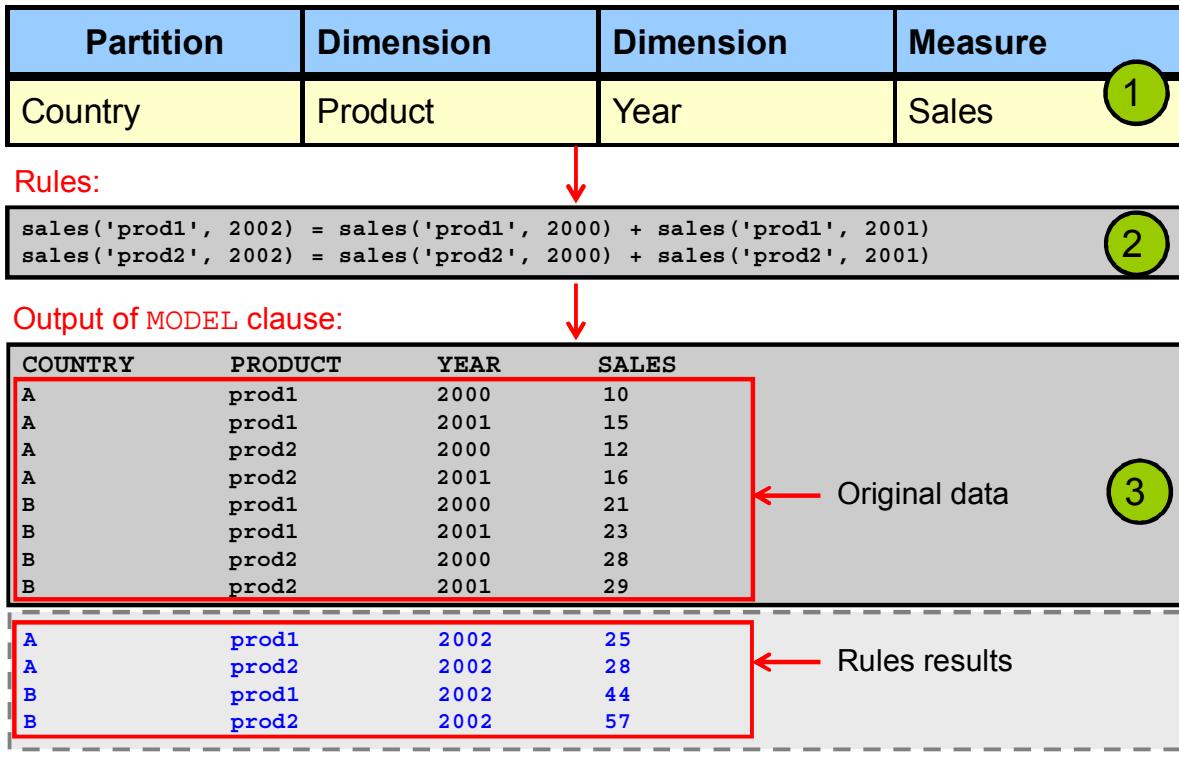
Concepts and Terminology

The MODEL clause defines a multidimensional array by mapping the columns of a query into three groups: partition, dimension, and measure columns. These elements perform the following tasks:

- Partitions define logical blocks of the result set in a way similar to the partitions of the analytic functions. Model formulas are applied to the cells of each partition.
- Measures are analogous to the measures of a fact table in a star schema. They typically contain numeric values such as sales units or cost. Each cell is accessed within its partition by specifying its full combination of dimensions.
- Dimensions identify each measure cell within a partition. These columns identify characteristics such as date, region, and product name.

To create formulas on these multidimensional arrays, you define computation rules expressed in terms of the dimension values. The rules are flexible and concise, and can use `S` and `FOR` loops for maximum expressiveness.

Learning the Concepts



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The figure shown gives a conceptual overview of the model feature using a hypothetical sales table. The table has columns for country, product, year, and sales amount. The figure has three parts.

1. The first segment shows the concept of dividing the table into partition, dimension, and measure columns.
2. The second segment shows two rules that forecast the value of Prod1 and Prod2 for the year 2002.
3. The third part shows the output of a query applying the rules to such a table with hypothetical data. The first eight rows of output are of data retrieved from the database, whereas the last four rows of output show rows that are calculated from rules. Note that the rules are applied within each partition. In the third segment, the first several rows are the original data. The outlined content is the new rule data results.

Note that the MODEL clause does not update existing data in tables or insert new data into tables. To change values in a table, the results of executing the MODEL clause must be supplied to an INSERT, UPDATE, or MERGE statement.

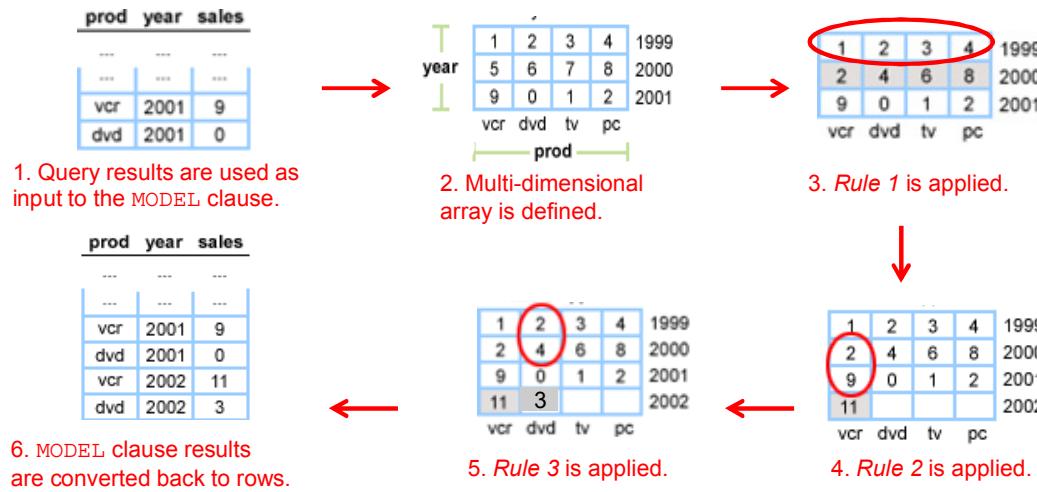
Note: The code examples in this lesson, use the SH schema.

Learning the Concepts: Example

```

MODEL                                -- incomplete example
DIMENSION BY (prod, year)
MEASURES (sales s)
RULES UPSERT
(s [ANY, 2000]=s[CV(prod), CV(year) - 1]*2,      -- Rule 1
 s[vcr, 2002]=s[vcr, 2001]+s[vcr, 2000],          -- Rule 2
 s[dvd, 2002]=AVG(s) [CV (prod), year < 2001])   -- Rule 3

```



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The figure in the slide shows the flow of processing within a simple MODEL clause. In this case, follow the data through a MODEL clause that includes three formulas. One of the formulas updates an existing value, whereas the other two create new values for a forecast. The figure shows that the rows of data retrieved by a query are fed into the MODEL clause and rearranged into an array. After the array is defined, formulas are applied one by one to the data. Finally the data, including both its updated values and newly created values, is rearranged into row form and presented as the results of the query.

Reviewing the Sample Data

```
CREATE OR REPLACE VIEW sales_view AS
  SELECT country_name country, prod_name prod,
         calendar_year year,
         SUM(amount_sold) sale, COUNT(amount_sold) cnt
    FROM   sales, times, customers, countries, products
   WHERE  sales.time_id = times.time_id AND
          sales.prod_id = products.prod_id
          AND sales.cust_id = customers.cust_id
          AND customers.country_id = countries.country_id
  GROUP BY country_name, prod_name, calendar_year;

SELECT COUNT(*)
FROM sales_view;
```

```
view SALES_VIEW created.
COUNT(*)
-----
3219
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

This sample data set is based on tables in the Sales History (SH) schema. The Sales History division tracks business statistics to facilitate business decisions.

MODEL Syntax: Example

```
SELECT SUBSTR(country,1,20) country,
       SUBSTR(prod,1,15) prod, year, sales
  FROM sales_view
 WHERE country IN ('Italy','Japan')
  MODEL RETURN UPDATED ROWS
    PARTITION BY (country)
    DIMENSION BY (prod, year)
    MEASURES (sale sales)
    RULES (
      sales['Bounce', 2002] = sales['Bounce', 2001] +
        sales['Bounce', 2000],
      sales['Y Box', 2002] = sales['Y Box', 2001],
      sales['2_Products', 2002] = sales['Bounce', 2002] +
        sales['Y Box', 2002])
 ORDER BY country, prod, year;
```

COUNTRY	PROD	YEAR	SALES
Italy	2_Products	2002	90387.54
Italy	Bounce	2002	9179.99
Italy	Y Box	2002	81207.55
Japan	2_Products	2002	101071.96
Japan	Bounce	2002	11437.13
Japan	Y Box	2002	89634.83

6 rows selected



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

This statement partitions by country, so the formulas are applied to the data one country at a time. Note that the data ends with 2001, so any rules defining values for 2002 or later insert new cells in the output. The first rule defines the sales of Bounce in 2002 as the sum of sales in 2000 and 2001. The second rule defines the sales for Y Box in 2002 as being the same value as they were for 2001. The third rule defines a category called 2_Products, which is simply the sum of the 2002 Bounce and Y Box values. Note that the values for 2_Products are derived from the results of the first two rules, so those rules must be executed before the 2_Products rule.

Syntax Guidelines

- Note that the RETURN UPDATED ROWS clause, which follows the MODEL keyword, limits the results to just those rows that were created or updated in this query. Using this clause is a convenient way to limit result sets to the newly calculated values. The RETURN UPDATED ROWS clause is used throughout the examples.
- The RULES keyword, shown in the examples at the start of the formulas, is optional, but recommended for easier reading.
- Many examples do not require ORDER BY on the COUNTRY column. It is included in the specification in case you want to modify the examples and add multiple countries.

MODEL Rules

- Only the columns that are listed in the MEASURES list can be updated in the MODEL clause.
- The MODEL clause is part of a query block.
- The MODEL clause is evaluated after all clauses in the query block except for SELECT expressions, DISTINCT, and ORDER BY.
- The SELECT and ORDER BY list cannot contain aggregates when the MODEL clause is present.

Cell References

- Must qualify all dimensions in the partition
- Consist of two types:
 - Positional: A dimension is implied by its position in the DIMENSION BY clause.



```
...
MODEL RETURN UPDATED ROWS
PARTITION BY (country)
DIMENSION BY (prod, year)
MEASURES (sale sales)
RULES ( sales['Bounce', 2000] = 10 )
```

- Symbolic: A dimension is qualified by using a Boolean condition.

```
...
sales[prod = 'Bounce', year > 1999] = 10
```

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

A reference to a cell must qualify all dimensions that are listed in the DIMENSION BY clause. You can use a symbolic reference or a positional reference.

In a positional reference, each value provided in the brackets is matched to the dimension in the equivalent position of the DIMENSION BY clause.

When you use a symbolic reference, a single dimension value is qualified by a Boolean condition. You must include as many conditions inside the brackets as there are dimensions in the DIMENSION BY clause. A single condition can qualify only one dimension and must be any single-column predicate. You can use single or multiple cell references. A single cell reference is qualified by using an equality predicate, `d=<value>`. References other than equality predicates are multiple cell references.

The symbolic and positional references handle nulls differently. With a symbolic reference such as `sales [prod=null, year=2000]`, this does not qualify any cell because `NULL=NULL` is never true. If you use the positional reference `sales [null, 2000]`, this qualifies a cell where `prod` is `null` and `year = 2000`.

The positional reference is a notation shortcut. Internally, it is interpreted as an equal predicate unless a value of null is present. If a null is present, it uses the `IS NULL` predicate.

Positional and Symbolic Cell References

- Positional:

- Dimension implied by its position in the DIMENSION BY clause
- Allows inserts and updates

```
sales ['Bounce', 2000] = 10
```

- Symbolic:

- Allows updates only

```
sales [prod = 'Bounce', year > 1999] = 10
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

When you use the symbolic notation, updates to existing rows are allowed. This means that if the row representing the left of an assignment is present in the table, the indicated cell is updated.

With the positional notation, both updates to existing rows and the insertion of new rows can occur. If the row representing the left of an assignment is present in the table, the indicated cell is updated; otherwise, it is inserted into the result set.

Using Cell Reference: Example

```
SELECT SUBSTR(country,1,20) country,
       SUBSTR(prod,1,15) prod, year, sales
  FROM sales_view
 WHERE country='Italy'
  MODEL RETURN UPDATED ROWS
    PARTITION BY (country)
    DIMENSION BY (prod, year)
    MEASURES (sale sales)
    RULES (
      sales['Bounce', 2005] = 20 )
 ORDER BY country, prod, year;
```

1

COUNTRY	PROD	YEAR	SALES
Italy	Bounce	2005	20
1 rows selected			

2

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

1. If you want to create a forecast value of SALES for the Bounce product in the year 2005 in Italy and set it to 20, then you can use a formula in the SELECT statement that sets the year value to 2005 and thus creates a new cell in the array.
2. The SQL statement generates the result set shown in the slide.

Note: If you want to create new cells, such as values for future years, you must use positional references or FOR loops (discussed later in this lesson). Positional references permit both updates and inserts into the array. This is called the UPSERT process. Symbolic references do not perform inserts; they perform only updates.

Range References

- Are references to a set of cells on the right of a rule and apply an aggregate operation to that set
- Loop over existing rows

```
...
MODEL RETURN UPDATED ROWS
PARTITION BY (country)
DIMENSION BY (prod, year)
MEASURES (sale sales)
RULES (sales['Bounce', 2005] =
       100 + max(sales)
       ['Bounce', year BETWEEN 1998 AND 2002] )
...
...
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In the example shown, you want to forecast the sales of Bounce in Italy for the year 2005 to be 100 more than the maximum sales in the period 1998 to 2002. To do so, you must use the BETWEEN clause to specify multiple cells on the right of the formula, and these are aggregated to a single value with the MAX() function.

IS ANY and ANY Predicates

The IS ANY predicate can provide an open range on both the left and right of a rule, looping over all values of a dimension.

```
sum(sales) [prod='Bounce', year IS ANY] -- symbolic
sum(sales) ['Bounce', 'ANY']           -- positional
```

This sums up all years in the partition where the products are Bounce.

Note: You can have any kind of Boolean expression on dimension keys in cell references.

Complete Query: Example

```
SELECT SUBSTR(country,1,20) country,
       SUBSTR(prod,1,15) prod, year, sales
  FROM sales_view
 WHERE country='Italy'
  MODEL RETURN UPDATED ROWS
    PARTITION BY (country)
    DIMENSION BY (prod, year)
    MEASURES (sale sales)
    RULES (
      sales['Bounce', 2005] =
        100 + max(sales)
      ['Bounce', year BETWEEN 1998 AND 2002] )
 ORDER BY country, prod, year;
```

COUNTRY	PROD	YEAR	SALES
Italy	Bounce	2005	4946.3
1 rows selected			



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The results of the sales of Bounce in Italy for the year 2005 being 100 more than the maximum sales in the period 1998 to 2002 are shown in the slide.

Lesson Agenda

- Using the MODEL clause and cell references
- Using the CV function, the FOR construct, and distinguishing missing cells from NULLs
- Using the UPDATE, UPSERT, and UPSERT ALL options
- Reference models and cyclic rules in models



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

CV() Function

- Returns the current value (CV) of a dimension from the left of an assignment
- Performs relative indexing with respect to the left

```
...  
sales['Bounce', year BETWEEN 1995 AND 2002] =  
    sales['Mouse Pad', CV(year)] +  
    0.2 * sales['Y Box', CV(year)])  
...
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The CV() function is a very powerful tool used at the right side of formulas to copy the left-side specifications that refer to multiple cells. This allows for very compact and flexible multicell formulas.

CV() allows for very flexible expressions. For instance, you can refer to other rows in the data set by subtracting from the CV(year) value. If you have the expression CV(year) - 2 in a cell reference, you can access data pertaining to two consecutive years preceding the year of the date specified.

CV () Function: Example

```

    . . .
    MODEL RETURN UPDATED ROWS
        PARTITION BY (country)
        DIMENSION BY (prod, year)
        MEASURES (sale sales)
        RULES (
            sales['Bounce', year BETWEEN 1995 AND 2002] =
            sales['Mouse Pad', CV(year)] + 0.2
                * sales['Y Box', CV(year)])
    . . .

```

COUNTRY	PROD	YEAR	SALES
...
Italy	Bounce	1999	7706.27
Italy	Bounce	2000	9527.41
Italy	Bounce	2001	20989.41
...



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You can use it to update the sales values for Bounce in Italy for multiple years by using a formula, where each year's sales is the sum of Mouse Pad sales for that year and 20 percent of the Y Box sales for that year as shown in the slide.

Note that in the results in the slide, you see values for the years 1999 through 2001 only, although any year in the range of 1995 through 2002 is accepted. This is because the table has data for only those years. The `CV()` function provides the current value of a `DIMENSION BY` key of the cell currently referenced on the left. When the left of the formula in the slide references the cells Bounce and 1999, the expression on the right resolves to:

```
sales['Mouse Pad', 1999] + 0.2 * sales['Y Box', 1999]
```

Similarly, when the left side references the cells Bounce and 2000, the expression on the right side evaluates to:

```
sales['Mouse Pad', 2000] + 0.2 * sales['Y Box', 2000]
```

The `CV()` function takes a dimension key as its argument. It is also possible to use `CV()` without any argument as in `CV()`, in which case, positional referencing is implied. The preceding formula can also be written as:

```
s ['Bounce', year BETWEEN 1995 AND 2002] =
s ['Mouse Pad', CV()] + 0.2 * s ['Y Box', CV()]
```

`CV()` functions can be used only in the right-side cell references.

Using the FOR Construct with IN List Operator

```
-- Without using a FOR construct to build a query:  
.  
.  
.  
RULES  
  ( sales['Mouse Pad', 2005] =  
    0.3 * sales['Mouse Pad', 2001],  
    sales['Bounce', 2005] = 0.3 * sales['Bounce', 2001],  
    sales['Y Box', 2005] = 0.3 * sales['Y Box', 2001] )  
.  
. . .
```

1

```
-- Using a FOR construct to express computations more  
-- concisely:  
.  
.  
.  
RULES (sales[FOR prod IN  
  ('Mouse Pad', 'Bounce', 'Y Box'), 2005] =  
  1.3 * sales[CV(prod), 2001] )  
.  
. . .
```

2

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Using the FOR Construct

The MODEL clause provides a FOR construct, which can be used inside formulas to express computations more concisely. The FOR construct is allowed on both sides of formulas. If the cell reference on the right has discrete values on each of the dimensions, you may improve performance by using the FOR construct.

When the FOR construct is evaluated, the process is called “unfolding the FOR loop.”

There is a 10,000-rule limit in the MODEL clause. Depending on the type of expressions used in the FOR construct, the FOR construct may generate virtual rules during processing. These rules are counted toward the 10,000-rule limit imposed on models. For more information about rules and the FOR loop, refer to the *Oracle Database Data Warehousing Guide*.

For example, consider the formulas that estimate the sales of several products for the year 2005 to be 30 percent higher than their sales for the year 2001. The code is shown next to label 1 in the example in the slide. The results set is as follows:

COUNTRY	PROD	YEAR	SALES
Italy	Bounce	2005	6407.24
Italy	Mouse Pad	2005	6402.63
Italy	Y Box	2005	108308.30

By using positional notation on the left of the formulas, you ensure that cells for these products in the year 2005 are inserted if they are not previously present in the array. This is rather bulky because you may have to have as many formulas as there are products. If you work with too many products, it becomes an unwieldy approach.

If you know that the needed dimension values come from a sequence with regular intervals, you can use another form of the `FOR` construct. The `INCREMENT` operator applies to all data types for which addition and subtraction are supported. This applies to number and date data types.

You can also use the `FOR ... INCREMENT` loop with strings. This uses the SQL wildcard operators “_” and “%.” The syntax is slightly different:

```
FOR dimension FROM value1 TO value2  
    FROM start_val TO stop_val INCREMENT <n>
```

This specification results in values between `value1` and `value2` by starting from `value1` and incrementing (or decrementing) by `value2`.

Using the FOR Construct with Incremental Values

```

SELECT SUBSTR(country,1,20) country,
       SUBSTR(prod,1,15) prod, year, sales
  FROM sales_view
 WHERE country='Italy'
 MODEL      RETURN UPDATED ROWS
   PARTITION BY (country)
   DIMENSION BY (prod, year)
   MEASURES (sale sales)
   RULES (
     sales['Mouse Pad'],
     FOR year FROM 2005 TO 2012 INCREMENT 1] =
       1.2 * sales[CV(prod), 2001] )
ORDER BY country, prod, year;

```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Use the example shown if you want to specify the projection sales values of mouse pads for the years 2005 to 2012 so that they are equal to 120 percent of the value in 2001.

This kind of FOR construct can be used for dimensions of numeric, date, and date-time data types. The increment or decrement expression value should be numeric for numeric dimensions and can be numeric or interval for dimensions of date or date-time types. There are other methods of using the FOR construct, and they are described in detail in the *Oracle Database Data Warehousing Guide*. The results of the example in the slide are as follows:

COUNTRY	PROD	YEAR	SALES
Italy	Mouse Pad	2005	5697.48
Italy	Mouse Pad	2006	5697.48
Italy	Mouse Pad	2007	5697.48
Italy	Mouse Pad	2008	5697.48
Italy	Mouse Pad	2009	5697.48
Italy	Mouse Pad	2010	5697.48
Italy	Mouse Pad	2011	5697.48
Italy	Mouse Pad	2012	5697.48

8 rows selected

Using the FOR Construct with a Subquery

- You can use a subquery to loop over a dimension with the FOR construct.

```
...  
sales[FOR year IN (SELECT yr FROM time_tb1),  
      FOR prod IN (SELECT product FROM prod_tb1)]  
...
```

- Follow these guidelines:
 - The subquery cannot be correlated.
 - The subquery cannot be a query defined in the WITH clause.
- FOR loops may be unfolded at query plan creation or query execution.



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You can use a subquery within the FOR construct to loop over a dimension. This is useful for densification of data, such as a procedure that guarantees that all values of a dimension are represented in the output of a query.

In the example shown, if the query on the TIME_TBL table returns 10 rows and the query on the PROD_TBL returns 100 rows, then 1,000 rules are generated.

Evaluation of Formulas with FOR Loops

FOR loops may be unfolded at either of the two stages of query processing: query plan creation or query execution.

- Query plan creation is the stage where certain rule references are resolved to create an efficient query execution plan.
- Query execution is the stage where all remaining unresolved references must be determined.

When FOR loops are unfolded at query execution, the models can be larger and include more powerful rule expressions. Additionally, you can avoid the 10,000-rule limit by the careful construction of unfolding FOR loops. For more details about the rules for unfolding FOR loops, refer to the *Oracle Database: Data Warehousing Guide*.

Using Analytic Functions in the SQL MODEL Clause

```
SELECT country, year, quarter, sale, csum
FROM sales_rollup_time
MODEL DIMENSION BY (country, year, quarter)
MEASURES (sale, gid, 0 csum)
( csum[any, any, any] =
  sum(sale) OVER (PARTITION BY country, decode(gid,0,year,null)
ORDER BY year, quarter) )
ORDER BY country, gid, year, quarter;
```

COUNTRY	YEAR	QUARTER SALE	CSUM
Japan	1998	1998-01 461847.79	461847.79
Japan	1998	1998-02 388309.93	850157.72
Japan	1998	1998-03 456844.91	1307002.63
Japan	1998	1998-04 396632.85	1703635.48
Japan	1999	1999-01 478874.52	478874.52
Japan	1999	1999-02 372006.14	850880.66
Japan	1999	1999-03 450930.72	1301811.38
Japan	1999	1999-04 423168.98	1724980.36
Japan	2000	2000-01 423976.38	423976.38
Japan	2000	2000-02 361597.37	785573.75
Japan	2000	2000-03 434912.24	1220485.99
Japan	2000	2000-04 401491.21	1621977.2
Japan	2001	2001-01 518526.36	518526.36
Japan	2001	2001-02 519492.56	1038018.92
Japan	2001	2001-03 529353.74	1567372.66
Japan	2001	2001-04 589914.39	2157287.05
Japan	1998	1703635.48	1703635.48
Japan	1999	1724980.36	3428615.84
Japan	2000	1621977.2	5050593.04
Japan	2001	2157287.05	7207880.09

20 rows selected



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You can use analytic functions on the right side of rules. The ability to use analytic functions adds expressive power and flexibility to the MODEL clause. The example in the slide combines an analytic function with the MODEL clause. First, you create a SALES_ROLLUP_TIME view that uses the GROUPING_ID() function to calculate an identifier for different levels of aggregations. Then you use the view in a query that calculates the cumulative sum of sales at both the quarter and year levels.

```
CREATE OR REPLACE VIEW sales_rollup_time AS
  SELECT country_name country, calendar_year year,
         calendar_quarter_desc quarter,
         GROUPING_ID(calendar_year, calendar_quarter_desc) gid,
         SUM(amount_sold) sale, COUNT(amount_sold) cnt
    FROM sh.sales, sh.times, sh.customers, sh.countries
   WHERE sales.time_id = times.time_id
     AND sales.cust_id = customers.cust_id
     AND customers.country_id = countries.country_id
     AND Country_name = 'Japan'
  GROUP BY country_name, calendar_year,
  ROLLUP(calendar_quarter_desc)
 ORDER BY gid, country, year, quarter;
```

Distinguishing Missing Cells from NULLS

- The two types of undetermined values are:
 - NULL in an existing cell
 - Nondetermined value from a missing cell
- The MODEL clause provides:
 - A default treatment for nulls and missing cells that is consistent with the ANSI SQL standard
 - Options to treat nulls and missing cells in other useful ways according to business logic



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

There are two types of undetermined values: a NULL in an existing cell and a nondetermined value from a missing cell. A missing cell is one for which there was no corresponding input row (no matching combination of dimension values) from the query.

By default, NULLs are treated like elsewhere in SQL except for positional references, such as `s[null, null]`, which indicates a cell whose two-dimension values are null.

Missing values are treated as NULLs in direct cell references and as nonexisting values in aggregates.

Distinguishing Missing Cells from NULLS : Using Conditions, Functions, and Clauses

Conditions, Functions, and Clauses	Description
IS PRESENT condition	Tests whether the cell referenced was present before the execution of the MODEL clause
PRESENTV function (cell, expr1, expr2)	Returns expr1 if the cell exists; otherwise, returns expr2
PRESENTNNV function (cell, expr1, expr2)	Returns expr1 if the cell exists and is not null; otherwise, returns expr2
IGNORE NAV clause	Uses defaults for nulls and missing cells: 0, empty string, 01-JAN-2001, or NULL
KEEP NAV clause	Treats missing cells as cells with NULL measure values



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The functions PRESENTV and PRESENTNNV enable you to identify missing cells and distinguish them from NULL values. These functions take a single cell reference and two expressions as arguments as in PRESENTV(cell, expr1, expr2). PRESENTV returns the first expression expr1 if the cell exists in the data input to the MODEL clause. Otherwise, it returns the second expression expr2. With the IS PRESENT condition, you can distinguish between an existing cell (a row returned in the query result) and a missing cell.

Because NULL values cause many formulas to return nulls, it may be more useful for you to treat nulls and missing values as 0 values. In this way, nulls are not propagated through a set of calculations. You can use the IGNORE NAV option (NAV stands for nonavailable values) to default nulls and missing cells to the following values:

- 0 for numeric data
- Empty string for character or string data
- 01-JAN-2001 for date-type data
- NULL for all other data types

Using the IS PRESENT Condition: Example

```

SELECT SUBSTR(country,1,20) country, SUBSTR(prod,1,15)
      prod, year
  FROM sales_view
 WHERE country='Italy'
  MODEL PARTITION BY (country)
    DIMENSION BY (prod, year)
    MEASURES (sale sales)
    ( sales['Bounce', 2001] =
        CASE WHEN sales['Bounce', 2000] IS PRESENT
              THEN sales['Bounce', 1999] ELSE 0
        END) ;

```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The IS PRESENT predicate returns true if the row indicated by the cell reference existed before the execution of the MODEL clause.

The query in the slide example checks whether the row with ['Bounce', 2000] exists and, if so, it assigns sales of Bounce in 1999 to sales of Bounce in 2001. If sales of Bounce in 2000 is not present, then it assigns 0 to the sales of Bounce in 2001.

The PRESENTV function works similarly to the IS PRESENT predicate. The function accepts three arguments, that is, a cell followed by two expressions. If the row containing the cell exists, it returns the first expression; otherwise, it returns the second expression.

```

sales['Bounce', 2001] =
PRESENTV(sales['Bounce', 2000], sales['Bounce', 1999], 0)

```

The PRESENTNNV (present and not null) function checks whether the cell is present and is not null. If the cell exists and is not null, it is assigned the first expression; otherwise, it is assigned the second expression.

```

sales['Bounce', 2001] =
PRESENTNNV(sales['Bounce', 2000], sales['Bounce', 1999], 0)

```

The partial results of the code example in the previous slide is as follows:

COUNTRY	PROD	YEAR
Italy	Laptop carrying	1998
Italy	External 101-ke	1998
Italy	Mouse Pad	1998
Italy	Multimedia spea	1998
Italy	Internal 8X CD-	1998
Italy	O/S Documentati	1998
Italy	Keyboard Wrist	1998
Italy	CD-R Mini Discs	1998
Italy	CD-R with Jewel	1998
Italy	Model K3822L Co	1998
Italy	Model C9827B Co	1998
Italy	Xtend Memory	1998
Italy	Extension Cable	1998
Italy	Multimedia spea	1999
Italy	Unix/Windows l-	1999
Italy	O/S Documentati	1999
Italy	Model C9827B Co	1999
Italy	Bounce	1999
Italy	External 101-ke	2000
Italy	SIMM- 16MB PCMC	2000
Italy	Envoy External	2000
Italy	O/S Documentati	2000
Italy	O/S Documentati	2000
Italy	Keyboard Wrist	2000
Italy	CD-RW, High Spe	2000
Italy	CD-R with Jewel	2000
Italy	256MB Memory Ca	2000
Italy	Endurance Racin	2000
Italy	Martial Arts Ch	2000
Italy	Comic Book Hero	2000
Italy	Extension Cable	2000
Italy	SMP Telephoto D	2001
. . .		
Italy	Envoy Ambassado	2001
Italy	Mini DV Camcord	2001
Italy	Laptop carrying	2001
Italy	Envoy External	2001
Italy	SIMM- 8MB PCMCI	2001
Italy	1.44MB External	2001
Italy	CD-RW, High Spe	2001
Italy	CD-R, Professio	2001
Italy	Model A3827H Bl	2001
Italy	O/S Documentati	2001

271 rows selected

Lesson Agenda

- Using the MODEL clause and cell references
- The CV function, the FOR construct, and distinguishing missing cells from NULLS
- Using the UPDATE, UPSERT, and UPSERT ALL options
- Reference models and cyclic rules in models



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Using the UPDATE, UPSERT, and UPSERT ALL Options

- Three choices for RULES behavior:
 - UPDATE: Updates existing cells
 - UPSERT: Updates existing cells and inserts new ones (default)
 - UPSERT ALL: Allows a broader set of ruled notations
- Example of UPDATE:

```
SELECT country, prod, year, sales
FROM   sales_view
MODEL
PARTITION BY (country) DIMENSION BY (prod, year)
MEASURES (sale sales) IGNORE NAV
RULES UPDATE
(sales[FOR prod IN('Bounce', 'Y Box'), 2002] =
 sales[CV(prod),2001] +
 sales[CV(prod), 2000] );
```

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You can determine how cells in rules behave by opting to have UPDATE, UPSERT, or UPSERT ALL semantics. By default, the rules in the MODEL clause have UPSERT semantics.

You can change the default behavior by using the UPDATE or UPSERT ALL options. In the example shown in the slide, the UPDATE option applies to all rules. The rule shown assigns a value to the cells for the products Bounce and Y Box in the year 2002 to the sum of sales in 2001 and 2000 for the current product—Bounce or Y Box. If a cell does not exist, a new cell is not created. The partial results of the example in the slide example are as follows:

COUNTRY	PROD	YEAR	SALES
1 United States of America	5MP Telephoto Digital Camera	1998	494553.1
2 Singapore	5MP Telephoto Digital Camera	1998	28053.97
3 Japan	17" LCD w/built-in HDTV Tuner	1998	197104.76
4 Spain	17" LCD w/built-in HDTV Tuner	1998	75557.39
...			

3217 Italy	Y Box	2001	81207.55
3218 Italy	Bounce	2001	4846.3
3219 Japan	Bounce	2001	6303.6

Using the Per-Rule Basis Option

```

SELECT country, prod, year, sales
FROM   sales_view
MODEL
  PARTITION BY (country) DIMENSION BY (prod, year)
  MEASURES (sale sales) IGNORE NAV
  RULES UPDATE
  (
    UPDATE sales['Bounce', 2002] =
      sales['Bounce', 2001] + sales['Bounce', 2000],
    UPSERT sales['Y Box', 2002] =
      sales['Y Box', 2001] + sales['Y Box', 2000],
    sales['Mouse Pad', 2002] =
      sales['Mouse Pad', 2001] + sales['Mouse Pad', 2000]
  );
  
```

#	COUNTRY	PROD	#	YEAR	#	SALES
1	Brazil	Envoy Ambassador		1998		4939.97
2	Denmark	PCMCIA modem/fax 28800 baud		1998		4846.69
3	Denmark	External 6X CD-ROM		1998		1504.45
4	Denmark	Music CD-R		1998		1068.13

...



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You can specify the UPDATE option at the MODEL clause level or on a per-rule basis. If the UPDATE or UPSERT option is on the left of a rule, the UPDATE or UPSERT option specified at the clause level is used. If no option is specified at the clause level, it defaults to UPSERT.

In the example shown in the illustration, the UPDATE option is used in the assignment for Bounce for the year 2002. If a row with ['Bounce' , 2002] does not exist, the assignment is ignored.

The default UPSERT option is used for Y Box for the year 2002. A row is updated if it exists, or created if it does not exist. The rules using UPSERT require positional notation for the cell reference or they will not insert new cells, and the behavior matches that of UPDATE rules.

Using the UPSERT and UPSERT ALL Options

- Positional upsert:

```
...  
UPSERT sales['Bounce', 2004] =  
    1.1 * sales['Bounce', 2002]  
...
```

1

- Symbolic upsert:

- Use the UPSERT option to update a cell that exists or insert a new cell if it does not exist:

```
...  
UPSERT sales['Bounce', year = 2004] =  
    1.1 * sales['Bounce', 2002];  
...
```

2

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

With the MODEL clause, you can update a cell that exists or insert a new cell. This is called “upsert” (update or insert).

UPSERT Option

When you use the UPSERT option, a new cell is created corresponding to the one referenced on the left of the rule when the cell is missing, and the cell reference contains only positional references qualified by constants. For example, the rule shown in the graphic with label 2, would not create any new cell because of the symbolic reference `year = 2004`. However, the rule with label 1 would create a new cell for the product Bounce for the year 2004 if that cell did not exist.

On a related note, new cells are not created if any of the references is ANY. This is because ANY is a predicate that qualifies all dimensional values including NULL. If there is an ANY reference for a dimension d, it means the same thing as the predicate (`d IS NOT NULL OR d IS NULL`).

UPSERT ALL Option

This option allows you to specify a broader set of ruled notations to insert new cells.

Order of Evaluation of Rules

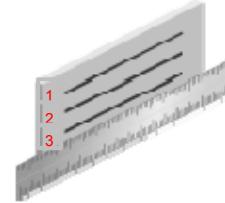
```

. . .
MODEL RETURN UPDATED ROWS
PARTITION BY (country)
DIMENSION BY (prod, year)
MEASURES (sale sales)
RULES AUTOMATIC ORDER (
    sales['2_Products', 2002] = sales['Bounce', 2002] +
        sales['Y Box', 2002],
    sales['Bounce', 2002] = sales['Bounce', 2001] +
        sales['Bounce', 2000],
    sales['Y Box', 2002] = sales['Y Box', 2001]
. . .

```

COUNTRY	PROD	YEAR	SALES
Italy	2_Products	2002	90387.54
Italy	Bounce	2002	9179.99
Italy	Y Box	2002	81207.55
Japan	2_Products	2002	101071.96
Japan	Bounce	2002	11437.13
Japan	Y Box	2002	89634.83

6 rows selected



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

By default, formulas are evaluated in the order in which they appear in the MODEL clause. You can use the SEQUENTIAL ORDER keyword to make such an evaluation order explicit. SQL models with sequential formula order of evaluation are called “Sequential Order” models.

When you specify SEQUENTIAL ORDER, the Oracle server evaluates the rules in the order in which they appear. In this case, a cell can be assigned a value more than once. SEQUENTIAL ORDER is the default. When you specify AUTOMATIC ORDER, the Oracle server evaluates the rules based on their dependency order. In this case, a cell can be assigned a value only once.

In the example in the slide, a model with three formulas creates new product values based on other products. The AUTOMATIC ORDER keywords are used to ensure that the formulas are executed in correct sequence so that no dependencies are missed. This query returns the results for the newly created 2_Products product and calculates the values for Bounce and Y Box before 2_Products:

You can force the specified order of evaluation by providing the SEQUENTIAL ORDER keyword in the RULES clause.

Ordered Rule Evaluation

This allows the specification of an ORDER BY clause for a rule so that the cells that qualify on the left of a rule are processed (that is, the rule is applied to them) in the specified order.

Order of Evaluation of Rules

```
SELECT SUBSTR(country,1,20) country,
       SUBSTR(prod,1,15) prod, year, sales
  FROM sales_view
 WHERE country IN ('Italy','Japan')
  MODEL RETURN UPDATED ROWS
    PARTITION BY (country)
    DIMENSION BY (prod, year)
    MEASURES (sale sales)
    RULES SEQUENTIAL ORDER (
      sales['2_Products', 2002] = sales['Bounce', 2002] +
      sales['Y Box', 2002],
      sales['Bounce', 2002] = sales['Bounce', 2001] +
      sales['Bounce', 2000],
      sales['Y Box', 2002] = sales['Y Box', 2001] )
 ORDER BY country, prod, year;
```

COUNTRY	PROD	YEAR	SALES
Italy	2_Products	2002	
Italy	Bounce	2002	9179.99
Italy	Y Box	2002	81207.55
Japan	2_Products	2002	
Japan	Bounce	2002	11437.13
Japan	Y Box	2002	89634.83



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The query in the slide uses sequential order and it should not calculate the values for Bounce and Y Box before 2_Products. Therefore, 2_Products is assigned null values as shown.

Nested Cell References

- Only one level of nesting is supported.
- Nested references can appear on the right of a rule for sequential and automatic order models.
- Nested cell references must be single cell references; aggregates are not supported.

```
...  
sales [product='Bounce', year =  
       best_year ['Bounce', 2003]]  
...
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In addition to simple cell references, you can use nested cell references. Cell references providing dimension values can be used within a cell reference.

In automatic order models, nested references appear on the left side of rules with some restrictions. In sequential order models, nested references appear on the left side of rules.

Assume that `BEST_YEAR` is a measure. The following demonstrates a nested reference:

```
sales [product='Bounce', year = best_year ['Bounce', 2003]]
```

Here, the nested cell reference `best_year ['Bounce', 2003]` provides the value for the dimension key `year` and is used in the symbolic reference for `year`. The `best_year` and `worst_year` measures give, for each year (`y`) and product (`p`) combination, the year for which the sales of product `p` were the highest or lowest.

Lesson Agenda

- Using the MODEL clause and cell references
- The CV function, the FOR construct, and distinguishing missing cells from NULLS
- Using the UPDATE, UPSERT, and UPSERT ALL options
- Reference models and cyclic rules in models



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Reference Models

- You can create one or more read-only multidimensional arrays, called “reference models,” and reference them in the MODEL clause to act as lookup tables for the main model.
- A reference model:
 - Is defined over a query block and has DIMENSION BY and MEASURES clauses
 - Is useful for looking up factors such as currency conversion and tax rates
 - Cannot be updated (read-only)
 - Has no rules



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In a single MODEL clause, you can reference multiple arrays with different dimensions. This is analogous to VARRAY () and other lookup functions in Excel and it enables you to relate models with different dimensionality.

For more information, see *Oracle: Data Warehousing Guide* and *Oracle Database: SQL Reference*.

Reference Models

```
...  
REFERENCE model_name ON (query)  
  DIMENSION BY (cols)  
  MEASURES (cols)  
  [reference options]  
...
```

- Created by the REFERENCE subclause of the MODEL clause
- Can be used only on the right side of formulas



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Using reference models, you can relate objects of different dimensionality. Like a main SQL model, a reference model is defined over a query block and has the DIMENSION BY and MEASURE clauses to indicate its dimensions and measures, respectively.

For example, you want to convert projected sales figures of different countries, each in their own currency, into the U.S. currency and show both figures.

Reference Models: Example

```

SELECT country, year, sales, dollar_sales
FROM sales_view
where country in ('France', 'Poland')
GROUP BY country, year
    MODEL   REFERENCE conv_ref ON
        (SELECT country, exchange_rate FROM dollar_conv_tbl)
    DIMENSION BY (country) MEASURES (exchange_rate)
        IGNORE NAV

MAIN conversion
    DIMENSION BY (country, year)
    MEASURES (SUM(sale) sales, SUM(sale) dollar_sales)
        IGNORE NAV

RULES
-- assuming that sales in France grows by 2%
(dollar_sales['France', 2002] =
    sales[CV(country), 2001] * 1.02 *
    conv_ref.exchange_rate['France'],
-- assuming that sales in Poland grows by 5%
dollar_sales['Poland', 2002] =
    sales['Poland', 2000] * 1.05 *
    exchange_rate['Poland']);
-- could have more code to show Canada, Brazil, etc.

```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Before running the slide example, you must run the following commands to create and populate the currency conversion table as a reference model as follows:

```
CREATE TABLE dollar_conv_tbl(country VARCHAR2(30), exchange_rate
NUMBER);
```

```
INSERT INTO dollar_conv_tbl VALUES('Poland', 0.25);
INSERT INTO dollar_conv_tbl VALUES('France', 0.14);
INSERT INTO dollar_conv_tbl VALUES('Canada', 0.75);
INSERT INTO dollar_conv_tbl VALUES('Brazil', 0.13);
```

The results of these commands are as follows:

```

CREATE TABLE succeeded.
1 rows inserted
1 rows inserted
1 rows inserted
1 rows inserted

```

Now, to convert the projected sales for 2002 to the U.S. dollar, you can use the dollar conversion table as a reference model as shown in the example in the slide.

The results of the code example in the slide are as follows:

COUNTRY	YEAR	SALES	DOLLAR_SALES
Poland	1999	6691.51	6691.51
France	1998	1005833.57	1005833.57
France	1999	859035.11	859035.11
Poland	1998	80.97	80.97
France	2001	1025156.84	1025156.84
France	2000	886244.61	886244.61
Poland	2000	1674.66	1674.66
Poland	2002		439.59825
France	2002		146392.396752

Note:

- A one-dimensional reference model named CONV_REF is created on rows from the DOLLAR_CONV_TBL table and its exchange_rate measure has been referenced in the rules of the main model.
- The main model (called conversion) has two dimensions: country and year, whereas the reference model CONV_REF has one dimension: country.
- There are different styles of accessing the exchange_rate measure of the reference model. For France, it is rather explicit with the model_name.measure_name notation: CONV_REF.EXCHANGE_RATE, whereas, for Poland, it is a simple measure_name reference exchange_rate. The former notation must be used to resolve any ambiguities in column names across main and reference models.

Growth rates in this example are hard-coded in the rules. The growth rate for France is two percent, for Poland it is five percent, for Canada it is 25 percent, and for Brazil it is two percent. But they could come from a separate table and you can have a reference model defined on top of that.

Your formulas could be much more flexible if they could work with growth values that are looked up from a separate table of growth rates. Such a table could cover many years and countries. You can create such a table defined as:

```
CREATE TABLE growth_rate(
    country      VARCHAR2(30),
    year         NUMBER,
    growth_rate  NUMBER);

INSERT INTO growth_rate VALUES('Poland', 2002, 2.5);
INSERT INTO growth_rate VALUES('Poland', 2003, 5);
INSERT INTO growth_rate VALUES('France', 2002, 3);
INSERT INTO growth_rate VALUES('France', 2003, 2.5);
INSERT INTO growth_rate VALUES('Canada', 2002, 2.5);
INSERT INTO growth_rate VALUES('Canada', 2003, 3.2);
INSERT INTO growth_rate VALUES('Brazil', 2002, 1.8);
INSERT INTO growth_rate VALUES('Brazil', 2003, 2.8);
```

The results of the commands are as follows:

```
CREATE TABLE succeeded.
1 rows inserted
```

The following example shows how to write a query that calculates sales for Poland, France, Canada, and Brazil, by applying the 2002 growth figures and converting the values to dollars.

To write a query that calculates sales for Poland and France, applying the 2002 growth figures and converting the values to dollars, use the following:

```

SELECT      SUBSTR(country,1,20) country, year,
            localsales, dollarsales
FROM        sales_view
WHERE       country IN
            ('Poland','France', 'Canada', 'Brazil')
GROUP BY    country, year
MODEL RETURN UPDATED ROWS
REFERENCE conv_refmodel ON (
    SELECT country, exchange_rate FROM dollar_conv_tbl)
DIMENSION BY (country c)
MEASURES (exchange_rate er) IGNORE NAV
REFERENCE growth_refmodel ON (
    SELECT country, year, growth_rate FROM growth_rate)
DIMENSION BY (country c, year y)
MEASURES (growth_rate gr) IGNORE NAV
MAIN main_model
DIMENSION BY (country, year)
MEASURES (SUM(sale) sales, 0 localsales, 0 dollarsales)
IGNORE NAV
RULES (
    localsales[FOR country IN
        ('Poland','France', 'Canada', 'Brazil'), 2002]
    =
        sales[cv(country), 2001] *
        (100 + gr[cv(country), cv(year)])/100 ,
    dollarsales[FOR country IN
        ('Poland','France', 'Canada', 'Brazil'),2002]
    =
        sales[cv(country), 2001] *
        (100 + gr[cv(country), cv(year)])/100   *
        er[cv(country)] );

```

COUNTRY	YEAR	LOCALSALES	DOLLARSALES
Poland	2002	0	0
France	2002	1055911.5452	147827.616328
Canada	2002	880698.66525	660523.9989375
Brazil	2002	5291.45202	687.8887626
4 rows selected			

Cyclic Rules in Models

- Cyclic dependency spanning two rules:

```
...  
sales['Bounce', 2002] = 1.5 * sales['Y Box', 2002],  
sales['Y Box', 2002] = 100000 / sales['Bounce', 2002]  
...
```

- Selfcyclic rule:

```
...  
sales['Bounce', 2002] = 25000 / sales['Bounce', 2002]  
...
```



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Rules can be specified in ways where the left side of one rule is referred to on the right side of that rule or some other rule. Whenever this happens, the rules have circular references (the rules are recursive). These rules have cyclic dependencies.

A cyclic dependency can involve multiple rules. It could have the form “rule A depends on rule B and rule B depends on rule A.” Alternatively, you may find a selfcyclic “rule depending on itself” form.

When rules have cyclic dependencies, the calculation is an endless loop. You can halt such calculations so that your models can complete their processing. The MODEL clause can deal with cyclic dependencies in two ways:

- You can specify that the MODEL clause should run its rules in SEQUENTIAL ORDER and repeat the rule calculations for a specific number of iterations.
- The MODEL clause checks for cyclic dependencies. If it finds any, and SEQUENTIAL ORDER is not specified, then it gives an error message.

Cyclic References and Iterations

- The `ITERATE` subclause is used to handle computations that do not have a convergence point (number of iterations to perform).

```
...
MODEL DIMENSION BY (x) MEASURES (s)
RULES UPDATE ITERATE(4)
...
```

- You can access the current iteration number by using the `ITERATION_NUMBER` system variable. This variable starts at value 0 and is incremented after each iteration.



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Using the `ITERATE` option of the `MODEL` clause, you can evaluate formulas iteratively for a specified number of times. The number of iterations is specified as an argument to the `ITERATE` clause. `ITERATE` can be specified only for `SEQUENTIAL ORDER` models. Use iterative models to calculate models where the formulas are interdependent.

The syntax of the `ITERATE` clause is:

```
ITERATE (number_of_iterations) [ UNTIL (condition) ]
```

The `number_of_iterations` argument to `ITERATE` clause is a positive integer constant. Iterative evaluation stops after finishing the specified number of iterations or when the termination condition evaluates to `TRUE`, whichever comes first. Optionally, you can specify an early termination condition to stop formula evaluation before reaching the maximum iteration. This condition is specified in the `UNTIL` subclause of `ITERATE` and is checked at the end of an iteration. Therefore, you always have at least one iteration when `ITERATE` is specified.

In some cases, you may want the termination condition to be based on the change, across iterations, in the value of a cell. Oracle Database provides a mechanism to specify such conditions by allowing you to access cell values as they existed before and after the current iteration in the `UNTIL` condition. Use the `PREVIOUS` function, which takes a single cell reference as an argument and returns the measure value of the cell as it existed after the previous iteration.

Cycles and Simultaneous Equations

```
CREATE TABLE ledger
(account VARCHAR2(20),
balance NUMBER(10,2));
```

1

```
INSERT INTO ledger VALUES ('Salary', 100000);
INSERT INTO ledger VALUES ('Capital_gains', 15000);
INSERT INTO ledger VALUES ('Net', 0);
INSERT INTO ledger VALUES ('Tax', 0);
INSERT INTO ledger VALUES ('Interest', 0);
```

2

```
SELECT * FROM ledger
MODEL IGNORE NAV
DIMENSION BY (account)
MEASURES (balance b)
RULES ITERATE (100) (
b['Net'] = b['Salary'] - b['Interest'] - b['Tax'],
b['Tax'] = (b['Salary'] - b['Interest']) * 0.38 +
b['Capital_gains'] * 0.28,
b['Interest'] = b['Net'] * 0.30 );
```

3

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You want to do financial planning for a person who earns a salary of \$100,000 and has a capital gain of \$15,000. His net income is calculated as salary minus interest payments minus taxes. He pays tax-deductible interest on a loan. He also pays taxes at two rates: 28 percent for the salary income after interest expense is deducted, and 38 percent on capital gains. This person wants his interest expense to represent exactly 30 percent of his income. How can you calculate the taxes, interest expense, and the net income that results?

All values of this scenario are stored in a table called LEDGER.

1. Create a table called LEDGER.
2. Insert rows into the LEDGER table.
3. Use the ITERATE option to have the calculations repeated as many times as desired. The first pass inserts the values stored in the LEDGER table on the right side of the formulas and creates a new set of values for NET, TAX, and INTEREST. The second pass calculates a new set of values for NET, TAX, and INTEREST using the TAX and INTEREST values calculated in the previous pass. This cycle is repeated 100 times.

The results of the example in the slide are as follows:

```
CREATE TABLE succeeded.  
1 rows inserted  
ACCOUNT          B  
-----  
Salary           100000  
Capital_gains   15000  
Net              48735.2445193929173693086003342575564068  
Tax              36644.18212478920741989881956089073613  
Interest         14620.573355817875210792580100277266922  
  
5 rows selected
```

Quiz

The _____ condition tests whether the referenced cell was present before the execution of the MODEL clause?



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Answer: The IS PRESENT condition

Summary

In this lesson, you should have learned how to:

- Perform interrow calculations with the SQL MODEL clause
- Use symbolic and positional references to identify cells
- Write SQL statements with the MODEL clause using incremental loops
- Use UPDATE, UPSERT, and UPSERT ALL options
- Describe the FOR loop enhancements
- Include the analytic functions within the MODEL clause



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In this lesson, you learned about the SQL MODEL clause. The SQL MODEL clause is a powerful enhancement that enables you to perform spreadsheet-like computations on relational data.

Practice 8 Overview

This practice covers using the MODEL clause to perform interrow calculations:

- Identifying positional cell references
- Using the CV() function
- Handling NULL values



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In this practice, you use the SH schema and write SQL SELECT statements that include the MODEL clause.

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

Data Warehousing Concepts: Overview



ORACLE®

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Objectives

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

- Benefits of a data warehouse
- Types of objects used in a data warehouse
- Schema types in a data warehouse
- Summaries to improve performance
- Benefits of using materialized views
- Benefits and types of query rewrites



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Lesson Agenda

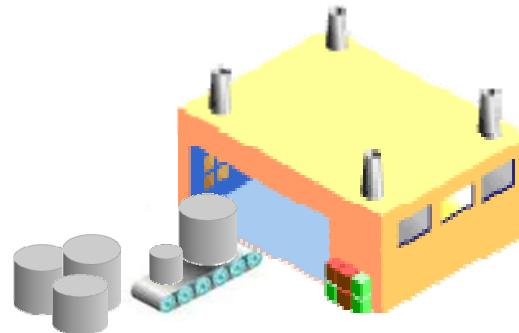
- Compare and contrast online transaction processing (OLTP) and data warehouses
- Identify data warehouse objects
- Identify the types of data warehouse schemas
- Review the benefits of using materialized views
- Review the benefits of using query rewrite
- Identify the Oracle curriculum and documentation resources



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Characteristics of a Data Warehouse

- Is a database designed for query, reporting, and analysis
- Contains historical data derived from transaction data
- Separates analysis workload from transaction workload
- Is primarily an analytical tool



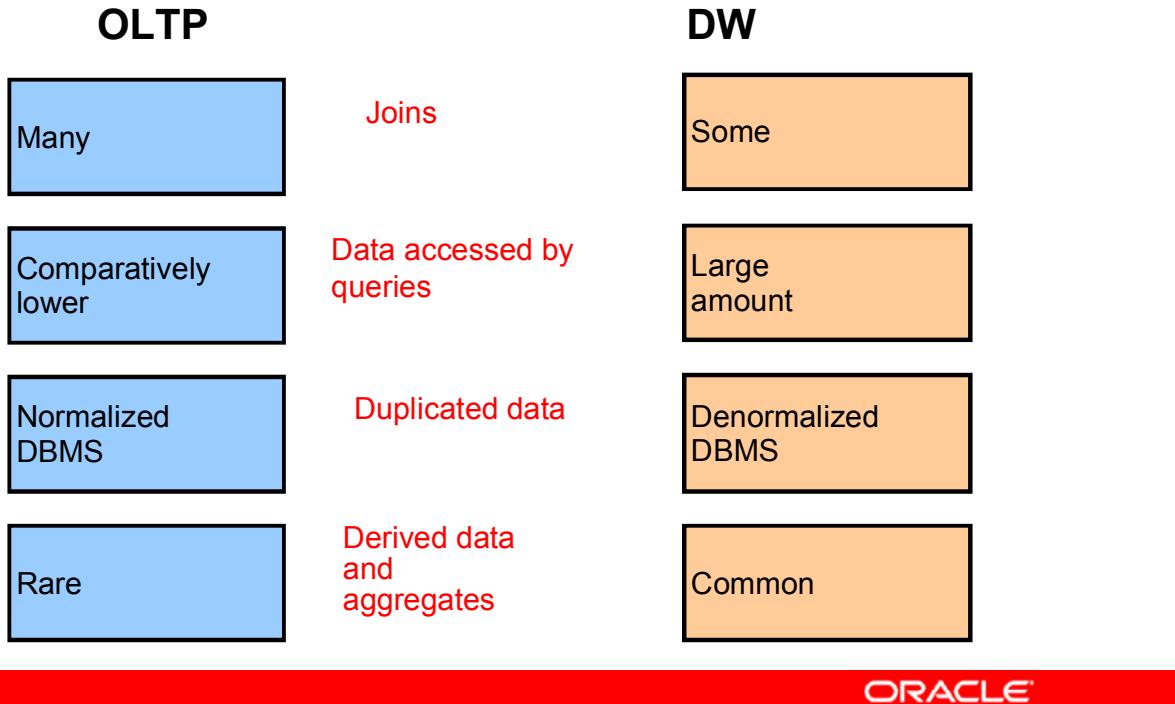
ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

A data warehouse is a database that is designed for query and analysis rather than for transaction processing. It usually contains historical data derived from transaction data, but can include data from other sources as well. Data warehouses separate analysis workload from transaction workload and enable an organization to consolidate data from several sources. The primary role that the data warehouse plays within a business entity is as an analytical tool.

In addition to a database, a data warehouse environment includes an extraction, transformation, and loading (ETL) solution, online analytical processing (OLAP) and data mining capabilities, client analysis tools, and other applications that manage the process of gathering data and delivering it to business users.

Comparing Online Transactional Processing (OLTP) with Data Warehouses (DW)



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Data warehouses and OLTP systems have very different requirements. Data warehouses are designed to accommodate ad hoc queries. You might not know the workload of your data warehouse in advance, so a data warehouse should be optimized to perform well for a wide variety of possible query operations. OLTP systems generally support predefined operations. Your applications might be specifically tuned or designed to support only these operations.

A data warehouse is updated on a regular basis by the ETL process—a carefully defined and controlled bulk data loading system. The end users of a data warehouse do not directly update the data warehouse. In OLTP systems, end users routinely issue individual data modification statements to the database. The OLTP database is always up-to-date, and reflects the current state of each business transaction. Data warehouses often use denormalized or partially denormalized schemas (such as a star schema) to optimize query performance. OLTP systems often use fully normalized schemas to optimize update, insert, and delete performance, and to guarantee data consistency. Note that the differences outlined here are generalized and you should not consider them as firm and unyielding distinctions. A typical data warehouse query scans millions of rows. For example, “Find the total sales for all customers last month.” A typical OLTP operation accesses only a handful of records. For example, “Retrieve the current order for this customer.” Data warehouses usually store many months or years of data. This is to support historical analysis. OLTP systems usually store data from only a few weeks or months. The OLTP system stores only historical data as needed to successfully meet the requirements of the current transaction.

Data Warehouses Versus OLTP

Property	OLTP	Data Warehouse
Response time	Subseconds to seconds	Seconds to hours
Operations	DML	Primarily read-only
Nature of data	Generally 30–60 days	Snapshots over time
Data organization	Application	Subject, time
Size	Small to large	Large to very large
Data sources	Operational, internal	Operational, internal, external
Activities	Processes	Analysis



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Response Time and Data Operations

Data warehouses are constructed for very different reasons than OLTP systems. OLTP systems are optimized for getting data in, that is, for storing data as a transaction occurs. Data warehouses are optimized for getting data out, that is, for providing quick responses for analysis purposes.

Rapid response is critical in the OLTP environment because there tends to be a high volume of activity. However, data warehouse applications are analytical rather than operational, so though not very critical, performance still plays an important role.

Nature of Data

The data stored in each database varies in nature. The data warehouse contains snapshots of data over time to support time-series analysis, whereas the OLTP system stores very detailed data for a short time, such as 30 to 60 days.

Data Organization

The data warehouse is subject-specific and supports analysis, so data is arranged accordingly. For the OLTP system to support subsecond response, the data must be arranged to optimize the application. For example, an order entry system may have tables that hold each of the elements of the order, whereas a data warehouse may hold the same data, but arrange it by subject such as customer, product, and so on.

Data Sources

Because the data warehouse is created to support analytical activities, data from a variety of sources can be integrated. The operational data store of the OLTP system holds only internal data or data necessary to capture the operation or transaction.

Apart from these differences, the usage curves and user expectations are also worth mentioning.

Usage Curves

Operational systems and data warehouses have different usage curves. An operational system has a more predictable usage curve, whereas the warehouse has a less predictable, more varied, and random usage curve.

Access to the warehouse varies not just on a daily basis, but may even be affected by forces such as seasonal variations. So you cannot expect the operational system to handle heavy analytical queries (Decision Support System) and at the same time be able to handle the load of transactions for the minute-by-minute processing that require fast transaction rates.

User Expectations

The difference in response time may be significant between a data warehouse and an OLTP environment fronted by personal computers. You must control the user's expectations regarding response. Set reasonable and achievable targets for query response time, which can be assessed and proved in the first increment of development. You can then define, specify, and agree on Service Level Agreements (SLA).

If users are accustomed to fast PC-based systems, they may find the warehouse excessively slow. However, it is up to those educating the users to ensure that they are aware of just how big the warehouse is, how much data there is, and of what benefit the information is to both the user and the business.

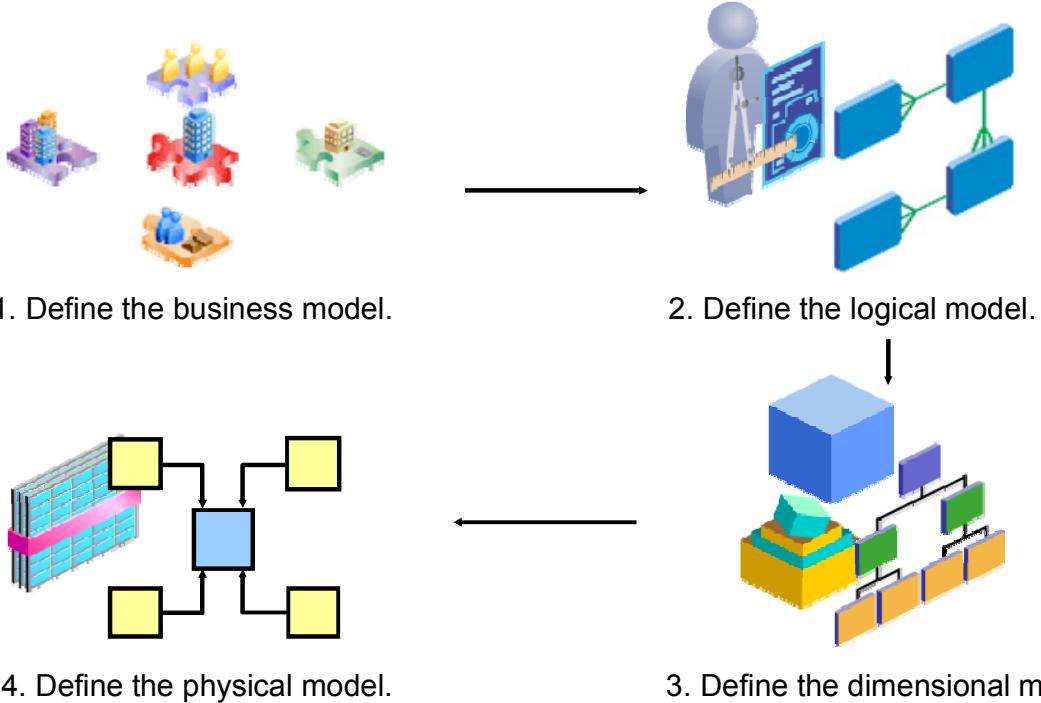
Lesson Agenda

- Compare and contrast OLTP and data warehouses
- Identify data warehouse objects
- Identify the types of data warehouse schemas
- Review the benefits of using materialized views
- Review the benefits of using query rewrite
- Identify the available Oracle curriculum and documentation resources



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Data Warehouse: Design Phases



ORACLE

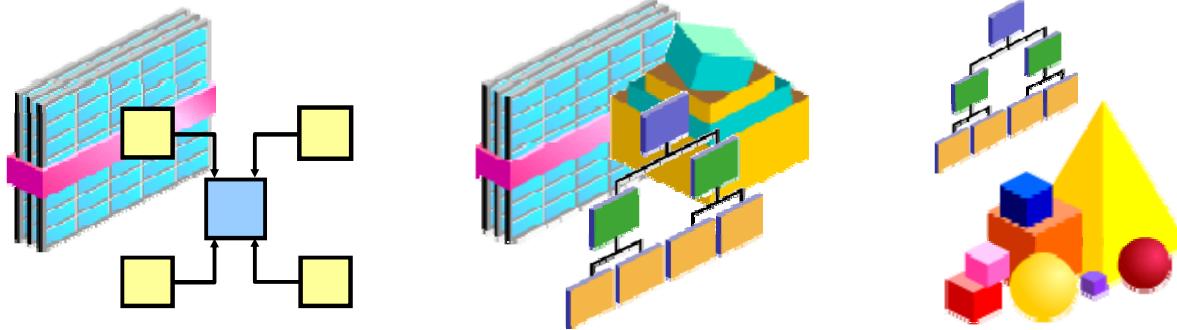
Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Several methods for designing a data warehouse have been published over the past years. Although these methods define certain terms differently, all include the same general tasks. These tasks have been grouped into four phases:

- **Defining the business model:** A strategic analysis is performed to identify business processes for implementation in the warehouse. Then, a business requirements analysis is performed, where the business measures and business dimensions for each business process are identified and documented.
- **Defining the logical model:** In the logical design, you look at the business processes and identify the logical relationships among the objects. The logical design is more conceptual and abstract than the physical design. Various methods of data modeling exist, each using a variety of diagrammatic conventions and tools. The most popular approach is called the entity-relationship (ER) approach developed by Peter Chen in the late 1970s.
- **Defining the dimensional model:** The business model is transformed into a dimensional model. Warehouse schema tables and table elements are defined, relationships between schema tables are established, and sources for warehouse data elements are recorded.

- **Defining the physical model:** In the physical design, you look at the most effective way of storing and retrieving the objects as well as handling them from a transportation and backup or recovery perspective. The dimensional model is transformed into a physical model. This includes the documentation of data element formats, database size planning, and the establishment of partitioning strategies, indexing strategies, and storage strategies.

Data Warehousing Objects



Fact tables: Large tables that store business measurements

Dimension tables: A structure composed of one or more hierarchies that categorize data

Relationships: Guarantee integrity of business information

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Fact tables and dimension tables are the two types of objects commonly found in dimensional data warehouse schemas.

Fact tables are the large tables in your data warehouse schema that store business measurements. Measures are the data that you want to analyze such as `total_sales` or `unit_cost`. Fact tables typically contain facts and foreign keys to the dimension tables. Fact tables represent data that can be analyzed and examined. Examples include `SALES`, `COST`, and `PROFIT`.

Dimension tables, also known as lookup or reference tables, contain the relatively static data in the data warehouse. Dimension tables store the information you normally use to contain queries. Dimension tables are usually textual and descriptive. You can use them as the row headers of the result set. Examples are `CUSTOMERS` or `PRODUCTS`.

Relationships guarantee the integrity of business information. An example is that if a business sells something, there is obviously a customer and a product. Designing a relationship between the sales information in the fact table and the dimension tables products and customers enforces the business rules in the databases.

Note: A dimension does not necessarily have to have hierarchies associated with it; however, it is more useful to have one or more hierarchies associated with a dimension that enables you to drill on the dimension among other things.

Characteristics of Fact Tables

- Contain numerical measures of the business
- Hold large volumes of data
- Grow quickly
- Can contain base, derived, and summarized data
- Are typically additive
- Are joined to dimension tables through foreign keys that reference primary keys in the dimension tables

Sales (Fact Table)

PROD_ID
CUST_ID
TIME_ID
CHANNEL_ID
PROMO_ID
QUANTITY SOLD
AMOUNT SOLD
...



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

A fact table typically has two types of columns: those that contain numeric facts (often called measures) and those that are foreign keys to dimension tables. The fact table is the largest table in the star schema and is composed of large volumes of data, usually making up 90 percent or more of the total database size. A fact table contains detail-level facts or facts that have been aggregated. Fact tables that contain aggregated facts are sometimes called summary tables. A fact table usually contains facts with the same level of aggregation. Though most facts are additive, they can also be semiadditive or nonadditive. Additive facts can be aggregated by simple arithmetical addition. A common example of this is sales. Nonadditive facts cannot be added at all. An example of this is averages. Semiadditive facts can be aggregated along some of the dimensions and not along others. An example of this is inventory levels, where you cannot tell what a level means by looking at it.

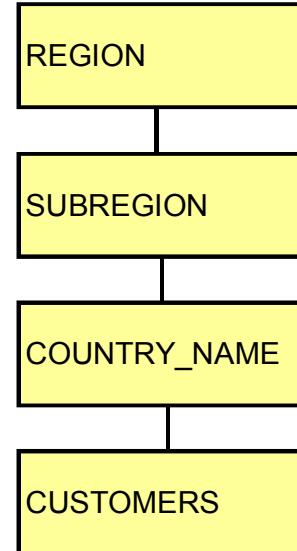
You must define a fact table for each star schema. From a modeling standpoint, the primary key of the fact table is usually a composite key that is made up of all its foreign keys.

Although a star schema typically contains one fact table, other schemas can contain multiple fact tables.

Dimensions and Hierarchies

- A dimension is a structure composed of one or more hierarchies that categorize data.
- Dimensional attributes help to describe the dimensional value.
- Dimension data is collected at the lowest level of detail and aggregated into higher level totals.
- Hierarchies:
 - Are ordered levels that organize the data
 - Enable you to aggregate and drill on the data

*CUSTOMERS
dimension hierarchy*



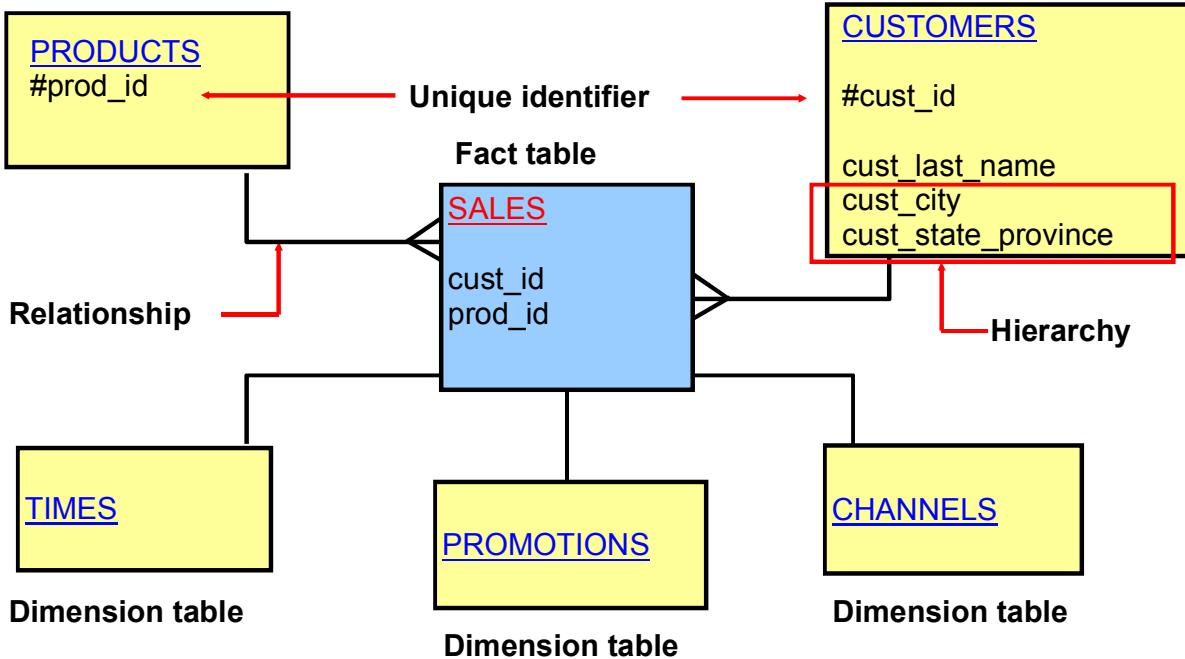
ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Dimensions are the textual descriptions of the business attributes. Dimension tables are typically smaller than fact tables and the data changes less frequently. A dimension is a structure composed of one or more hierarchies that categorize or describe the data. Dimensions are descriptive labels that provide supplemental information about facts and are stored in dimension tables. They are normally descriptive, textual values. Several distinct dimensions, combined with facts, enable you to answer business questions (Why, How, What, and When). Dimensions are essential for analysis. Commonly used dimensions are customers, products, and time. Dimension data is typically collected at the lowest level of detail and then aggregated into higher level totals that are more useful for analysis. These natural rollups or aggregations within a dimension table are called hierarchies. Hierarchies are logical structures that use ordered levels as a means of organizing data. A hierarchy can be used to define data aggregation. For example, in a time dimension, a hierarchy might aggregate data from the month level to the quarter level to the year level. Within a hierarchy, each level is logically connected to the levels above and below it. Data values at lower levels aggregate into the data values at higher levels. A dimension can be composed of more than one hierarchy. For example, in the product dimension, there might be two hierarchies—one for product categories and one for product suppliers.

Note: Oracle supports both level- and value-based hierarchies.

Dimensions and Hierarchies



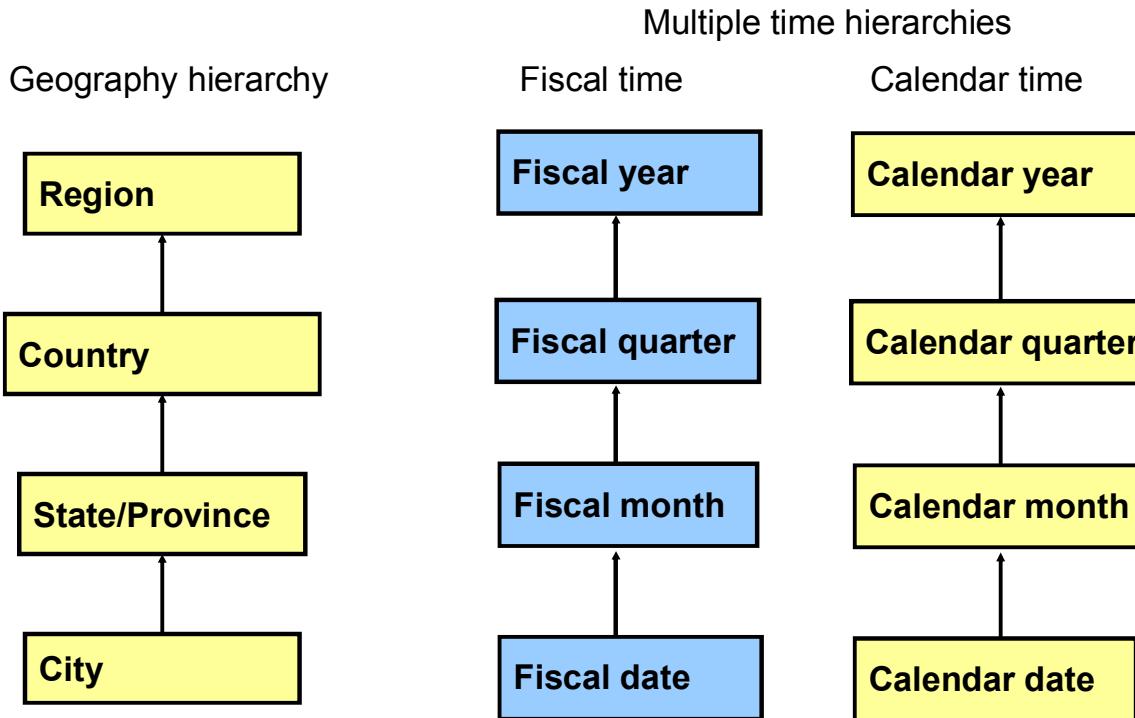
ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Dimension hierarchies also group levels from general to granular. Query tools use hierarchies, allowing you to drill down into your data to view different levels of granularity. This is a key benefit of a data warehouse. When designing hierarchies, you should consider the relationships in business structures. Hierarchies impose a family structure on dimension values. For a level value, a value at the next higher level is its parent and values at the next lower level are its children. These relationships allow you to access data quickly.

A level represents a position in a hierarchy. For example, a time dimension might have a hierarchy that represents data at the month, quarter, and year levels. Levels range from general to specific, with the root level as the highest or most general level. The levels in a dimension are organized into one or more hierarchies. Level relationships specify top-to-bottom ordering of levels from most general (the root) to most specific information. They define the parent-child relationship between the levels in a hierarchy. Hierarchies are also essential components in enabling more complex rewrites. For example, the database can aggregate an existing sales revenue on a quarterly base to a yearly aggregation when the dimensional dependencies between quarter and year are known. The dimension statement stores metadata about hierarchies within a row of a dimension table.

Identify Hierarchies for Dimensions



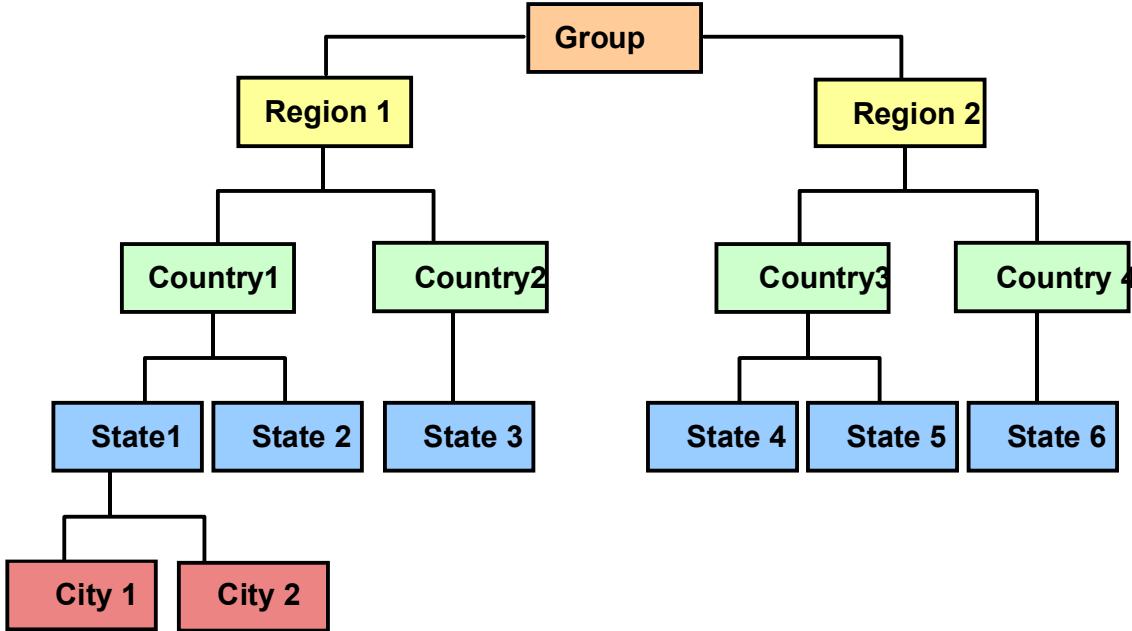
Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Dimension tables contain hierarchical data. There can be a single hierarchy or multiple hierarchies for a dimension. The example in the slide shows the Geography hierarchy associated with the Customers dimension, and the Time dimension described by multiple hierarchies to support both calendar and fiscal year.

Representation of time is critical in the data warehouse. You may decide to store multiple hierarchies in the data warehouse to satisfy the varied definitions of units of time. If you are using external data, you may find that you create a hierarchy or translation table simply to be able to integrate the data.

A simple time hierarchy corresponds to a calendar approach: days, months, quarters, and years. You can also have a hierarchy based on fiscal approach: days, months, quarters, and years.

Using Hierarchies to Drill on Data



ORACLE

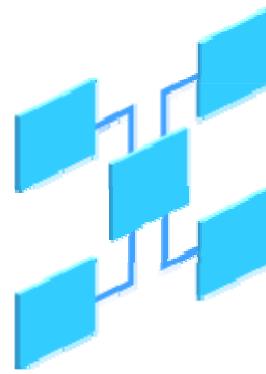
Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Hierarchies in dimensions enable you to perform data drilling. Drilling refers to the investigation of data to greater or lesser detail from a starting point. Typically, in an analytical environment, you start with less detail, at a higher level within a hierarchy, and investigate down through a hierarchy to greater detail. This process is drilling down (to more detailed data). Drilling down means retrieving data at a greater level of granularity. For example, you may want to analyze the sales revenue or profits at the Region level, and further drill down to Country, State, and City levels.

Drilling up is the reverse of this process. Consider the Geography hierarchy example. If your starting point was an analysis of data at the City level, drilling up would mean looking at a lesser level of detail, such as country, or the region, or group level.

Data Warehousing Schemas

- Objects can be arranged in data warehousing schema models in a variety of ways:
 - Star schema
 - Snowflake schema
 - Third normal form (3NF) schema
 - Hybrid schemas
- The source data model and user requirements should steer the data warehouse schema.
- Implementation of the logical model may require changes to adapt it to your physical system.



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

A schema is a collection of database objects that includes tables, views, indexes, and synonyms. You can arrange schema objects in the schema models designed for data warehousing in a variety of ways. Most data warehouses use a dimensional model. The model of your source data and the requirements of your users help you design the data warehouse schema. You can sometimes get the source model from your enterprise data model and reverse-engineer the logical data model for the data warehouse from this. The physical implementation of the logical data warehouse model may require some changes to adapt it to your system parameters—size of machine, number of users, storage capacity, type of network, and software.

A common data warehouse schema model is the star schema. However, there are other schema models that are commonly used for data warehouses. The most prevalent of these schema models is the third normal form (3NF) schema. The snowflake schema is a type of star schema, but slightly more complex. Additionally, some data warehouse schemas are not star schemas or 3NF schemas, but instead share characteristics of both. These are referred to as hybrid schema models. The important thing to remember when designing your schema is not to get lost in theory and academic comparisons. Most successful data warehouses employ a hybrid approach to schemas today.

Schema Characteristics

- Star schema:
 - It is characterized by one or more large fact tables and a number of much smaller dimension tables.
 - Each dimension table is joined to the fact table using a primary key to foreign key join.
- Snowflake schema:
 - Dimension data is grouped into multiple tables instead of one large table.
 - The number of dimension tables are increased, requiring more foreign key joins.
- Third normal form (3NF) schema
 - It is a classical relational-database model that minimizes data redundancy through normalization.



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The star schema is perhaps the simplest data warehouse schema. It is called a star schema because the entity-relationship diagram of this schema resembles a star, with points radiating from a central table. A star schema is characterized by one or more very large fact tables that contain the primary information in the data warehouse, and a number of much smaller dimension tables (or lookup tables), each of which contains information about the entries for a particular attribute in the fact table.

A star query is a join between a fact table and a number of dimension tables. Each dimension table is joined to the fact table using a primary key to foreign key join, but the dimension tables are not joined to each other. The cost-based optimizer recognizes star queries and generates efficient execution plans (star transformation) for them.

Snowflake schemas normalize dimensions to eliminate redundancy. That is, the dimension data has been grouped into multiple tables instead of one large table. For example, a product dimension table in a star schema might be normalized into a products table, a product_category table, and a product_manufacturer table in a snowflake schema. While this saves space, it increases the number of dimension tables and requires more foreign key joins. The result is more complex queries and reduced query performance.

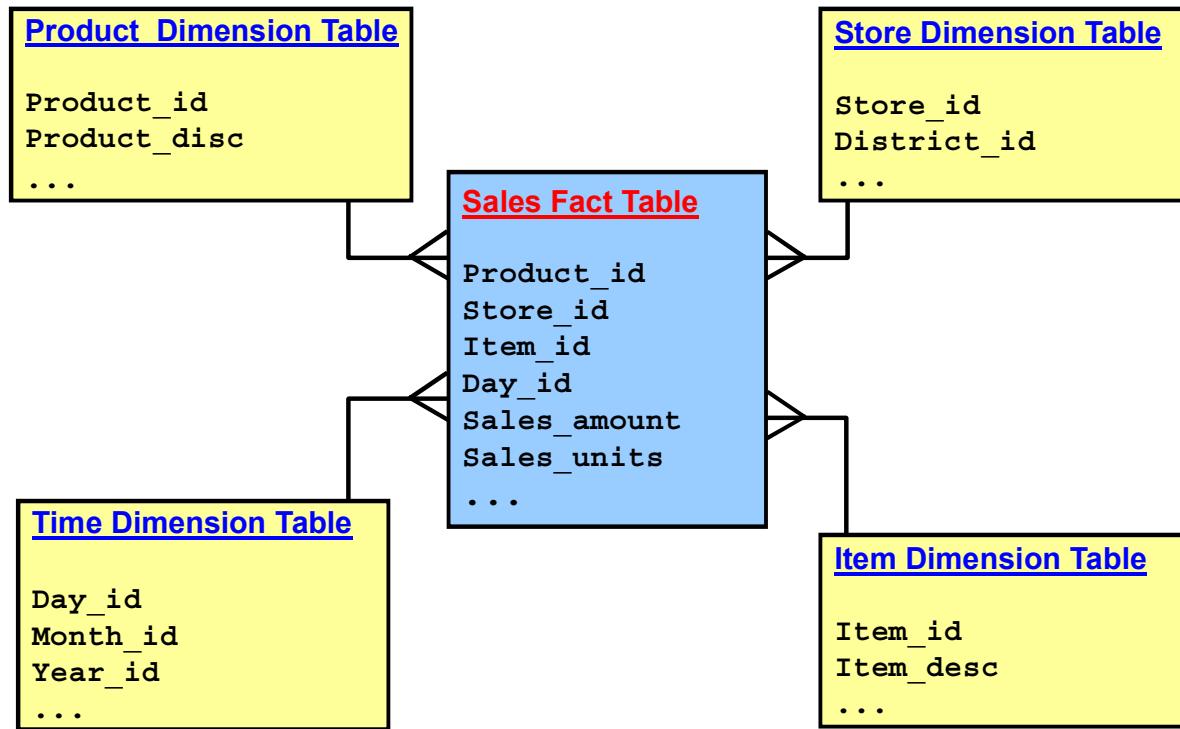
3NF schema is a classic relational-database modeling technique that minimizes data redundancy through normalization. A relationship can be considered to be 3NF if none of the nonprimary key attributes are a fact about any other nonprimary key attribute. When compared to a star schema, a 3NF schema typically has a larger number of tables due to this normalization process.

3NF schemas are typically chosen for large data warehouses, especially environments with significant data-loading requirements that are used to feed data marts and execute long-running queries.

Note:

- Today, most of the very large data warehouse schemas are not star schemas or 3NF schemas, but instead share characteristics of both schemas; these are referred to as hybrid schema models.
- Normalized structures store the greatest amount of data in the least amount of space. Entity relationship modeling (ERM) also seeks to eliminate data redundancy. This is immensely beneficial to transaction processing, OLTP systems.
- Dimensional modeling (DM) is a design that presents the data in an intuitive manner and allows for high-performance access. For these two reasons, dimensional modeling, such as star and snowflake schemas, has become the standard design for data marts and data warehouses.

Star Schema Model: Central Fact Table and Denormalized Dimension Tables



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

A star schema model can be depicted as a simple star: a central table contains fact data, and multiple tables radiate out from it, connected by database primary and foreign keys. Unlike other database structures, a star schema has denormalized dimensions.

A star model:

- Is easy to understand by the users because the structure is simple and straightforward
- Provides fast response to queries with optimization and reductions in the physical number of joins required between fact and dimension tables
- Contains simple metadata
- Is supported by many front-end tools
- Is slow to build because of the level of denormalization

The star schema is emerging as the predominant model for data warehouses and data marts.

Star Dimensional Schema: Advantages

- Supports multidimensional analysis
- Creates a design that improves performance
- Enables optimizers to yield better execution plans
- Parallels end-user perceptions
- Provides an extensible design
- Broadens the choices for data access tools

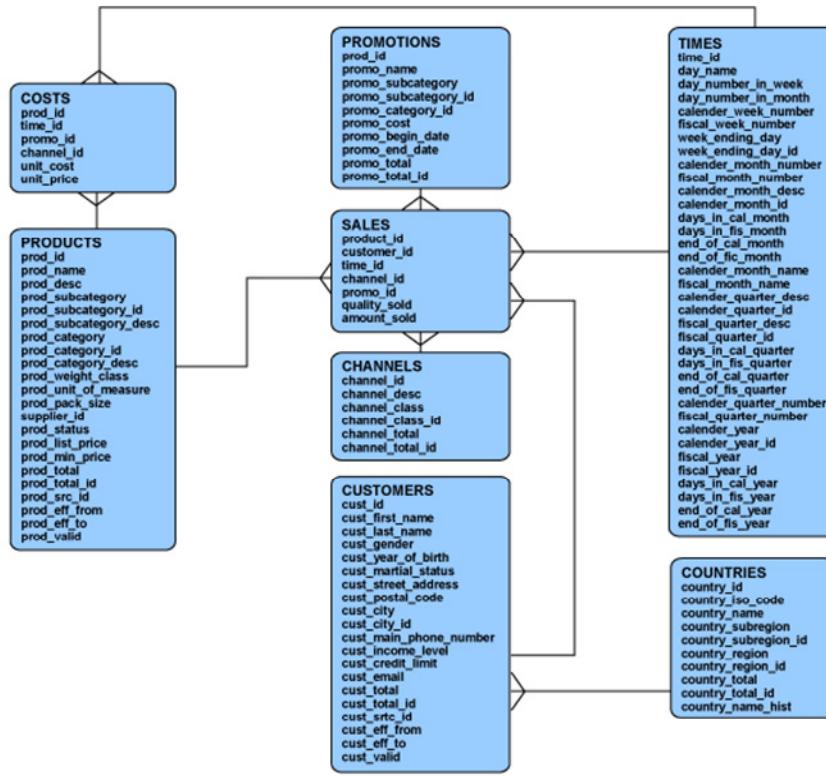


Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

- Provides rapid analysis across different dimensions for drilling down, rotation, and analytical calculations for the multidimensional cube
- Creates a database design that improves performance
- Enables database optimizers to work with a more simple database design to yield better execution plans
- Parallels how end users usually think of and use the data
- Provides an extensible design, which supports changing business requirements
- Broadens the choices for data access tools because some products require a star schema design

Note: The definitions of star and snowflake models vary among practitioners. Here, the assumption is that the star model contains a fact table with one level of related dimensional tables, which may have an Oracle hierarchy defined over columns in the one table. An example is Sales Fact and Product Dimension. The snowflake, on the other hand, has more than one level of dimensional tables, which may have an Oracle hierarchy defined over columns from multiple tables in the single dimension.

Star Dimensional Modeling: The SH Schema



ORACLE

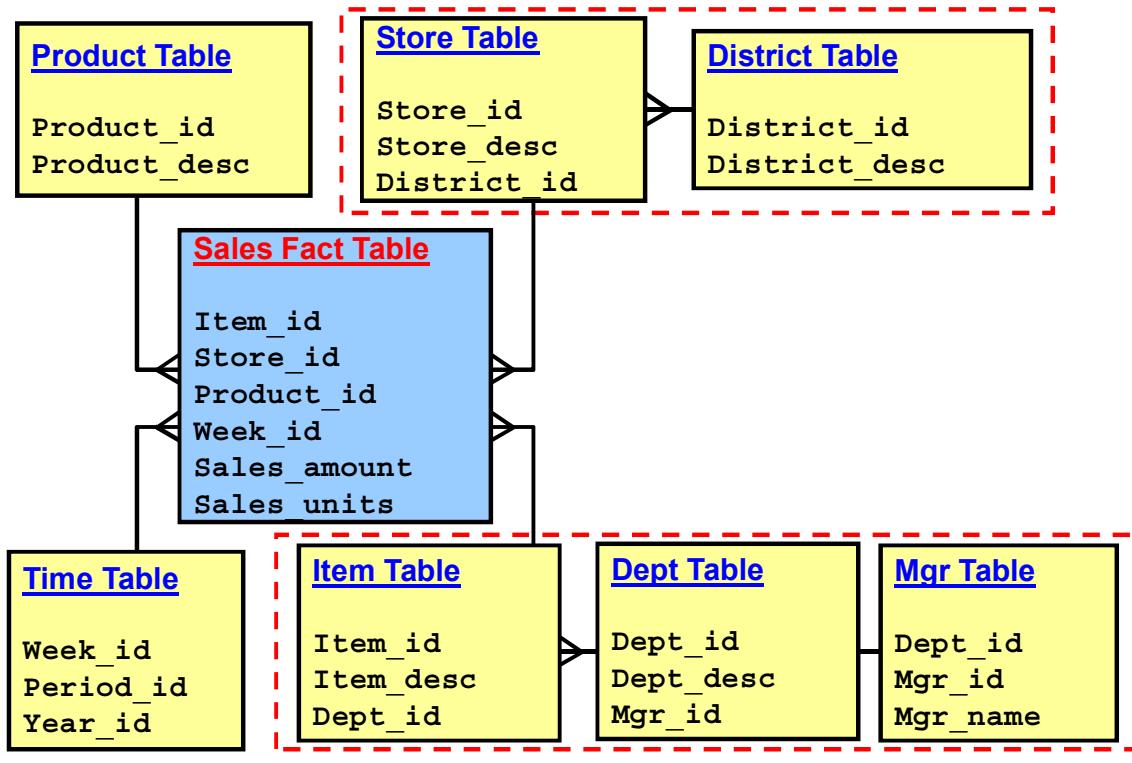
Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Star dimensional modeling is a logical design technique that seeks to present the data in a standard framework that is intuitive and provides high performance. Every dimensional model is composed of one table called the fact table, and a set of smaller tables called dimension tables. This characteristic (denormalized, star-like structure) is commonly known as a star model. Within this star model, redundant data is posted from one object to another for performance considerations.

A fact table has a multipart primary key composed of two or more foreign keys and expresses a many-to-many relationship. Each dimension table has a single-part primary key that corresponds exactly to one of the components of the multipart key in the fact table.

The slide depicts the star dimensional model of the SH schema, where Sales is the facts table, joined with the dimensions such as CHANNELS, COUNTRIES, TIMES, PROMOTIONS, PRODUCTS, and so on.

Snowflake Schema Model



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

According to Ralph Kimball, “A dimension is said to be snowflaked when the low cardinality fields in the dimension have been moved to separate tables and linked back into the original table with artificial keys.”

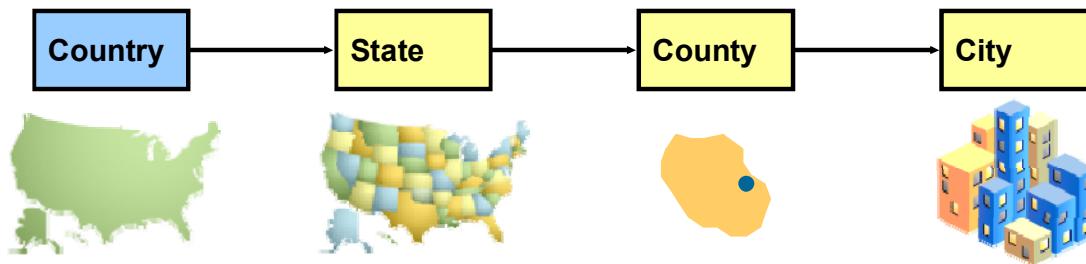
A snowflake model is closer to an entity relationship diagram than the classic star model because the dimension data is more normalized. Developing a snowflake model means building class hierarchies out of each dimension (normalizing the data).

One of the major reasons why the star schema model has become more predominant than the snowflake model is its query performance advantage. In a warehouse environment, the snowflake’s quicker load performance is much less important than its slower query performance.

Note: Each extra dimension table in a snowflake schema often represents another level of a hierarchy.

Snowflake Schema Model

- Can be used directly by some tools
- Is more flexible to change
- Provides for speedier data loading
- Can become large and unmanageable
- Degrades query performance
- Has more complex metadata



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

A snowflake model:

- Results in severe performance degradation because of its greater number of table joins
- Provides a structure that is easier to change as requirements change
- Is quicker at loading data into its smaller normalized tables, compared to loading into a star schema's larger denormalized tables
- Allows using history tables for changing data, rather than level fields (indicators)
- Has a complex metadata structure that is harder for end-user tools to support

Besides the star and snowflake schemas, there are other models that can be considered.

- **Constellation:** A constellation model (also called galaxy model) comprises a series of star models. Constellations are a useful design feature if you have a primary fact table and summary tables of a different dimensionality. It can simplify design by allowing you to share dimensions among many fact tables.
- **Third normal form warehouse:** Some data warehouses consist of a set of relational tables that have been normalized to 3NF. Their data can be directly accessed by using SQL code. They may have more efficient data storage at the price of slower query performance due to extensive table joins. Some large companies build a 3NF central data warehouse, feeding dependent star data marts for specific lines of business.

Lesson Agenda

- Compare and contrast OLTP and data warehouses
- Identify data warehouse objects
- Identify the types of data warehouse schemas
- Review the benefits of using materialized views
- Review the benefits of using query rewrite
- Identifying the Oracle curriculum and documentation resources



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Using Summaries to Improve Performance

- Improved query execution time through precomputing of expensive joins and aggregation operations before execution and storing of results in a database table
- Implemented by a combination of metadata in the data dictionary known as the materialized view (MV) and a container table to contain the output of the defining query of the summary

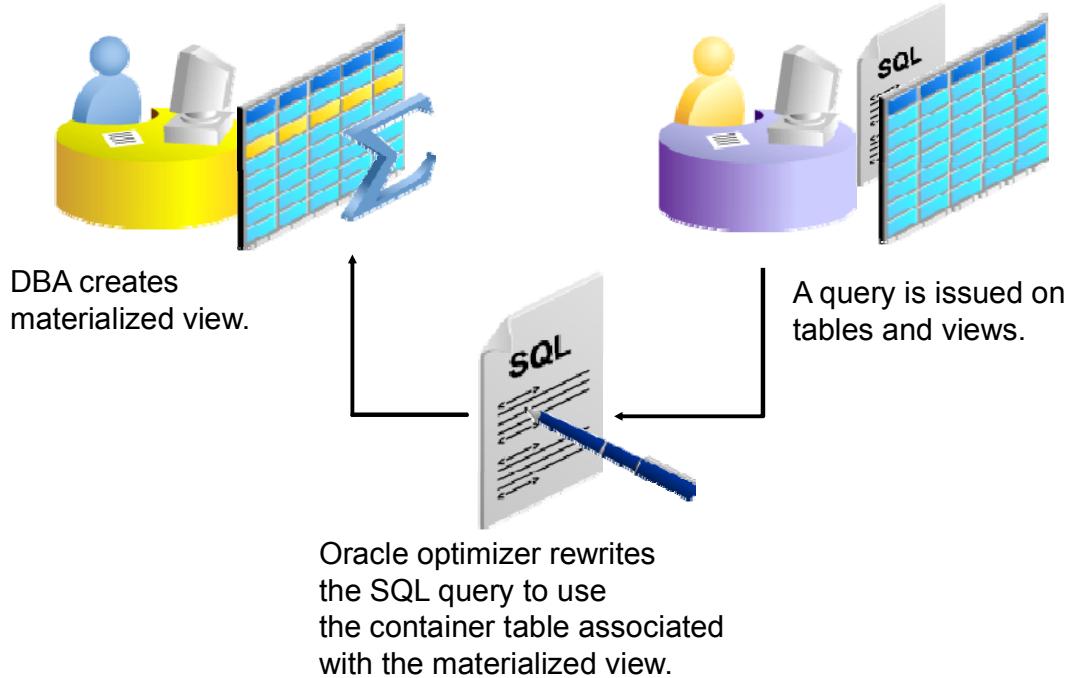


Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Summaries are implemented by using materialized views, whose container tables contain precomputed results of the defining query of the MV. Those results may be precomputed joins only, precomputed aggregates only, or joins with aggregates depending on requirements for performance during refresh.

Note: For additional information, see the Oracle Database documentation or the course titled *Oracle Database: Implement and Administer a Data Warehouse*.

Summary Management: Overview



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In a typical use of summary management, the database administrator creates the materialized view (summary table). When the end user queries tables and views, the Oracle server query rewrite mechanism automatically rewrites the SQL query to use the summary table. The use of the materialized view is transparent to the end user or the application querying the data.

Note: The term “summary” comes from the fact that most of the time users in data warehouse environments compute expensive joins with aggregations. When creating summaries using materialized views in Oracle Database, you are not required to use joins or aggregations.

Summary Management Capabilities

- Mechanisms to define MVs and dimensions
- Query rewrite capability to transparently rewrite a query to use a materialized view
- Enables the use of dimensional hierarchies to facilitate more intelligent query rewrites
- Refresh mechanism to ensure materialized views contain the latest data
- SQL Access Advisor: Recommends materialized views and indexes to be created
- DBMS_ADVISOR.TUNE_MVIEW procedure: Shows you how to make your materialized view fast refreshable and use general query rewrite



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The implementation of summary management in Oracle Database includes the use of the capabilities listed in the slide.

Using Summary Management

- Use the SQL Access Advisor to determine how you use materialized views.
- Create materialized views and design how queries are rewritten.
- Use DBMS_ADVISOR.TUNE_MVIEW to obtain an optimized materialized view as necessary.
- Use EXPLAIN_MVIEW to learn what is possible with an MV (or a potential MV) such as being fast refreshable and what types of query rewrite you can perform with an MV.
- Use EXPLAIN_REWRITE to learn why a query failed to rewrite, or, if it rewrites, which materialized views are used.



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

After your data has been transformed, staged, and loaded into the detail tables, you invoke the summary management process as described in the slide.

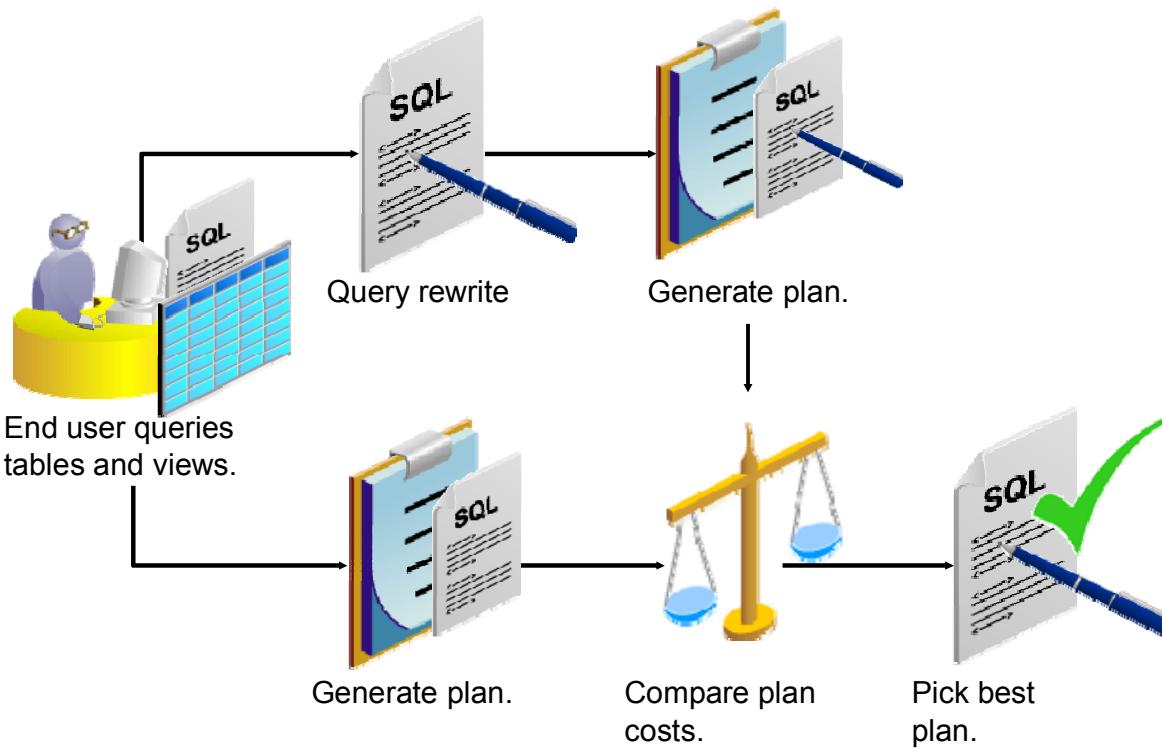
Query Rewrite: Overview

- Tries to use materialized views instead of base tables to return query results
- Can save orders of magnitude of CPU and elapsed time to return results because queries are precomputed
- Queries rewritten even if not in the exact form of the MV
- Various requirements for query rewrite to take place



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The Optimizer Query Rewrite Process



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Oracle optimizes the input query with and without rewrite and selects the least costly alternative. The optimizer rewrites a query by rewriting one or more query blocks, one at a time.

If the rewrite logic has a choice between multiple materialized views to rewrite a query block, it selects the one that can result in reading the least amount of data.

After a materialized view has been picked for a rewrite, the optimizer performs the rewrite and then tests whether the rewritten query can be rewritten further with another materialized view (this could be the case only when nested materialized views exist). This process continues until no further rewrites are possible. Query rewrite is attempted recursively to take advantage of nested materialized views.

The Available Query Rewrite Types

- Text match:
 - Full text match
 - Partial text match
- General rewrite



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Text Match Rewrite

Initially, the query rewrite engine always tries to compare the text of incoming query with the text of the definition of any potential materialized views to rewrite the query. This is because the overhead of doing a simple text comparison is usually negligible compared to the cost of doing a complex analysis required for the general rewrite.

The query rewrite engine uses two text match methods, full text match rewrite and partial text match rewrite.

In full text match, the entire text of a query is compared against the entire text of a materialized view definition (that is, the entire `SELECT` expression), ignoring the white space during text comparison.

When full text match fails, the optimizer then attempts a partial text match. In this method, the text starting from the `FROM` clause of a query is compared against the text starting with the `FROM` clause of a materialized view definition.

What Can Be Rewritten?

- Queries and subqueries in the following types of SQL statements:
 - SELECT
 - CREATE TABLE ... AS SELECT
 - INSERT INTO ... SELECT
- Subqueries in data manipulation language (DML) statements:
 - INSERT
 - UPDATE
 - DELETE
- Subqueries in the set operators:
 - UNION
 - UNION ALL
 - INTERSECT
 - MINUS



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Query rewrite operates on queries and subqueries in the following types of SQL statements:

- SELECT
- CREATE TABLE ... AS SELECT
- INSERT INTO ... SELECT

Query rewrite operates on subqueries in DML statements: INSERT, DELETE, and UPDATE. It also operates on subqueries in set operators: UNION, UNION ALL, INTERSECT, and MINUS.

Requirements for Query Rewrite

- User running the statement must have the query rewrite privilege.
- The MV that you use for the rewrite must have rewrite enabled.
- The session of the query must have the query rewrite enabled.
- The session query rewrite integrity must not be violated in respect to “stale” data in the MV’s container table.
- The optimizer must decide if the query rewrite is possible based on the text of the statement, the availability of MVs, dimensions, or both.



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

A query is rewritten only when a certain number of conditions are met:

- Query rewrite must be enabled for the session.
- A materialized view must be enabled for query rewrite.
- The rewrite integrity level should allow the use of the materialized view. For example, if a materialized view is not fresh and query rewrite integrity is set to ENFORCED, then the materialized view is not used.
- All or part of the results requested by the query must be obtainable from the precomputed result stored in the materialized view or views.

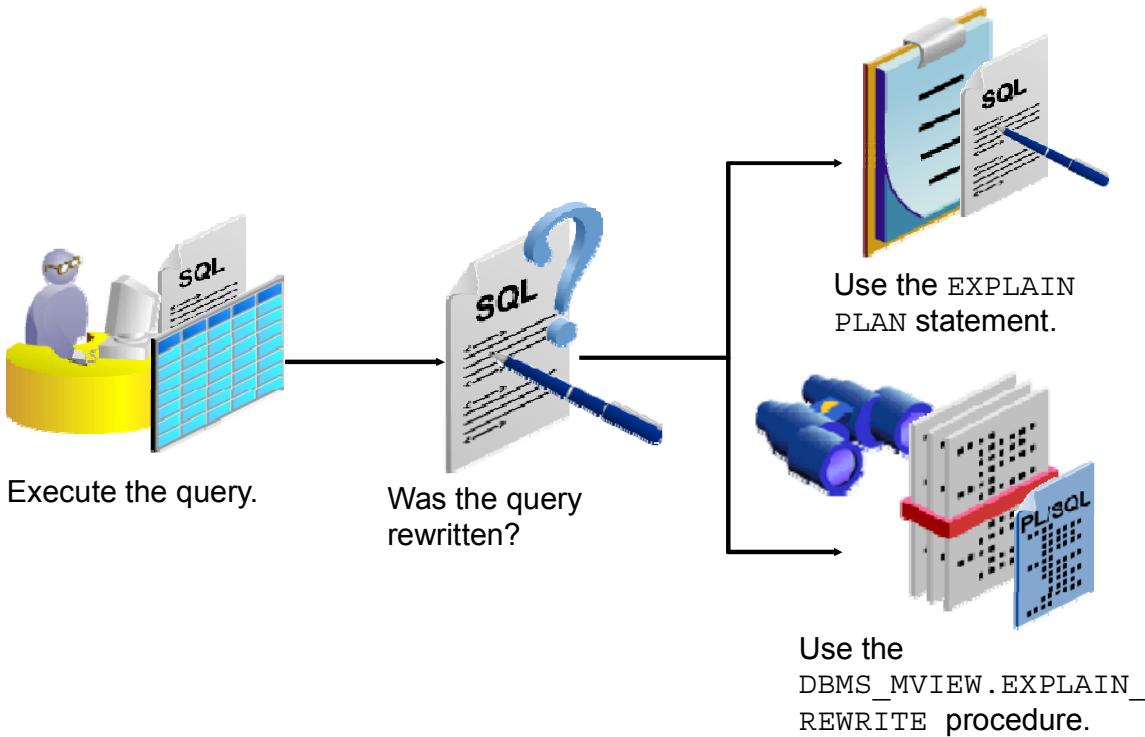
To test these conditions, the optimizer may depend on some of the data relationships declared by the user using constraints and dimensions, among others, hierarchies, referential integrity, and uniqueness of key data, and so on.

You must follow several conditions to enable query rewrite:

1. Individual materialized views must have the `ENABLE QUERY REWRITE` clause.
2. The session parameter `QUERY_REWRITE_ENABLED` must be set to `TRUE` (the default) or `FORCE`.
3. Cost-based optimization must be used by setting the initialization parameter `OPTIMIZER_MODE` to `ALL_ROWS`, `FIRST_ROWS`, or `FIRST_ROWS_n`.

If step 1 has not been completed, a materialized view is not eligible for query rewrite.

Verifying Whether Query Rewrite Occurred



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Because query rewrite occurs transparently, it is not always evident that it has taken place. The rewritten statement is not stored in the V\$SQL view and cannot be dumped in a trace file. If the query runs faster, rewrite should have occurred; but there is no proof to confirm this. There are two ways to confirm that the query rewrite has occurred:

- Use the EXPLAIN PLAN statement and check whether the OBJECT_NAME column contains the name of a materialized view.
- Use the DBMS_MVIEW.EXPLAIN_REWRITE procedure to see whether a query is rewritten or not.

Note: For additional information about verifying whether or not the query was rewritten using the EXPLAIN PLAN statement and the DBMS_MVIEW.EXPLAIN_REWRITE procedure, see the course titled *Oracle Database: Implement and Administer a Data Warehouse* or the *Oracle Database Data Warehousing Guide* documentation.

Analyzing Materialized View Capabilities: The DBMS_MVIEW.EXPLAIN_MVIEW Procedure

You can use this procedure to learn what is possible with a materialized view or potential materialized view. In particular, this procedure enables you to determine:

- Whether a materialized view is fast refreshable
- The types of query rewrite that you can perform with this materialized view
- Whether Partition Change Tracking (PCT) refresh is possible



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

For additional information on materialized views and the DBMS_MVIEW.EXPLAIN_MVIEW procedure, see the *Oracle Database Data Warehousing Guide*.

Summary

In this lesson, you should have learned how to review:

- Benefits of a data warehouse
- Schema types in a data warehouse
- Types of objects used in a data warehouse
- Summaries to improve performance
- Benefits of using materialized views
- Benefits of query rewrite



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

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

b

Table Descriptions

ORACLE®

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Schema Descriptions

Overall Description

Oracle Database sample schemas portray a sample company that operates worldwide to fill orders for several different products. The company has three divisions:

- **Human Resources:** Tracks information about the employees and facilities
- **Order Entry:** Tracks product inventories and sales through various channels
- **Sales History:** Tracks business statistics to facilitate business decisions

Each of these divisions is represented by a schema. In this course, you have access to the objects in all the schemas. However, the emphasis of the examples, demonstrations, and practices is on the Sales History (SH) schema.

Human Resources (HR)

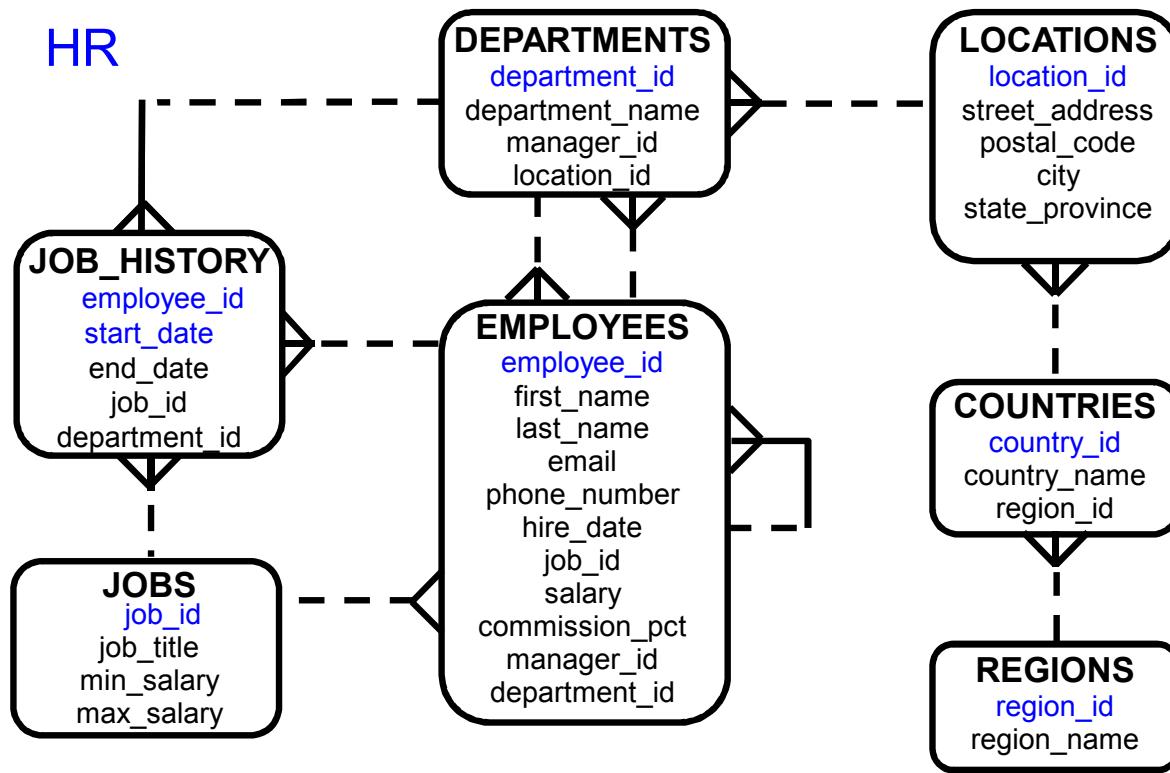
In the HR records, each employee has an identification number, email address, job identification code, salary, and manager. Some employees earn commissions in addition to their salary.

The company also tracks information about 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 resigns, information about the duration for which the employee worked, the job identification number, and the department name are recorded.

The sample company is regionally diverse, so it tracks the locations of its warehouses and departments. Each employee is assigned to a department, and each department is identified by a unique department number or a short name. Each department is associated with one location, and each location has a full address that includes the street name, postal code, city, state or province, and the country code.

In places where the departments and warehouses are located, the company records details such as the country name, currency symbol, currency name, and the geographical region where the country exists.

The HR Entity Relationship Diagram



Human Resources (HR) Row Counts

```
SELECT COUNT(*) FROM regions;  
COUNT(*)
```

```
-----  
4
```

```
SELECT COUNT(*) FROM countries;  
COUNT(*)
```

```
-----  
25
```

```
SELECT COUNT(*) FROM locations;  
COUNT(*)
```

```
-----  
23
```

```
SELECT COUNT(*) FROM departments;  
COUNT(*)
```

```
-----  
27
```

```
SELECT COUNT(*) FROM jobs;  
COUNT(*)
```

```
-----  
19
```

```
SELECT COUNT(*) FROM employees;  
COUNT(*)
```

```
-----  
107
```

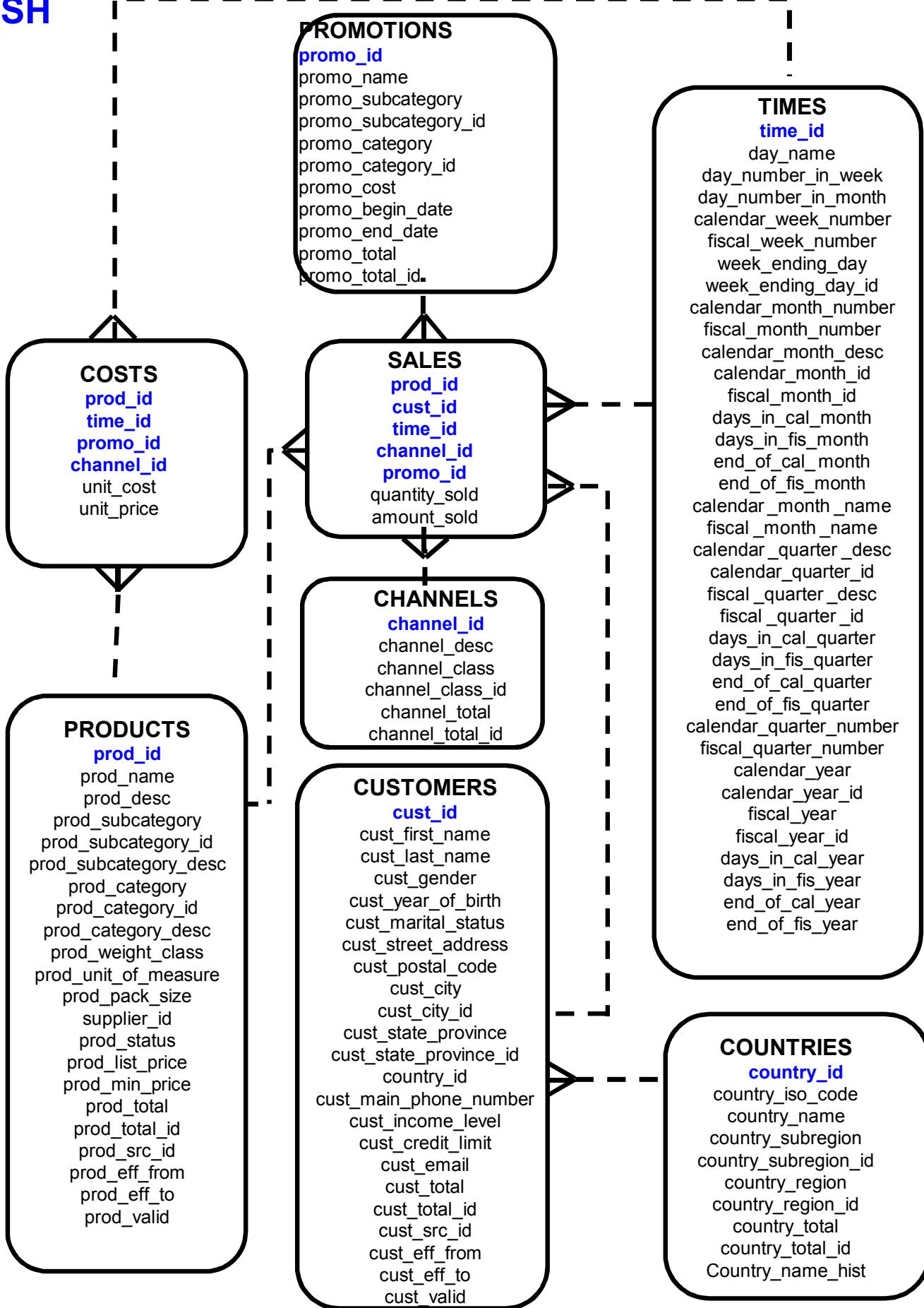
```
SELECT COUNT(*) FROM job_history;  
COUNT(*)
```

```
-----  
10
```

Sales History (SH)

The company does a high volume of business, so it runs business statistics reports to aid in decision support. Many of these reports are time-based and nonvolatile. That is, they analyze past data trends. The company loads data into its data warehouse regularly to gather statistics for the reports. The reports include annual, quarterly, monthly, and weekly sales figures by product.

The company also runs reports on the distribution channels through which its sales are delivered. When the company runs special promotions on its products, it analyzes the impact of the promotions on sales. It also analyzes sales by geographical area.

Schema Descriptions**SH**

Sales History (SH) Row Counts

```
SELECT COUNT(*) FROM channels;  
COUNT(*)
```

```
-----  
5
```

```
SELECT COUNT(*) FROM costs;  
COUNT(*)
```

```
-----  
82112
```

```
SELECT COUNT(*) FROM countries;  
COUNT(*)
```

```
-----  
23
```

```
SELECT COUNT(*) FROM customers;  
COUNT(*)
```

```
-----  
55500
```

```
SELECT COUNT(*) FROM products;  
COUNT(*)
```

```
-----  
72
```

```
SELECT COUNT(*) FROM promotions;  
COUNT(*)
```

```
-----  
503
```

```
SELECT COUNT(*) FROM sales;  
COUNT(*)
```

```
-----  
918843
```

```
SELECT COUNT(*) FROM times;  
COUNT(*)
```

```
-----  
1826
```

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.

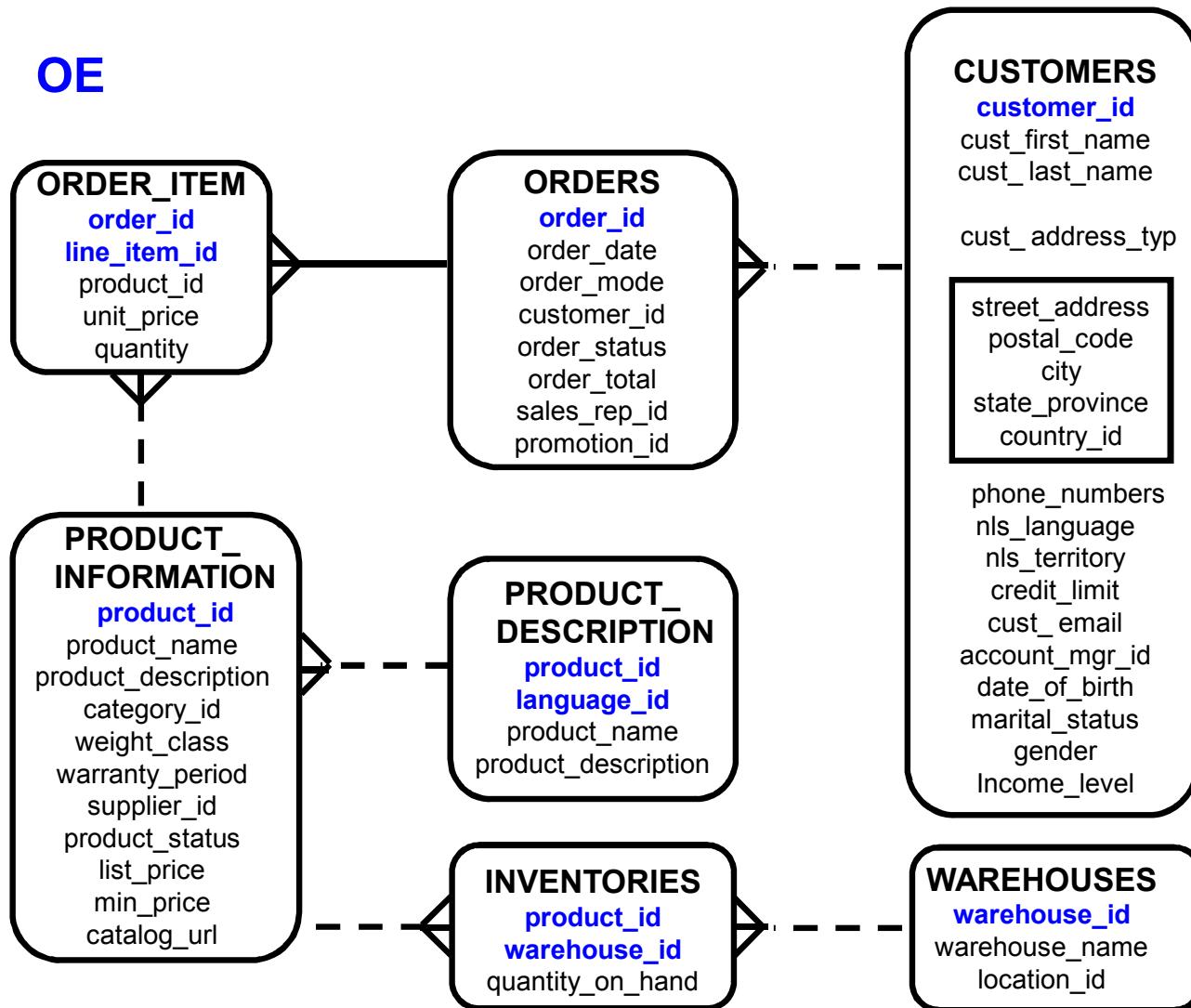
The company maintains warehouses in several locations to facilitate filling customer orders. Each warehouse has a warehouse identification number, name, and location identification number.

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 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. A customer's phone number is also tracked. The language and territory of each customer are identified, taking into consideration language differences.

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 or may not be the same individual as the account manager for a customer, or in the case of an order over the Internet, the sales representative is not recorded. In addition to the order information, the number of items ordered, the unit price, and the products ordered are also tracked.

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.

Order Entry (OE)



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
```

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 © 2013, Oracle and/or its affiliates. All rights reserved.

Objectives

After completing this appendix, you should be able to:

- 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

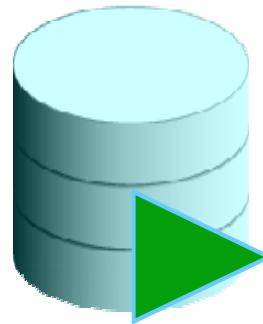


Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

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 © 2013, 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 onto 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 version 9.2.0.1 and later



Copyright © 2013, 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.

Notes

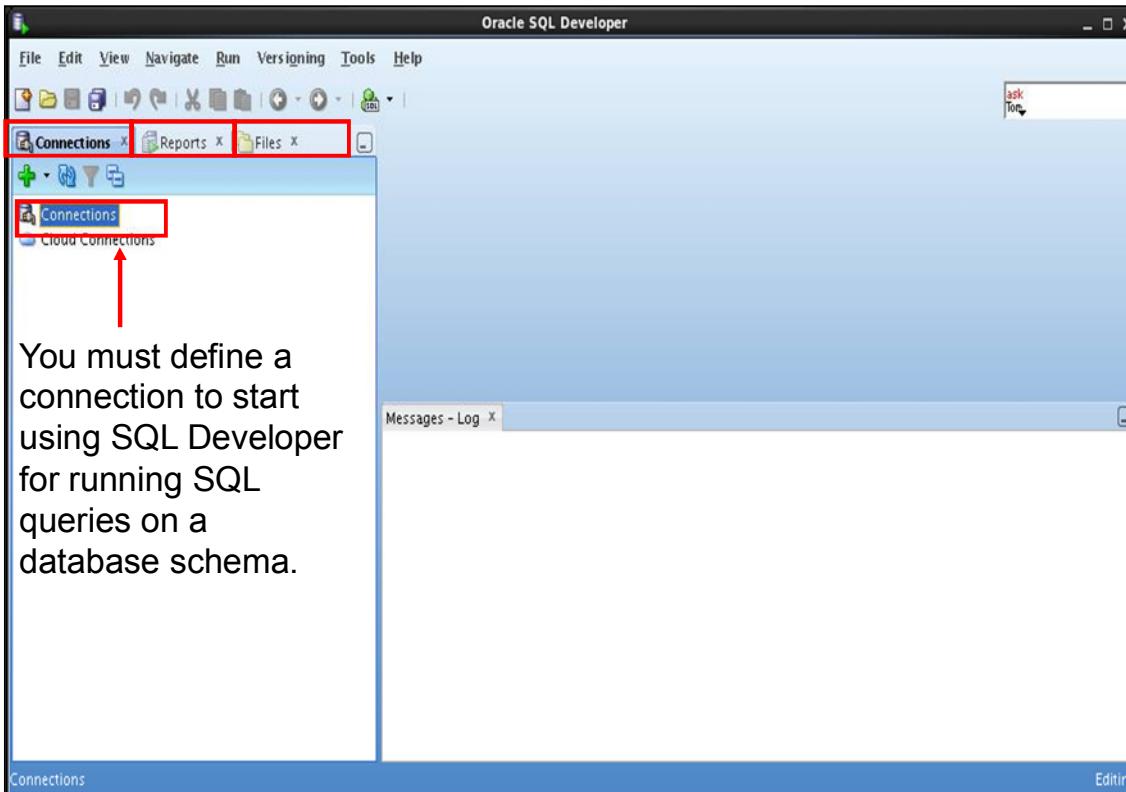
For Oracle Database 12c Release 1, you will have to download and install SQL Developer. SQL Developer is freely downloadable from the following link:

<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 3.2 Interface



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

ORACLE

The SQL Developer interface contains three main navigation tabs, from left to right:

- **Connections tab:** By using this tab, you can browse database objects and users to which you have access.
- **Reports tab:** Identified by the Reports icon, this tab enables you to run predefined reports or create and add your own reports.
- **Files tab:** 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.

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 and functions.

Menus

The following menus contain standard entries, plus entries for features that are specific to SQL Developer:

- **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
- **Versioning:** Provides integrated support for the following versioning and source control systems – Concurrent Versions System (CVS) 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.

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
 - Multiple 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 © 2013, 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.

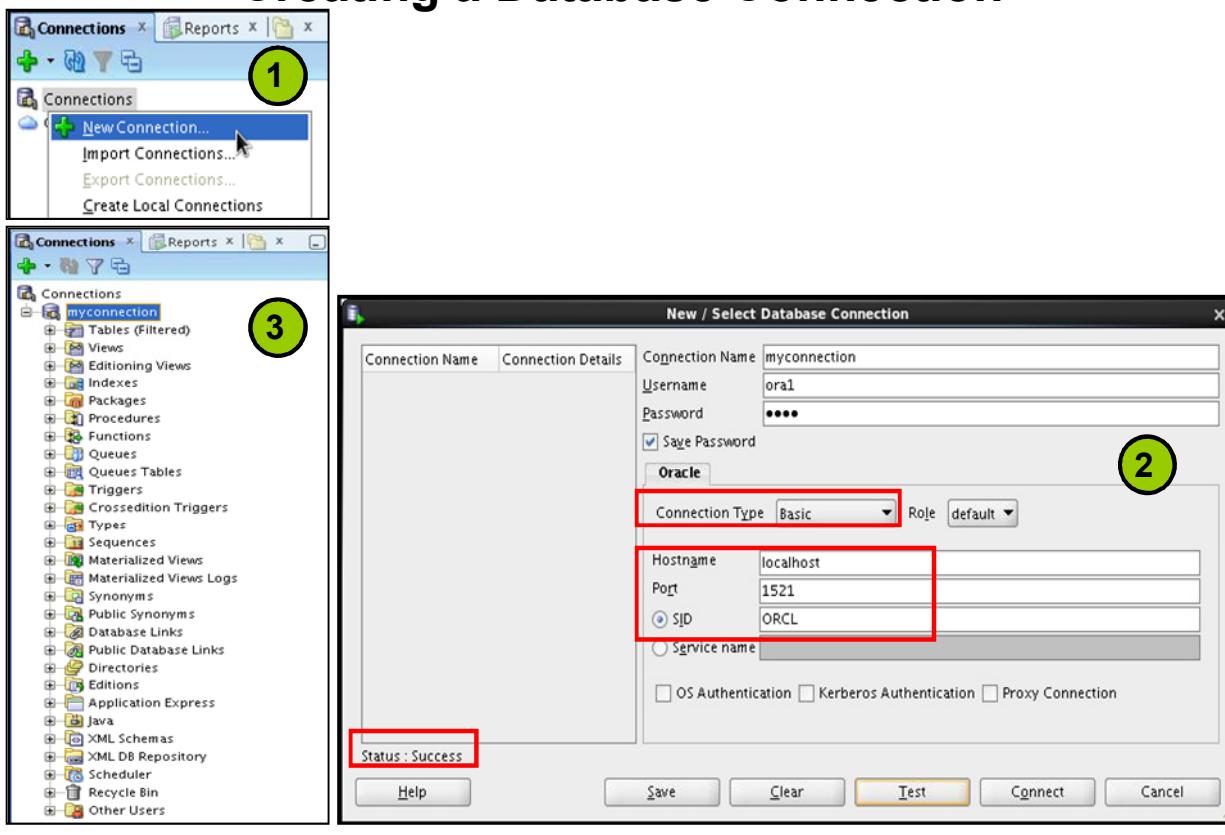
By default, the `tnsnames.ora` file is located in the `$ORACLE_HOME/network/admin` directory, but it can also be in the directory specified by the `TNS_ADMIN` environment variable or registry value. When you start SQL Developer and open the Database Connections dialog box, SQL Developer automatically 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 © 2013, 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 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.

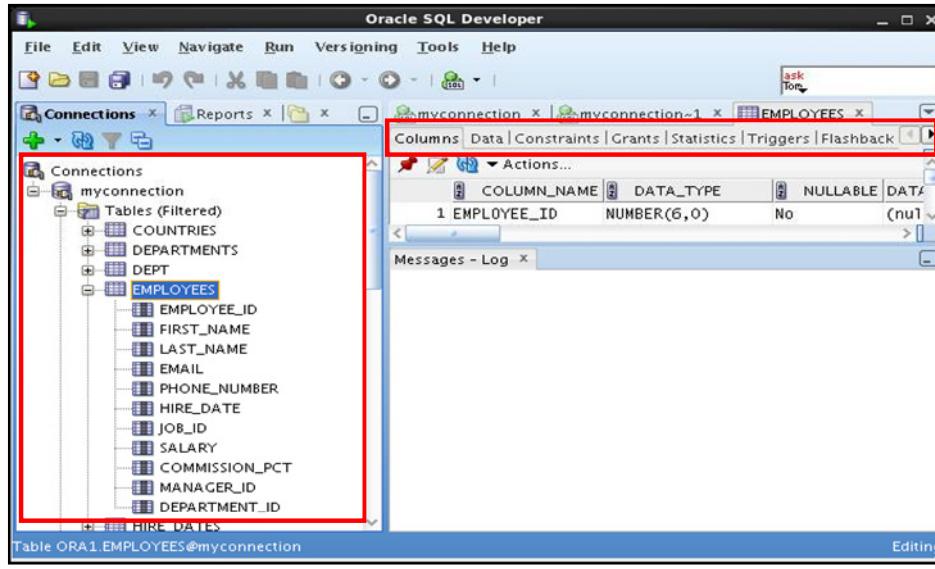
3. The connection gets added in the Connections Navigator. You can expand the connection to view the database objects and view object definitions (dependencies, details, statistics, and so on).

Note: From the same New>Select Database Connection window, you can define connections to non-Oracle data sources 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



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

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.

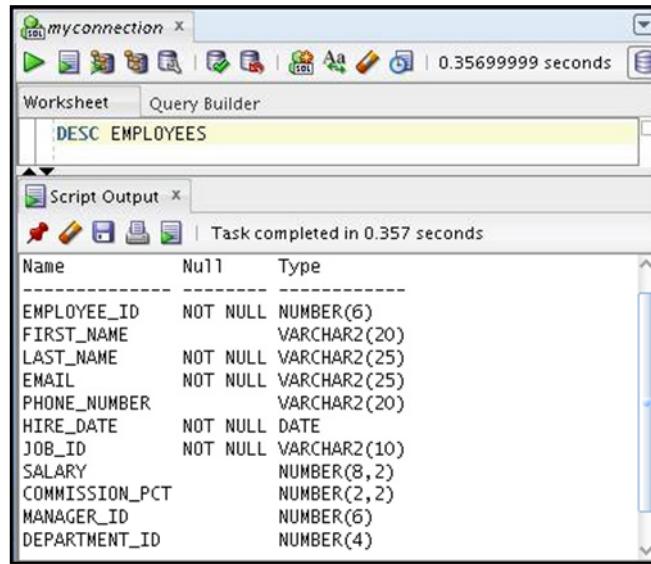
You can see the definition of the objects broken into tabs of information that is pulled out of the data dictionary. For example, if you select a table in the Navigator, details about columns, constraints, grants, statistics, triggers, and so on are displayed on an easy-to-read tabbed page.

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, there's a connection named "myconnection". Below it, the "Worksheet" tab is active, showing the command "DESC EMPLOYEES" in the text area. To the right of the worksheet is a "Script Output" window which displays the results of the command. The results are presented as a table with three columns: Name, Null, and Type. The table lists the columns of the EMPLOYEES table, their constraints (whether they can be null), and their data types.

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)

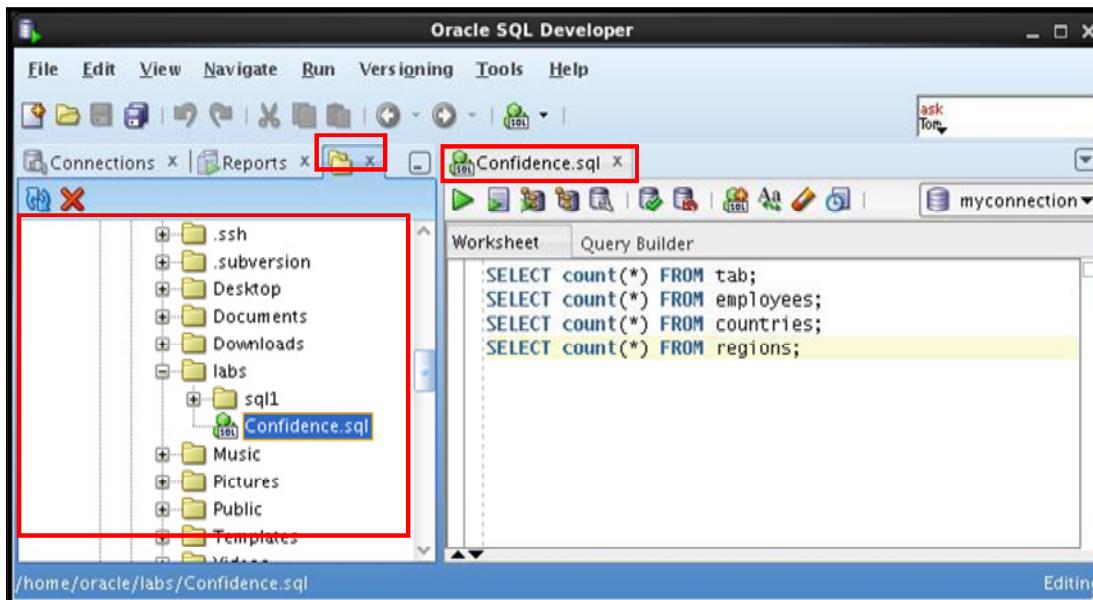
ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In SQL Developer, you can also display the structure of a table 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.



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

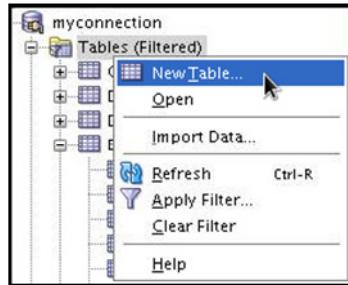
Browsing Database Objects

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.

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

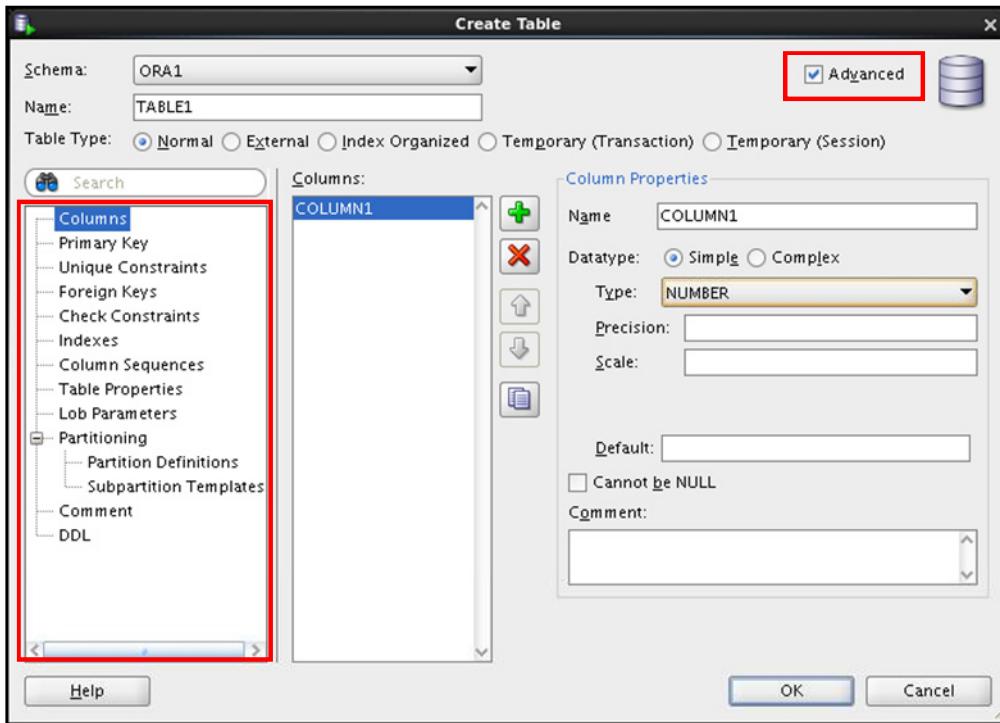
Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

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 © 2013, 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.

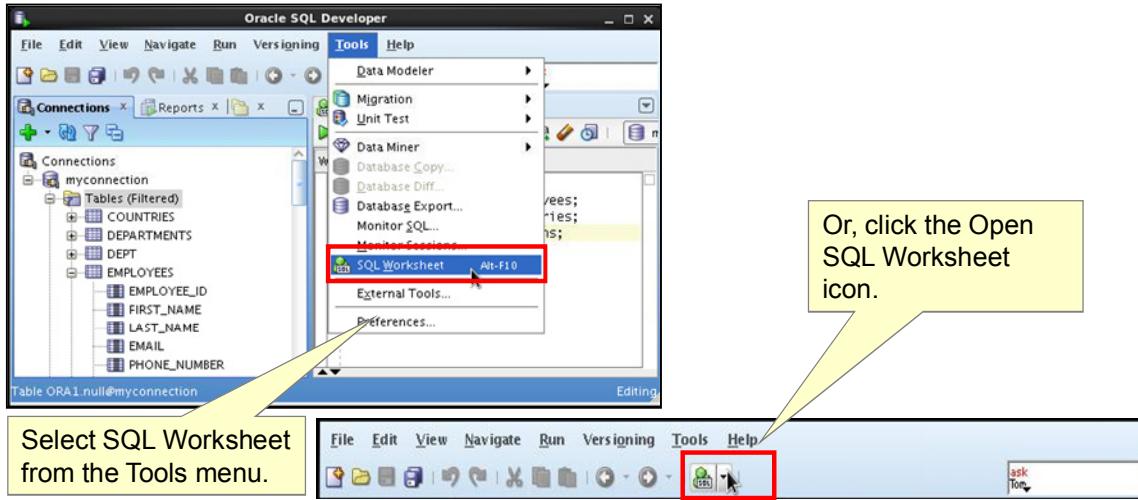
To create a new table, perform the following steps:

1. In the Connections Navigator, right-click Tables and select Create 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.



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

ORACLE

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.

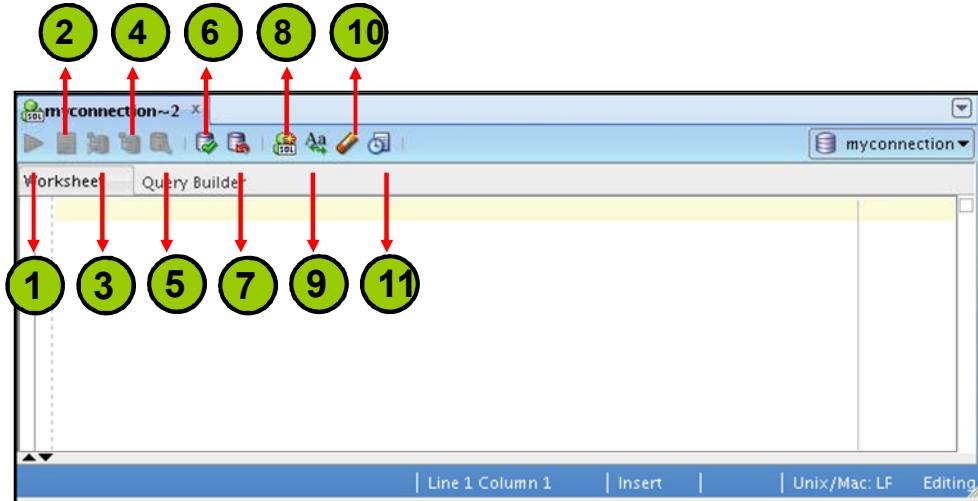
You can specify the actions that can be processed by the database connection associated with the worksheet, such as:

- 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



The Oracle logo, consisting of the word "ORACLE" in a white sans-serif font inside a red horizontal bar.

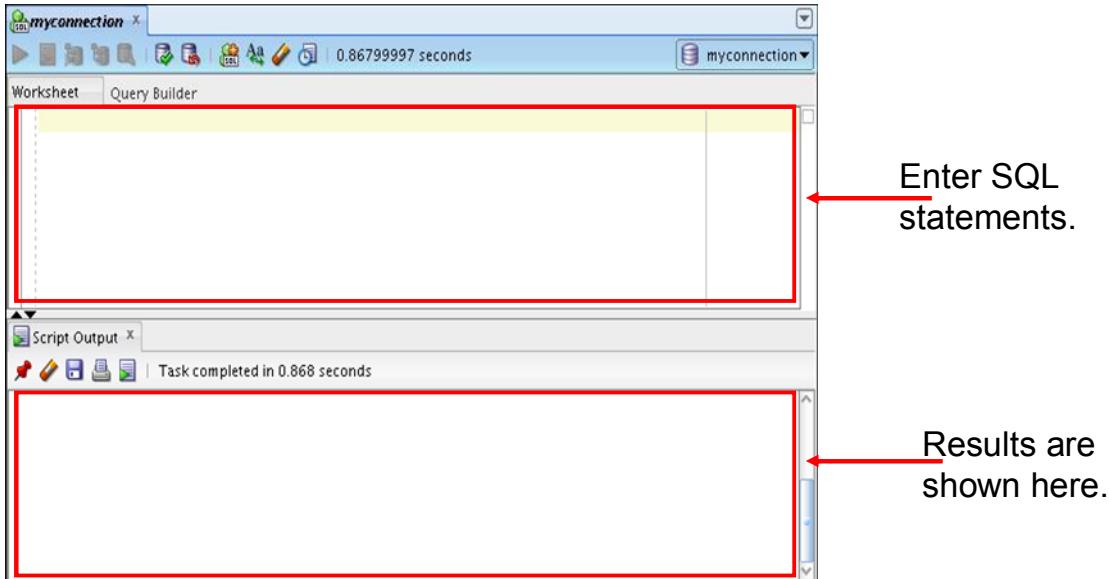
Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

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 © 2013, 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.

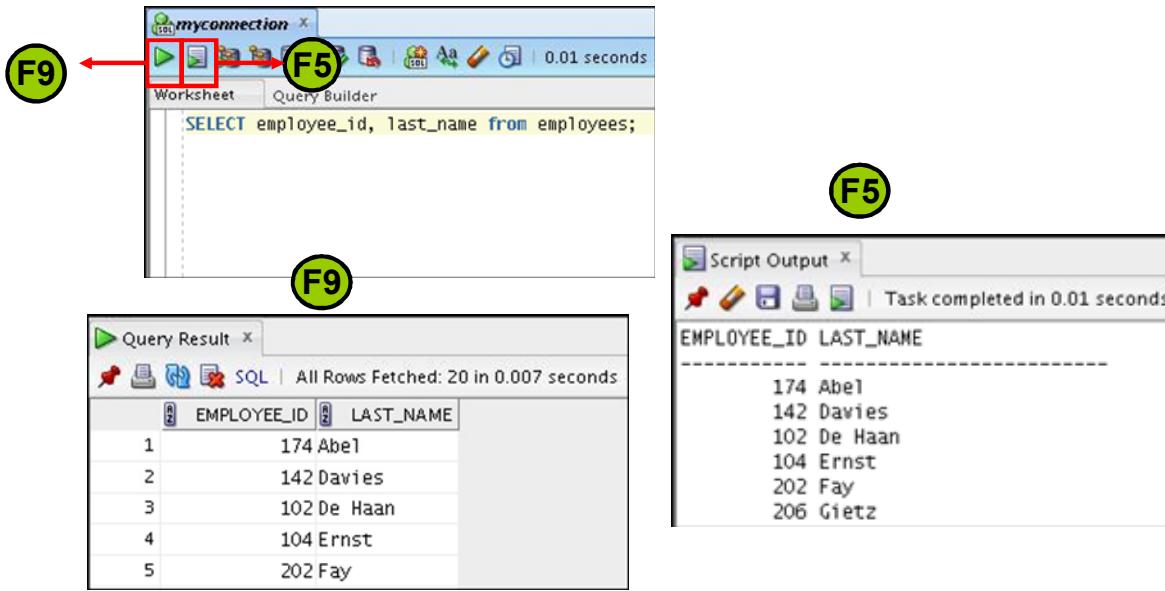
The SQL Worksheet currently supports a number of SQL*Plus commands. Commands that are not supported by the SQL Worksheet are ignored and not sent to the Oracle database. Through the SQL Worksheet, you can execute the SQL statements and some of the SQL*Plus commands.

You can display a SQL Worksheet by performing either of the following actions:

- Select Tools > SQL Worksheet.
- Click the Open SQL Worksheet icon.

Executing SQL Statements

Use the Enter SQL Statement box to enter single or multiple SQL statements.

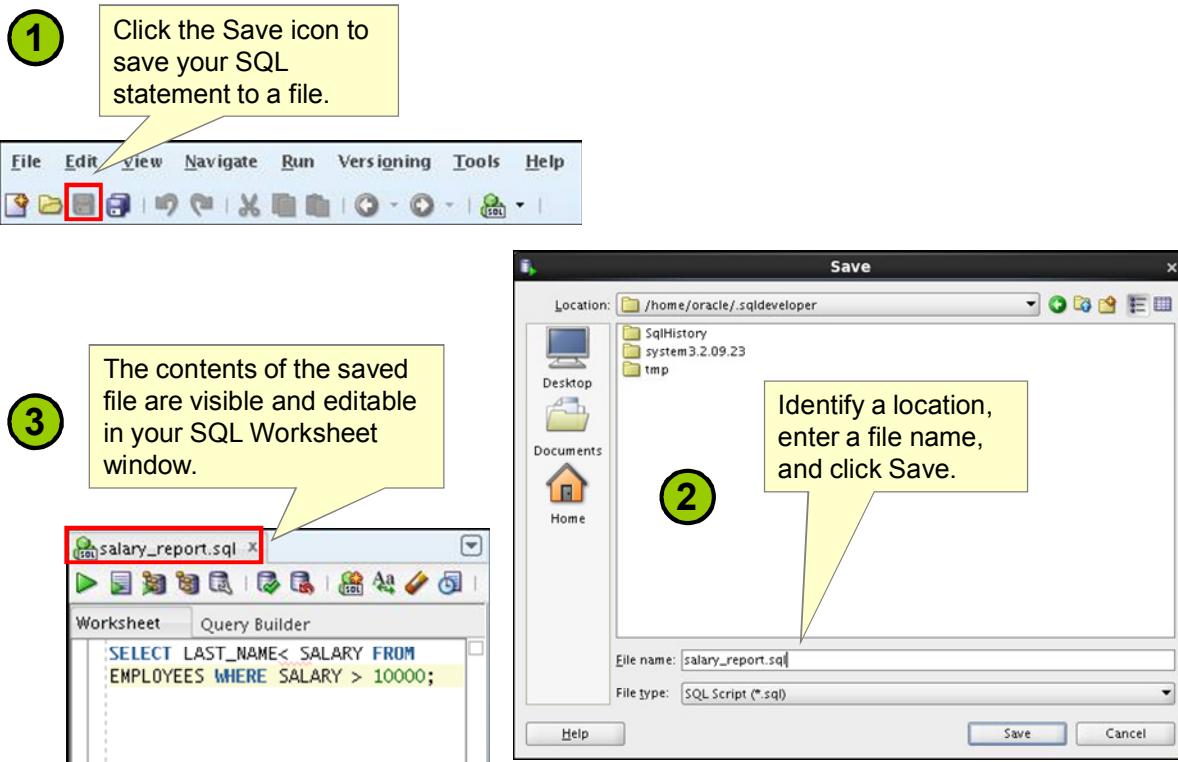


ORACLE

Copyright © 2013, 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 © 2013, 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:

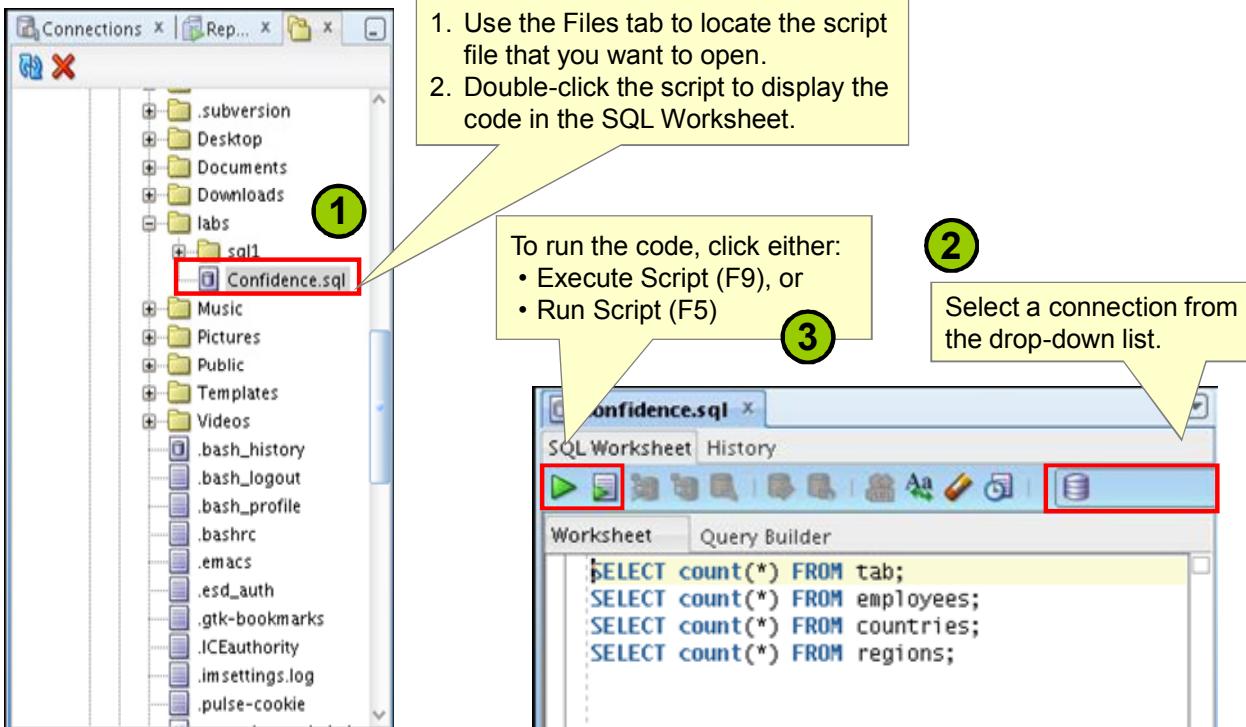
1. Click the Save icon or use the File > Save menu item.
2. In the Save dialog box, enter a file name and the location where you want the file saved.
3. Click Save.

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 © 2013, Oracle and/or its affiliates. All rights reserved.

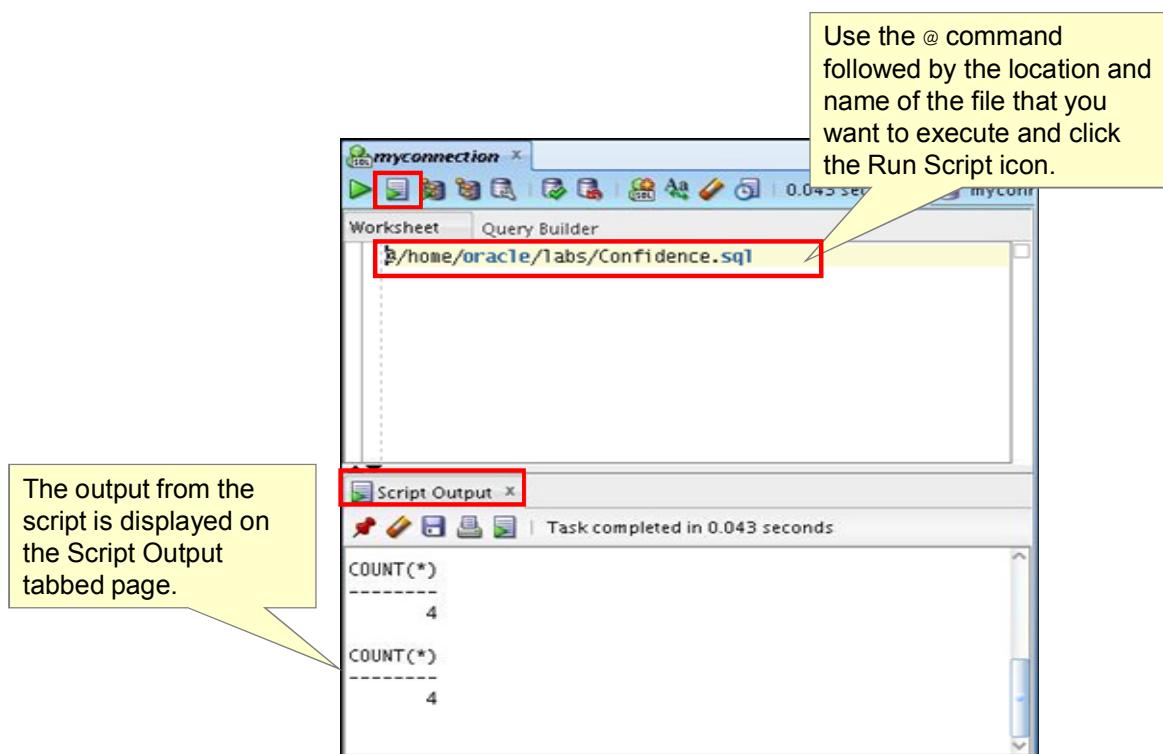
To open a script file and display the code in the SQL Worksheet area, perform the following steps:

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



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

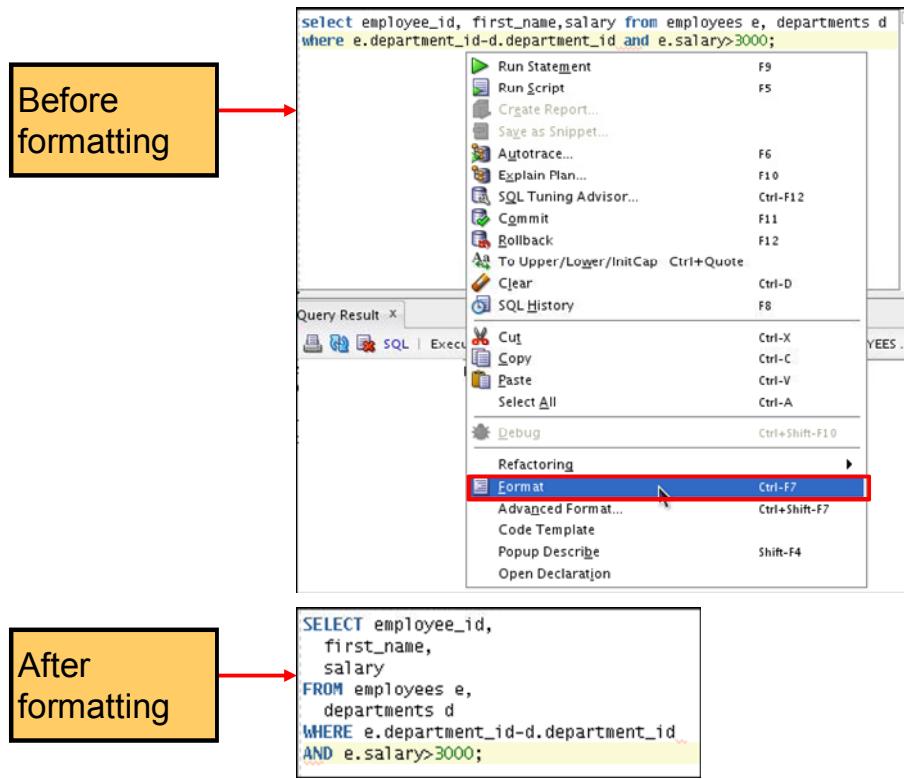
ORACLE

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 © 2013, Oracle and/or its affiliates. All rights reserved.

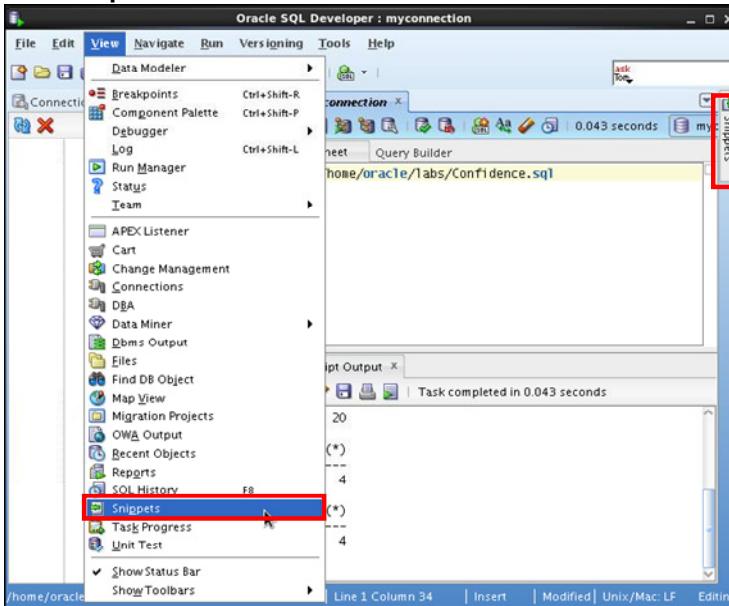
You may want to format the indentation, spacing, capitalization, and line separation of the SQL code. SQL Developer has a feature for formatting SQL code.

To format the SQL code, right-click in the statement area and select Format.

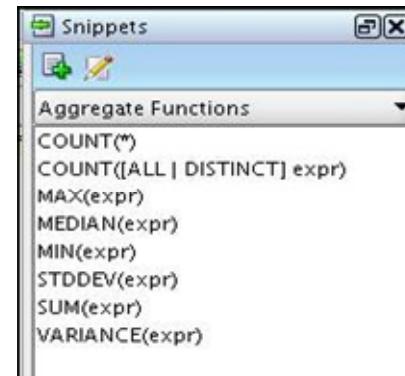
In the example in the slide, before formatting, the SQL code has the keywords not capitalized and the statement not properly indented. After formatting, the SQL code is beautified with the keywords capitalized and the statement properly indented.

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

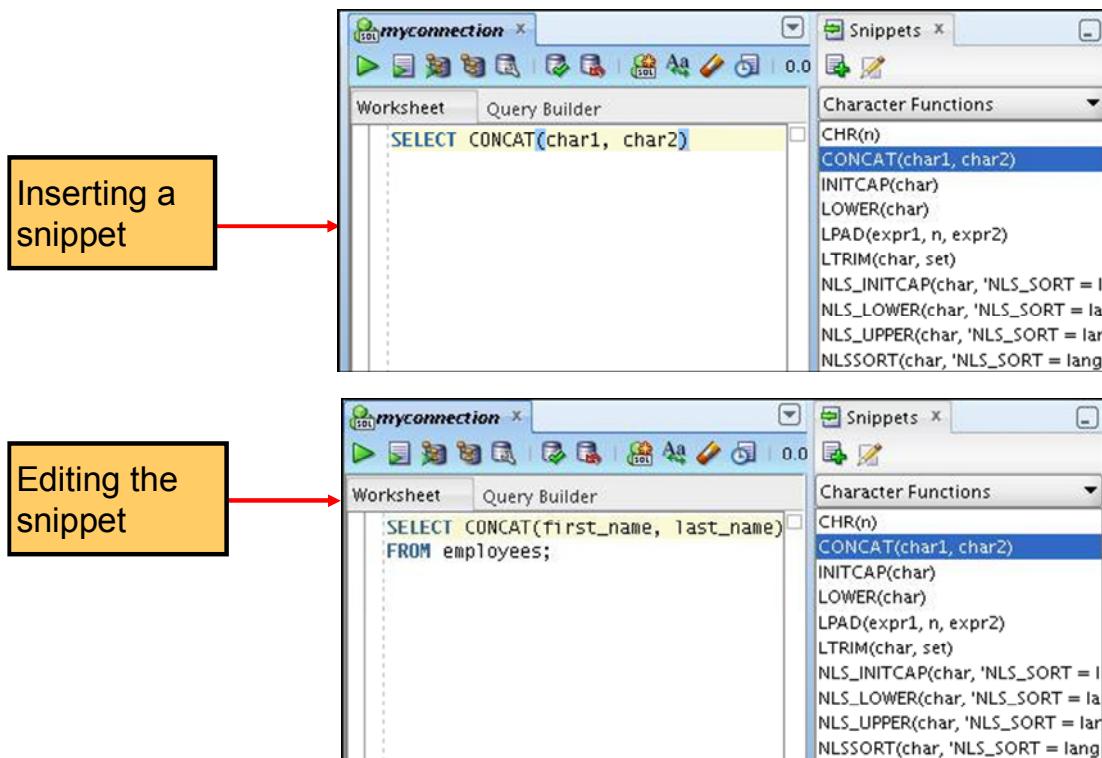
Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

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 a feature called Snippets. Snippets are code fragments such as SQL functions, optimizer hints, and miscellaneous PL/SQL programming techniques. You can drag snippets to the Editor window.

To display Snippets, select View > Snippets.

The Snippets window is displayed on 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 © 2013, Oracle and/or its affiliates. All rights reserved.

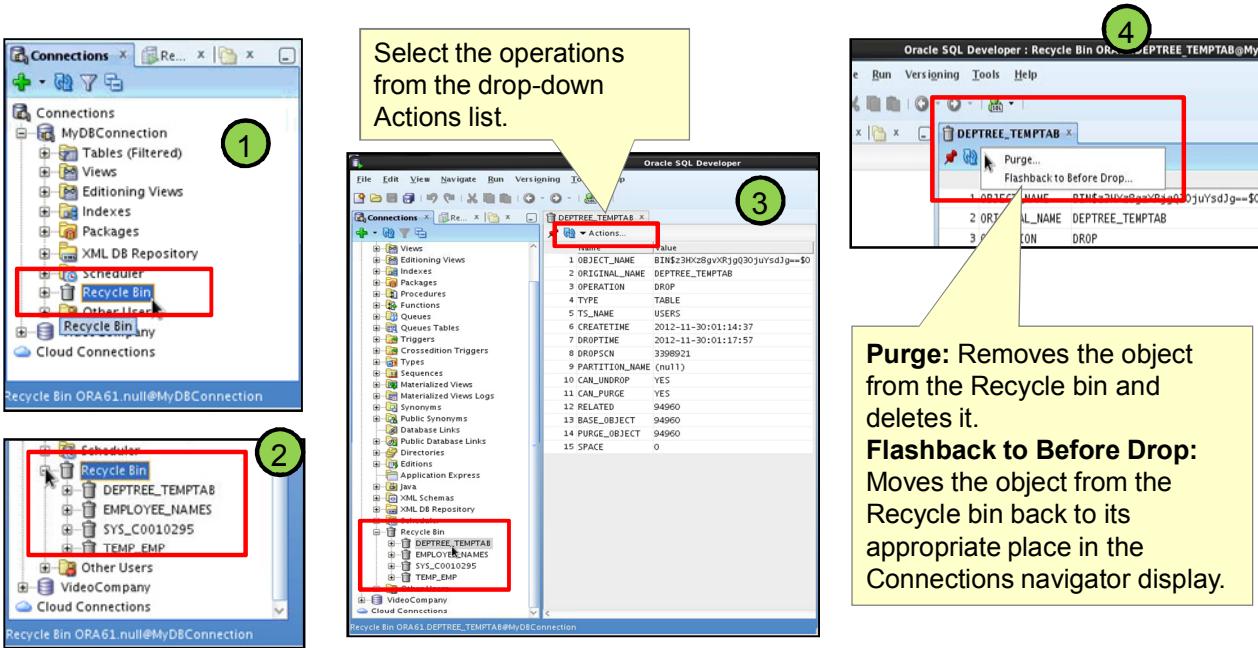
To insert a Snippet into your code in a SQL Worksheet or in a PL/SQL function or procedure, drag the snippet from the Snippets window to the desired place in your code. Then you can edit the syntax so that the SQL function is valid in the current context. To see a brief description of a SQL function in a tool tip, 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 © 2013, Oracle and/or its affiliates. All rights reserved.

ORACLE

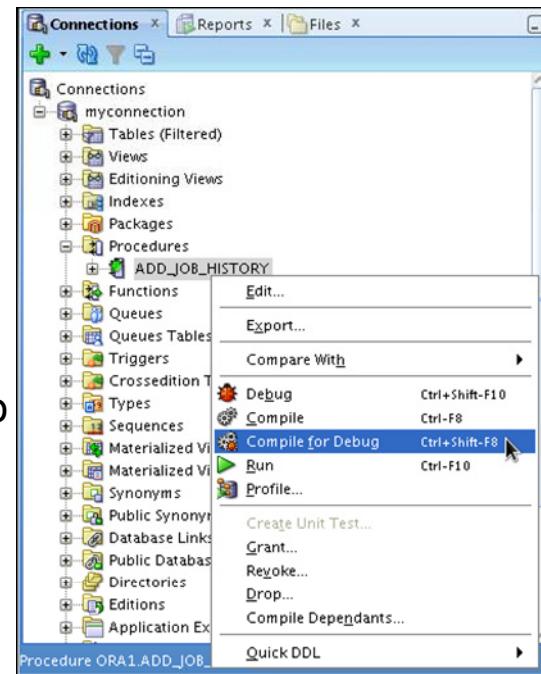
The recycle bin is a data dictionary table containing information about dropped objects. Dropped tables and any associated objects such as indexes, constraints, nested tables, and the likes are not removed and still occupy space. They continue to count against user space quotas, until specifically purged 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) the 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 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, step over tasks.



ORACLE

Copyright © 2013, 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 tab of the output window.

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
APEX_040100	APEX	PROCEDURE	APEX_040100	WW_FLOW	PACKAGE
APEX_040100	APEX	PROCEDURE	APEX_040100	WW_FLOW_ISC	PACKAGE
APEX_040100	APEX	PROCEDURE	APEX_040100	WW_FLOW_SECURITY	PACKAGE
APEX_040100	APEX	PROCEDURE	SYS	STANDARD	PACKAGE
APEX_040100	APEX	PROCEDURE	SYS	SYS_STUB_FOR_PURITY_ANALYSIS	PACKAGE
APEX_040100	APEXWS	PACKAGE	SYS	STANDARD	PACKAGE
APEX_040100	APEXADMIN	PROCEDURE	APEX_040100	F	PROCEDURE
APEX_040100	APEX_ADMIN	PROCEDURE	SYS	STANDARD	PACKAGE
APEX_040100	APEX_APPLICATIONS	VIEW	APEX_040100	NV	FUNCTION
APEX_040100	APEX_APPLICATIONS	VIEW	APEX_040100	WW_FLOWS	TABLE
APEX_040100	APEX_APPLICATIONS	VIEW	APEX_040100	WW_FLOW_APPLICATION_GROUPS	TABLE
APEX_040100	APEX_APPLICATIONS	VIEW	APEX_040100	WW_FLOW_AUTHENTIFICATIONS	TABLE
APEX_040100	APEX_APPLICATIONS	VIEW	APEX_040100	WW_FLOW_COMPANIES	TABLE
APEX_040100	APEX_APPLICATIONS	VIEW	APEX_040100	WW_FLOW_COMPANY_SCHEMAS	TABLE
APEX_040100	APEX_APPLICATIONS	VIEW	APEX_040100	WW_FLOW_COMPUTATIONS	TABLE
APEX_040100	APEX_APPLICATIONS	VIEW	APEX_040100	WW_FLOW_ICON_BAR	TABLE
APEX_040100	APEX_APPLICATIONS	VIEW	APEX_040100	WW_FLOW_INSTALL_SCRIPTS	TABLE
APEX_040100	APEX_APPLICATIONS	VIEW	APEX_040100	WW_FLOW_ITEMS	TABLE
APEX_040100	APEX_APPLICATIONS	VIEW	APEX_040100	WW_FLOW_LANGUAGE_MAP	TABLE

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

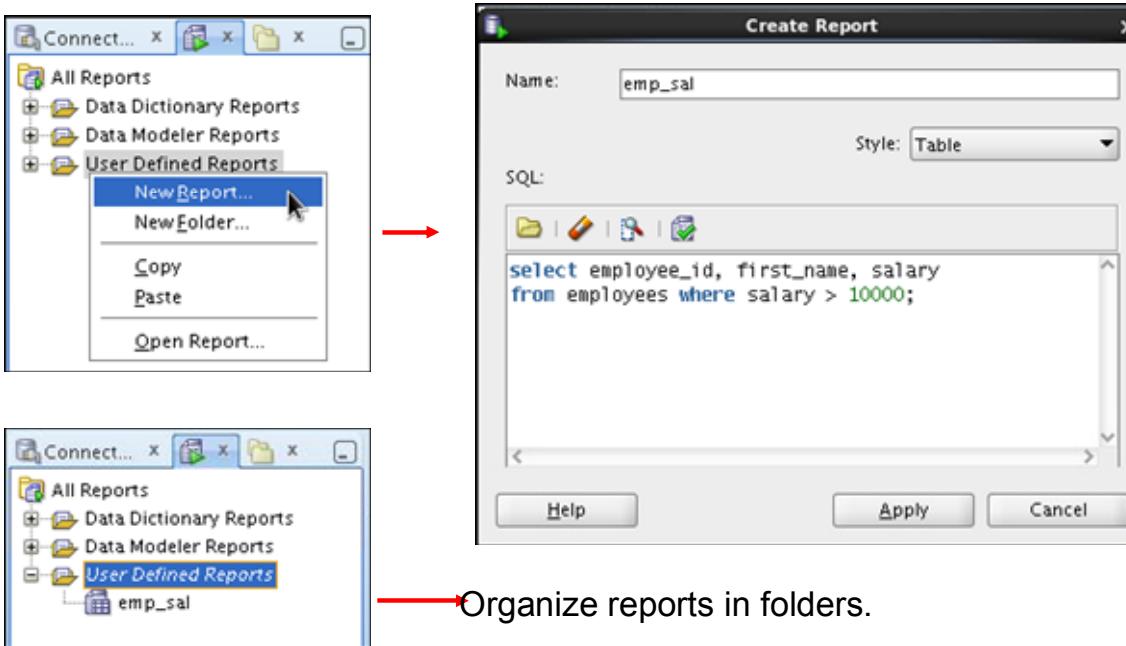
SQL Developer provides many reports about the database and its objects. These reports can be grouped into the following categories:

- 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 on the left of the window. Individual reports are displayed in tabbed panes on 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 © 2013, Oracle and/or its affiliates. All rights reserved.

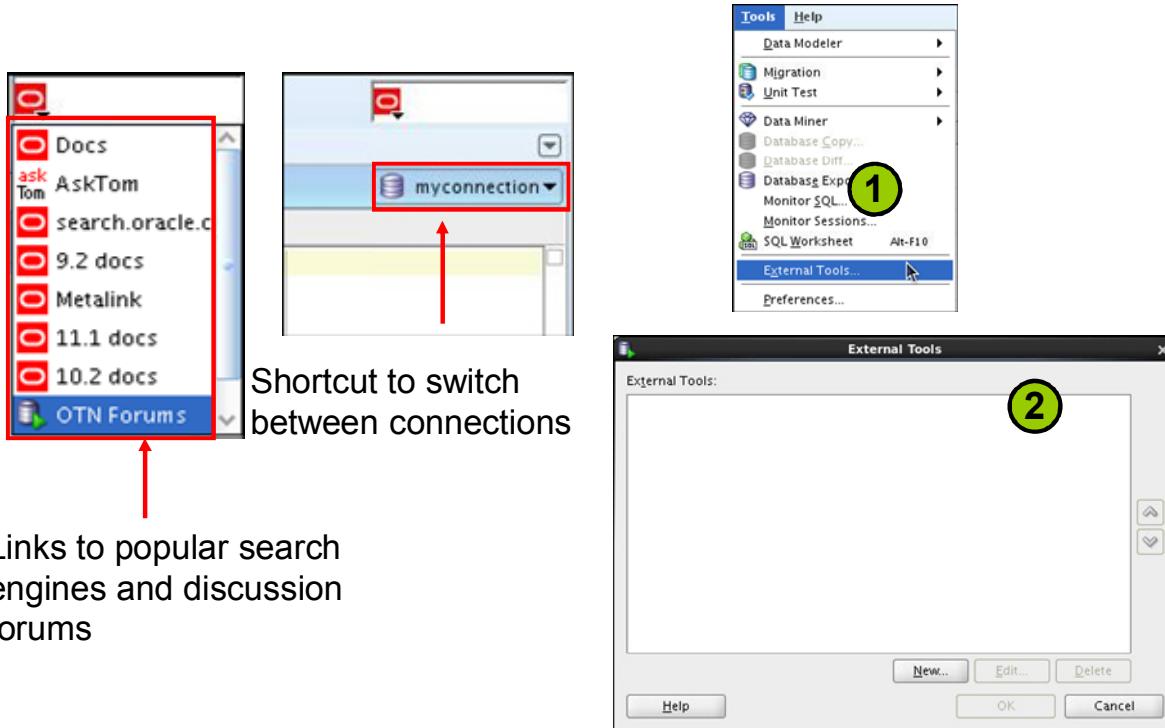
User-defined reports are reports created by SQL Developer users. To create a user-defined report, perform the following steps:

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 tool tip to be displayed when the cursor stays briefly over the report name in the Reports navigator display.

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 Add Folder. Information about user-defined reports, including any folders for these reports, is stored in a file named `UserReports.xml` in the directory for user-specific information.

Search Engines and External Tools



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

ORACLE

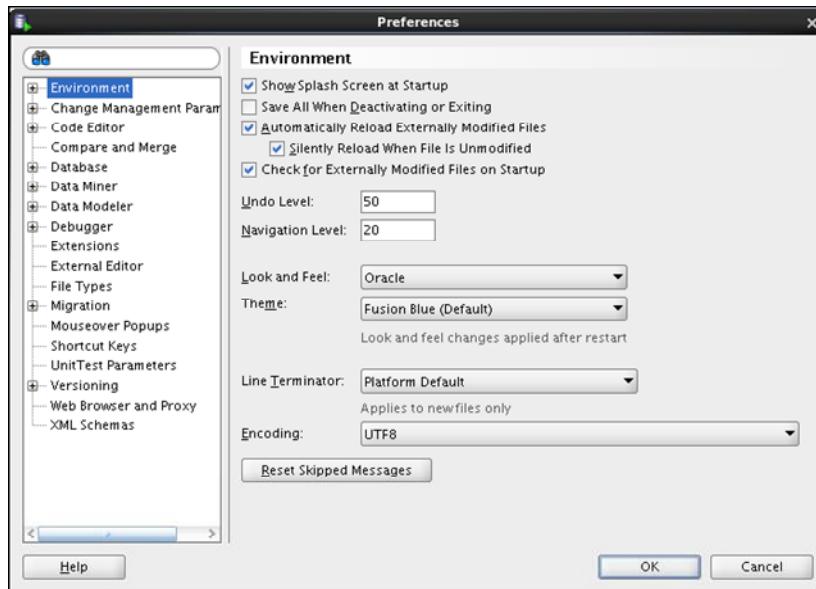
To enhance the productivity of developers, SQL Developer has added quick links to popular search engines and discussion forums such as AskTom, Google, and so on. Also, you have shortcut icons to some of the frequently used tools such as Notepad, Microsoft Word, and Dreamweaver, available to you.

You can add external tools to the existing list or even delete shortcuts to the tools that you do not use frequently. To do so, perform the following steps:

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 © 2013, Oracle and/or its affiliates. All rights reserved.

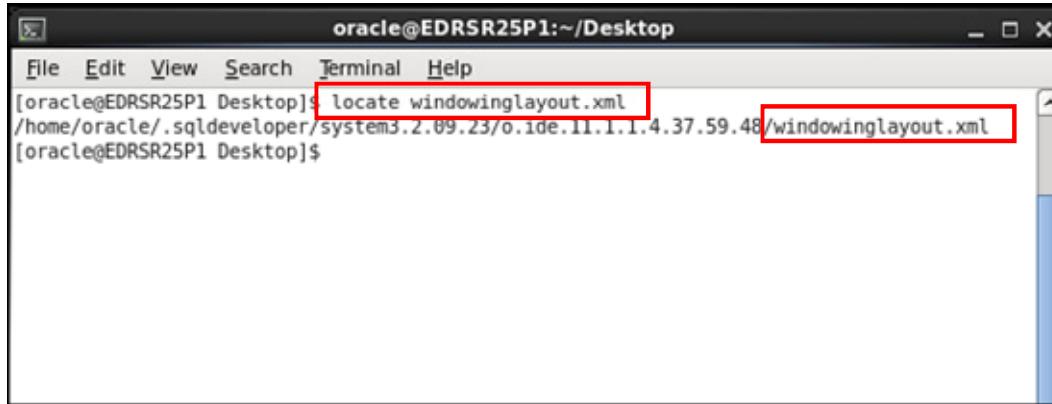
You can customize many aspects of the SQL Developer interface and environment by modifying SQL Developer preferences according to your needs. To modify SQL Developer preferences, select Tools, and then Preferences.

The preferences are grouped into the following categories:

- Environment
- Change Management parameter
- Code Editors
- Compare and Merge
- Database
- Data Miner
- Data Modeler
- Debugger
- Extensions
- External Editor
- File Types
- Migration

- Mouseover Popups
- Shortcut Keys
- Unit Test Parameters
- Versioning
- Web Browser and Proxy
- XML Schemas

Resetting the SQL Developer Layout



A screenshot of a terminal window titled "oracle@EDRSR25P1:~/Desktop". The window shows the following command and its output:

```
[oracle@EDRSR25P1 Desktop]$ locate windowinglayout.xml
/home/oracle/.sqldeveloper/system3.2.09.23/o.ide.11.1.1.4.37.59.48/windowinglayout.xml
[oracle@EDRSR25P1 Desktop]$
```

The command "locate windowinglayout.xml" is highlighted with a red box. The resulting file path "/home/oracle/.sqldeveloper/system3.2.09.23/o.ide.11.1.1.4.37.59.48/windowinglayout.xml" is also highlighted with a red box.

A red rectangular box covers the Oracle logo, which typically consists of the word "ORACLE" in white capital letters on a red background.

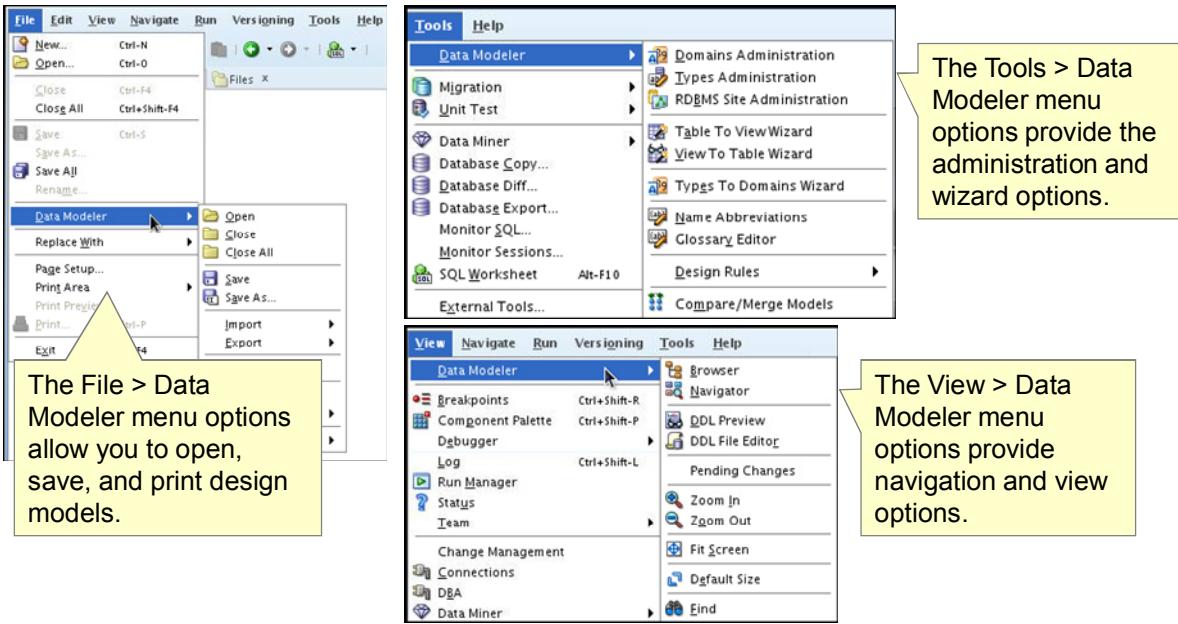
Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

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 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.



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Using the integrated version of the 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, and then Data Modeler. The Data Modeler menu under Tools includes additional commands, for example, that enable you to specify design rules and preferences.

Summary

In this appendix, you should have learned how to use SQL Developer to do:

- 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 © 2013, 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.

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*Plus

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Objectives

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

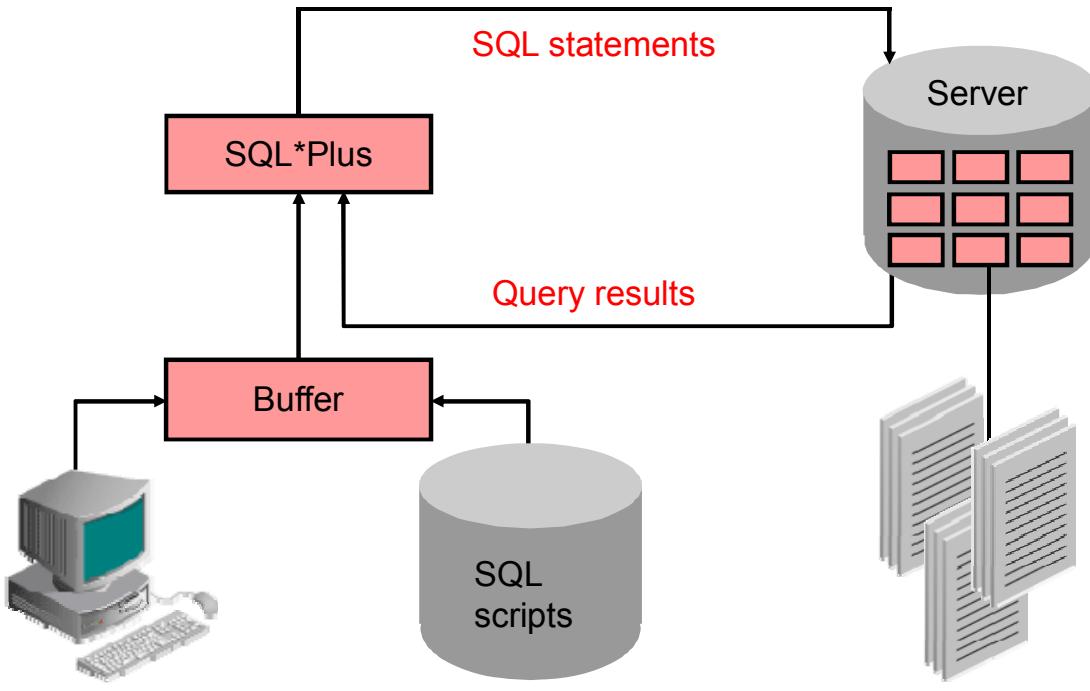
- Log in to SQL*Plus
- Edit SQL commands
- Format the output using SQL*Plus commands
- Interact with script files



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You might want to create `SELECT` statements that can be used repeatedly. This appendix covers the use of SQL*Plus commands to execute SQL statements. You learn how to format output using SQL*Plus commands, edit SQL commands, and save scripts in SQL*Plus.

SQL and SQL*Plus Interaction



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

SQL and SQL*Plus

SQL is a command language that is 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

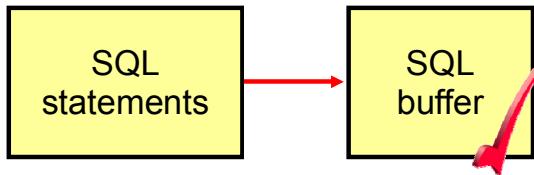
Features of SQL*Plus

- 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

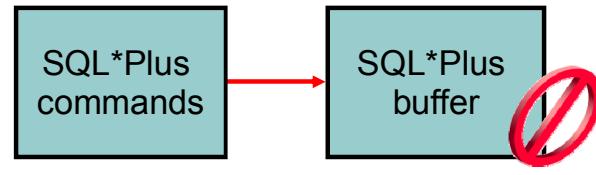
SQL

- 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 © 2013, Oracle and/or its affiliates. All rights reserved.

The following table compares SQL and 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

Overview of 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 from the file to buffer to edit.



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

SQL*Plus

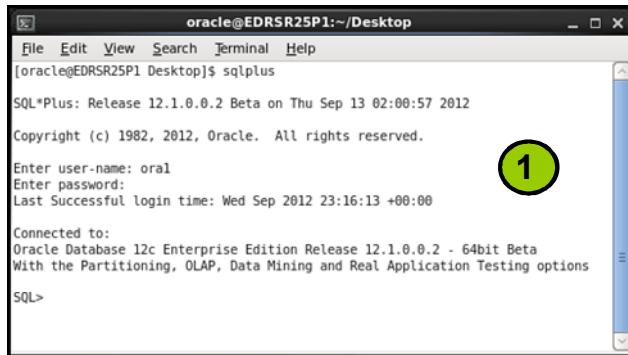
SQL*Plus is an environment in which you can:

- 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 can be divided into the following main 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

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 © 2013, Oracle and/or its affiliates. All rights reserved.

How you invoke SQL*Plus depends on the type of operating system that you are running Oracle Database on.

To log in from a Linux environment, perform the following steps:

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 © 2013, Oracle and/or its affiliates. All rights reserved.

In SQL*Plus, you can display the structure of a table using the DESCRIBE command. The result of the command is a display of column names and data types, as well as an indication if 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 the following 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 © 2013, Oracle and/or its affiliates. All rights reserved.

The example in the slide displays 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

- A [PPEND] *text*
- C [HANGE] / *old* / *new*
- C [HANGE] / *text* /
- CL [EAR] BUFF [ER]
- DEL
- DEL *n*
- DEL *m n*



Copyright © 2013, 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.

Command	Description
A [PPEND] <i>text</i>	Adds <i>text</i> to the end of the current line
C [HANGE] / <i>old</i> / <i>new</i>	Changes <i>old</i> text to <i>new</i> in the current line
C [HANGE] / <i>text</i> /	Deletes <i>text</i> from the current line
CL [EAR] BUFF [ER]	Deletes all lines from the SQL buffer
DEL	Deletes current line
DEL <i>n</i>	Deletes line <i>n</i>
DEL <i>m n</i>	Deletes lines <i>m</i> to <i>n</i> inclusive

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 appears.

SQL*Plus Editing Commands

- I [NPUT]
- I [NPUT] *text*
- L [IST]
- L [IST] *n*
- L [IST] *m n*
- R [UN]
- *n*
- *n text*
- 0 *text*



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Command	Description
I [NPUT]	Inserts an indefinite number of lines
I [NPUT] <i>text</i>	Inserts a line consisting of <i>text</i>
L [IST]	Lists all lines in the SQL buffer
L [IST] <i>n</i>	Lists one line (specified by <i>n</i>)
L [IST] <i>m n</i>	Lists a range of lines (<i>m</i> to <i>n</i>) inclusive
R [UN]	Displays and runs the current SQL statement in the buffer
<i>n</i>	Specifies the line to make the current line
<i>n text</i>	Replaces line <i>n</i> with <i>text</i>
0 <i>text</i>	Inserts a line before line 1

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 © 2013, 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
```



Copyright © 2013, 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

- `SAVE filename`
- `GET filename`
- `START filename`
- `@ filename`
- `EDIT filename`
- `SPOOL filename`
- `EXIT`



Copyright © 2013, 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 following table:

Command	Description
<code>SAV[E] filename [.ext]</code> [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.
<code>GET filename [.ext]</code>	Writes the contents of a previously saved file to the SQL buffer. The default extension for the file name is .sql.
<code>STA[RT] filename [.ext]</code>	Runs a previously saved command file
<code>@ filename</code>	Runs a previously saved command file (same as START)
<code>ED[IT]</code>	Invokes the editor and saves the buffer contents to a file named afiedt.buf
<code>ED[IT] [filename [.ext]]</code>	Invokes the editor to edit the contents of a saved file
<code>SPO[OL] [filename [.ext]] OFF OUT</code>	Stores query results in a file. OFF closes the spool file. OUT closes the spool file and sends the file results to the printer.
<code>EXIT</code>	Quits SQL*Plus

Using the SAVE, 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 © 2013, Oracle and/or its affiliates. All rights reserved.

SAVE

Use the `SAVE` command to store the current contents of the buffer in a file. Thus, you can store frequently used scripts for use in the future.

START

Use the `START` command to run a script in SQL*Plus. You can also, alternatively, use the symbol `@` to run a script.

```
@my_query
```

SERVERTOUTPUT 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 © 2013, 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 requires 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 *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 © 2013, 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 use SPOOL only 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 screen, use SET TERMOUT OFF. SET TERMOUT OFF does not affect the output from commands that run interactively.

You must use quotation marks around file names that contain 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 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 © 2013, 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, refer to *Oracle Database PL/SQL Packages and Types Reference*.
- For additional information about the EXPLAIN PLAN, refer to *Oracle Database SQL Reference*.
- For additional information about Execution Plans and the statistics, refer to *Oracle Database Performance Tuning Guide*.

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 © 2013, 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.

Manipulating Large Data Sets

ORACLE®

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Objectives

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

- Manipulate data using subqueries
- Specify explicit default values in the `INSERT` and `UPDATE` statements
- Describe the features of multitable `INSERTS`
- Use the following types of multitable `INSERTS`:
 - Unconditional `INSERT`
 - Pivoting `INSERT`
 - Conditional `INSERT ALL`
 - Conditional `INSERT FIRST`
- Merge rows in a table
- Track the changes to data over a period of time



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In this appendix, you learn how to manipulate data in the Oracle database by using subqueries. You learn how to use the `DEFAULT` keyword in `INSERT` and `UPDATE` statements to identify a default column value. You also learn about multitable `INSERT` statements, the `MERGE` statement, and tracking changes in the database.

Using Subqueries to Manipulate Data

You can use subqueries in data manipulation language (DML) statements to:

- Retrieve data using an inline view
- Copy data from one table to another
- Update data in one table based on the values of another table
- Delete rows from one table based on rows in another table



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Subqueries can be used to retrieve data from a table that you can use as input to an `INSERT` into a different table. In this way, you can easily copy large volumes of data from one table to another with one single `SELECT` statement. Similarly, you can use subqueries to do mass updates and deletes by using them in the `WHERE` clause of the `UPDATE` and `DELETE` statements. You can also use subqueries in the `FROM` clause of a `SELECT` statement. This is called an inline view.

Note: You learned how to update and delete rows based on another table in the course titled *Oracle Database: SQL Fundamentals I*.

Retrieving Data Using a Subquery as Source

```
SELECT department_name, city
FROM departments
NATURAL JOIN (SELECT l.location_id, l.city, l.country_id
               FROM loc l
               JOIN countries c
                 ON(l.country_id = c.country_id)
               JOIN regions USING(region_id)
              WHERE region_name = 'Europe'
                and c.country_id = 'UK');
```

DEPARTMENT_NAME	CITY
1 Human Resources	London
2 Sales	Oxford



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You can use a subquery in the `FROM` clause of a `SELECT` statement, which is very similar to how views are used. A subquery in the `FROM` clause of a `SELECT` statement is also called an *inline view*. A subquery in the `FROM` clause of a `SELECT` statement defines a data source for that particular `SELECT` statement, and only that `SELECT` statement. As with a database view, the `SELECT` statement in the subquery can be as simple or as complex as you like.

When a database view is created, the associated `SELECT` statement is stored in the data dictionary. In situations where you do not have the necessary privileges to create database views, or when you would like to test the suitability of a `SELECT` statement to become a view, you can use an inline view.

With inline views, you can have all the code needed to support the query in one place. This means that you can avoid the complexity of creating a separate database view. The slide example shows how to use an inline view to display the department name and the city in Europe. The subquery in the `FROM` clause fetches the location ID, city name, and the country by joining three different tables. The output of the inner query is considered as a table for the outer query. The inner query is similar to that of a database view but does not have any physical name.

For the example in the slide, the `loc` table is created by running the following statement:

```
CREATE TABLE loc AS SELECT * FROM locations;
```

You can display the same output as in the slide example by performing the following two steps:

1. Create a database view:

```
CREATE OR REPLACE VIEW european_cities
AS
SELECT l.location_id, l.city, l.country_id
FROM   loc l
JOIN   countries c
ON(l.country_id = c.country_id)
JOIN regions USING(region_id)
WHERE region_name = 'Europe' and c.country_id ='UK';
```

2. Join the EUROPEAN_CITIES view with the DEPARTMENTS table:

```
SELECT department_name, city
FROM   departments
NATURAL JOIN european_cities;
```

Note: You learned how to create database views in the course titled *Oracle Database: SQL Fundamentals I*.

Inserting Using a Subquery as a Target

```
INSERT INTO (SELECT l.location_id, l.city, l.country_id
             FROM loc l
             JOIN countries c
               ON(l.country_id = c.country_id)
             JOIN regions USING(region_id)
            WHERE region_name = 'Europe')
VALUES (3300, 'Cardiff', 'UK');

1 rows inserted.
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You can use a subquery in place of the table name in the `INTO` clause of the `INSERT` statement. The `SELECT` list of this subquery must have the same number of columns as the column list of the `VALUES` clause. Any rules on the columns of the base table must be followed in order for the `INSERT` statement to work successfully. For example, you cannot put in a duplicate location ID or leave out a value for a mandatory `NOT NULL` column.

This application of subqueries helps you avoid having to create a view just for performing an `INSERT`.

The slide example uses a subquery in the place of `LOC` to create a record for a new European city.

Note: You can also perform the `INSERT` operation on the `EUROPEAN_CITIES` view by using the following code:

```
INSERT INTO european_cities
VALUES (3300, 'Cardiff', 'UK');
```

Inserting Using a Subquery as a Target: Viewing the Results

```
SELECT location_id, city, country_id  
FROM loc;
```

	LOCATION_ID	CITY	COUNTRY_ID
1	1000 Roma	IT	
2	1100 Venice	IT	
...			
22	3100 Utrecht	NL	
23	3200 Mexico City	MX	
24	3300 Cardiff	UK	



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The slide example shows that the insert via the inline view created a new record in the LOC base table.

The following example shows the results of the subquery that was used to identify the table for the INSERT statement.

```
SELECT l.location_id, l.city, l.country_id  
FROM loc l  
JOIN countries c  
ON(l.country_id = c.country_id)  
JOIN regions USING(region_id)  
WHERE region_name = 'Europe' and c.country_id = 'UK';
```

	DEPARTMENT_NAME	CITY
1	Human Resources	London
2	Sales	Oxford

Using the WITH CHECK OPTION Keyword on DML Statements

The WITH CHECK OPTION keyword prohibits you from changing rows that are not in the subquery.

```
INSERT INTO ( SELECT location_id, city, country_id
    FROM loc
    WHERE country_id IN
        (SELECT country_id
            FROM countries
            NATURAL JOIN regions
            WHERE region_name = 'Europe')
        WITH CHECK OPTION )
VALUES (3600, 'Washington', 'US');
```

Error starting at line 1 in command:
 INSERT INTO (SELECT location_id, city, country_id
 FROM loc
 WHERE country_id IN
 (SELECT country_id
 FROM countries
 NATURAL JOIN regions
 WHERE region_name = 'Europe')
 WITH CHECK OPTION)
 VALUES (3600, 'Washington', 'US')
 Error report:
 SQL Error: ORA-01402: view WITH CHECK OPTION where-clause violation
 01402. 00000 - "view WITH CHECK OPTION where-clause violation"
 *Cause:
 *Action:

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Specify the WITH CHECK OPTION keyword to indicate that if the subquery is used in place of a table in an INSERT, UPDATE, or DELETE statement, no changes that produce rows that are not included in the subquery are permitted to that table.

The example in the slide shows how to use an inline view with the WITH CHECK OPTION. The INSERT statement prevents the creation of records in the LOC table for a city that is not in Europe.

The following example executes successfully because of the changes in the VALUES list.

```
INSERT INTO (SELECT location_id, city, country_id
    FROM loc
    WHERE country_id IN
        (SELECT country_id
            FROM countries
            NATURAL JOIN regions
            WHERE region_name = 'Europe')
        WITH CHECK OPTION)
VALUES (3500, 'Berlin', 'DE');
```

1 rows inserted.

The application of an inline view with the WITH CHECK OPTION provides an easy method to prevent changes to the table.

To prevent the creation of a non-European city, you can also use a database view by performing the following steps:

1. Create a database view:

```
CREATE OR REPLACE VIEW european_cities
AS
SELECT location_id, city, country_id
FROM locations
WHERE country_id in
(SELECT country_id
FROM countries
NATURAL JOIN regions
WHERE region_name = 'Europe')
WITH CHECK OPTION;
```

2. Verify the results by inserting data:

```
INSERT INTO european_cities
VALUES (3400, 'New York', 'US');
```

The second step produces the same error as shown in the slide.

Overview of the Explicit Default Feature

- Use the `DEFAULT` keyword as a column value where the default column value is desired.
- This allows the user to control where and when the default value should be applied to data.
- Explicit defaults can be used in `INSERT` and `UPDATE` statements.



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The `DEFAULT` keyword can be used in `INSERT` and `UPDATE` statements to identify a default column value. If no default value exists, a null value is used.

The `DEFAULT` option saves you from hard-coding the default value in your programs or querying the dictionary to find it, as was done before this feature was introduced. Hard coding the default is a problem if the default changes because the code consequently needs changing. Accessing the dictionary is not usually done in an application program, so this is a very important feature.

Using Explicit Default Values

- Create a new table:

```
CREATE TABLE deptm3 AS  
SELECT * FROM departments;
```

- DEFAULT with INSERT:

```
INSERT INTO deptm3  
(department_id, department_name, manager_id)  
VALUES (300, 'Engineering', DEFAULT);
```

- DEFAULT with UPDATE:

```
UPDATE deptm3  
SET manager_id = DEFAULT  
WHERE department_id = 10;
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Specify `DEFAULT` to set the column to the value previously specified as the default value for the column. If no default value for the corresponding column has been specified, the Oracle server sets the column to null.

In the first example in the slide, the `INSERT` statement uses a default value for the `MANAGER_ID` column. If there is no default value defined for the column, a null value is inserted instead.

The second example uses the `UPDATE` statement to set the `MANAGER_ID` column to a default value for department 10. If no default value is defined for the column, it changes the value to null.

Note: When creating a table, you can specify a default value for a column. This is discussed in the course titled *Oracle Database: SQL Fundamentals I*.

Copying Rows from Another Table

```
CREATE TABLE sales_reps AS SELECT employee_id id, last_name  
    name, salary, commission_pct  
FROM   employees  
WHERE  1=2;
```

- Write your `INSERT` statement with a subquery.

```
INSERT INTO sales_reps(id, name, salary, commission_pct)  
SELECT employee_id, last_name, salary, commission_pct  
FROM   employees  
WHERE  job_id LIKE '%REP%';
```

33 rows inserted.

- Match the number of columns in the `INSERT` clause with that in the subquery.



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You can use the `INSERT` statement to add rows to a table where the values are derived from existing tables. Do not use the `VALUES` clause, instead use a subquery.

Syntax

```
INSERT INTO table [ column (, column) ] subquery;
```

In the syntax:

<code>table</code>	Is the table name
<code>column</code>	Is the name of the column in the table to populate
<code>subquery</code>	Is the subquery that returns rows into the table

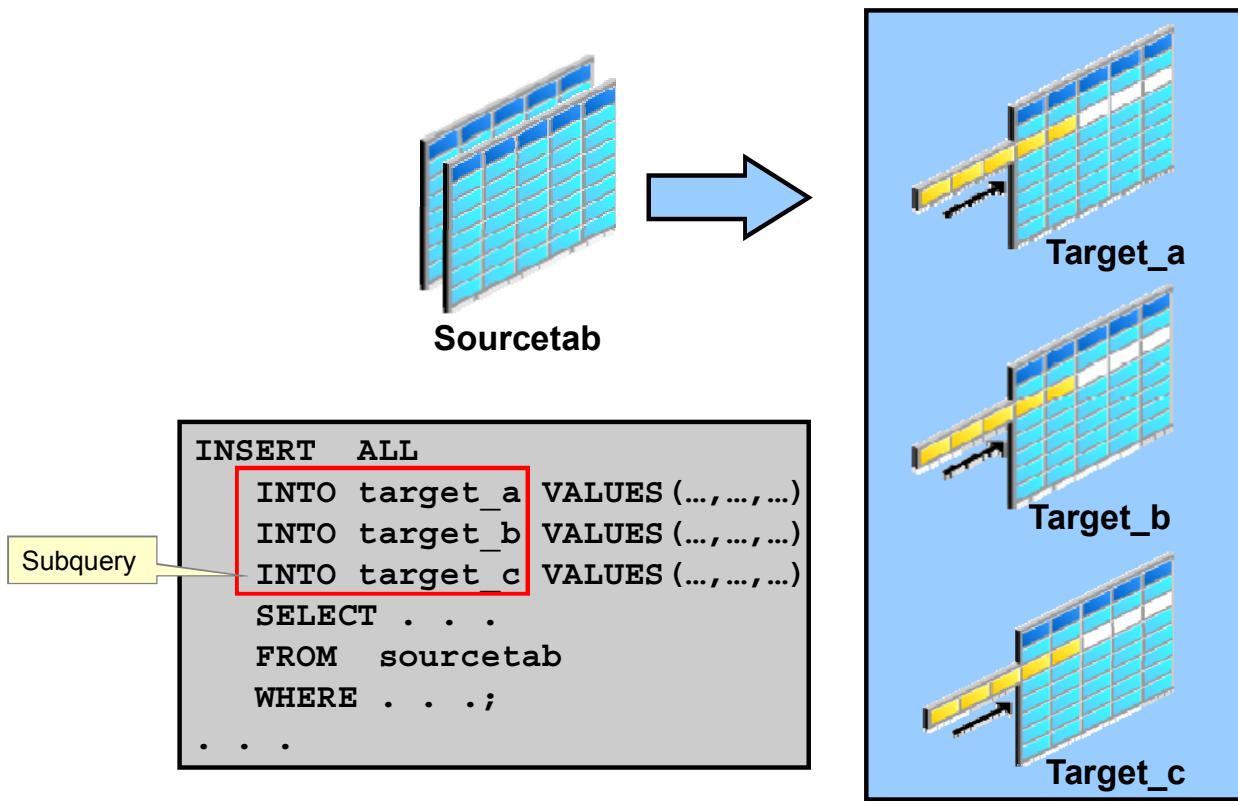
The number of columns and their data types in the column list of the `INSERT` clause must match the number of values and their data types in the subquery. To create a copy of the rows of a table, use `SELECT *` in the subquery.

...

```
INSERT INTO EMPL3  
SELECT * FROM employees;
```

Note: You use the `LOG ERRORS` clause in your DML statement to enable the DML operation to complete regardless of errors. Oracle SQL writes the details of the error message to an error-logging table that you have created. For more information, see *Oracle Database SQL Reference*.

Overview of Multitable INSERT Statements



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In a multitable `INSERT` statement, you insert computed rows derived from the rows returned from the evaluation of a subquery into one or more tables.

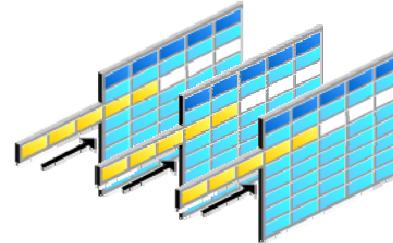
Multitable `INSERT` statements are useful in a data warehouse scenario. You need to load your data warehouse regularly so that it can serve its purpose of facilitating business analysis. To do this, data from one or more operational systems must be extracted and copied into the warehouse. The process of extracting data from the source system and bringing it into the data warehouse is commonly called extraction, transformation, and loading (ETL).

During extraction, the desired data must be identified and extracted from many different sources, such as database systems and applications. After extraction, the data must be physically transported to the target system or an intermediate system for further processing. Depending on the chosen means of transportation, some transformations can be done during this process. For example, a SQL statement that directly accesses a remote target through a gateway can concatenate two columns as part of the `SELECT` statement.

After data is loaded into the Oracle Database, data transformations can be executed using SQL operations. A multitable `INSERT` statement is one of the techniques for implementing SQL data transformations.

Overview of Multitable INSERT Statements

- Use the `INSERT...SELECT` statement to insert rows into multiple tables as part of a single DML statement.
- Multitable `INSERT` statements are used in data warehousing systems to transfer data from one or more operational sources to a set of target tables.
- They provide significant performance improvement over:
 - Single DML versus multiple `INSERT...SELECT` statements
 - Single DML versus a procedure to perform multiple inserts by using the `IF...THEN` syntax



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Multitable `INSERT` statements offer the benefits of the `INSERT . . . SELECT` statement when multiple tables are involved as targets. Without multitable `INSERT`, you had to deal with n independent `INSERT . . . SELECT` statements, thus processing the same source data n times and increasing the transformation workload n times.

As with the existing `INSERT . . . SELECT` statement, the new statement can be parallelized and used with the direct-load mechanism for faster performance.

Each record from any input stream, such as a nonrelational database table, can now be converted into multiple records for a more relational database table environment. To alternatively implement this functionality, you were required to write multiple `INSERT` statements.

Types of Multitable INSERT Statements

The different types of multitable INSERT statements are:

- **Unconditional INSERT**
- **Conditional INSERT ALL**
- **Pivoting INSERT**
- **Conditional INSERT FIRST**



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You can use different clauses to indicate the type of INSERT to be executed. The types of multitable INSERT statements are:

- **Unconditional INSERT:** For each row returned by the subquery, a row is inserted into each of the target tables.
- **Conditional INSERT ALL:** For each row returned by the subquery, a row is inserted into each target table if the specified condition is met.
- **Pivoting INSERT:** This is a special case of the unconditional INSERT ALL.
- **Conditional INSERT FIRST:** For each row returned by the subquery, a row is inserted into the very first target table in which the condition is met.

Multitable INSERT Statements

- Syntax for multitable INSERT:

```
INSERT [conditional_insert_clause]
[insert_into_clause values_clause] (subquery)
```

- conditional_insert_clause:

```
[ALL] [FIRST]
[WHEN condition THEN] [insert_into_clause values_clause]
[ELSE] [insert_into_clause values_clause]
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The slide displays the generic format for multitable INSERT statements.

Unconditional INSERT: ALL into_clause

Specify ALL followed by multiple insert_into_clauses to perform an unconditional multitable INSERT. The Oracle server executes each insert_into_clause once for each row returned by the subquery.

Conditional INSERT: conditional_insert_clause

Specify the conditional_insert_clause to perform a conditional multitable INSERT. The Oracle server filters each insert_into_clause through the corresponding WHEN condition, which determines whether that insert_into_clause is executed. A single multitable INSERT statement can contain up to 127 WHEN clauses.

Conditional INSERT: ALL

If you specify ALL, the Oracle server evaluates each WHEN clause regardless of the results of the evaluation of any other WHEN clause. For each WHEN clause whose condition evaluates to true, the Oracle server executes the corresponding INTO clause list.

Conditional INSERT: FIRST

If you specify FIRST, the Oracle server evaluates each WHEN clause in the order in which it appears in the statement. If the first WHEN clause evaluates to true, the Oracle server executes the corresponding INTO clause and skips subsequent WHEN clauses for the given row.

Conditional INSERT: ELSE Clause

For a given row, if no WHEN clause evaluates to true:

- If you have specified an ELSE clause, the Oracle server executes the INTO clause list associated with the ELSE clause
- If you did not specify an ELSE clause, the Oracle server takes no action for that row

Restrictions on Multitable INSERT Statements

- You can perform multitable INSERT statements only on tables, and not on views or materialized views.
- You cannot perform a multitable INSERT on a remote table.
- You cannot specify a table collection expression when performing a multitable INSERT.
- In a multitable INSERT, all insert_into_clauses cannot combine to specify more than 999 target columns.

Unconditional INSERT ALL

- Select the EMPLOYEE_ID, HIRE_DATE, SALARY, and MANAGER_ID values from the EMPLOYEES table for those employees whose EMPLOYEE_ID is greater than 200.
- Insert these values into the SAL_HISTORY and MGR_HISTORY tables by using a multitable INSERT.

```
INSERT ALL
  INTO sal_history VALUES (EMPID, HIREDATE, SAL)
  INTO mgr_history VALUES (EMPID, MGR, SAL)
  SELECT employee_id EMPID, hire_date HIREDATE,
         salary SAL, manager_id MGR
    FROM employees
   WHERE employee_id > 200;
```

12 rows inserted.



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The example in the slide inserts rows into both the SAL_HISTORY and the MGR_HISTORY tables.

The SELECT statement retrieves the details of employee ID, hire date, salary, and manager ID of those employees whose employee ID is greater than 200 from the EMPLOYEES table. The details of the employee ID, hire date, and salary are inserted into the SAL_HISTORY table. The details of employee ID, manager ID, and salary are inserted into the MGR_HISTORY table.

This INSERT statement is referred to as an unconditional INSERT because no further restriction is applied to the rows that are retrieved by the SELECT statement. All the rows retrieved by the SELECT statement are inserted into the two tables: SAL_HISTORY and MGR_HISTORY. The VALUES clause in the INSERT statements specifies the columns from the SELECT statement that must be inserted into each of the tables. Each row returned by the SELECT statement results in two insertions: one for the SAL_HISTORY table and one for the MGR_HISTORY table.

A total of eight rows were inserted:

```
SELECT COUNT(*) total_in_sal FROM sal_history;
```

TOTAL_IN_SAL	
1	6

```
SELECT COUNT(*) total_in_mgr FROM mgr_history;
```

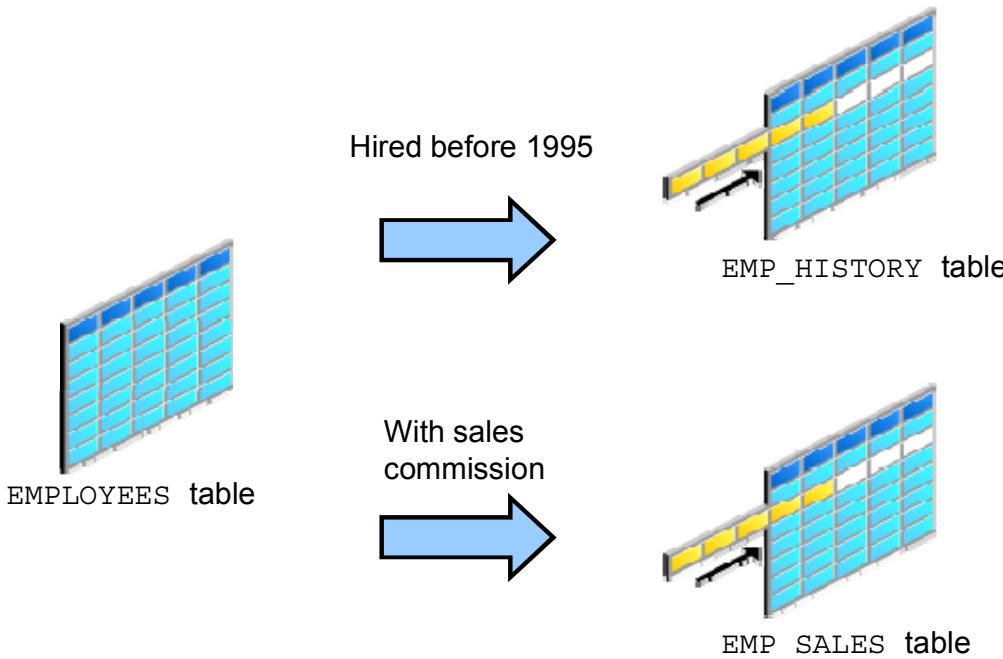
TOTAL_IN_MGR	
1	6

The definitions of the `sal_history` and `mgr_history` are as follows:

```
CREATE table SAL_HISTORY
(EMPLOYEE_ID    NUMBER(6),
HIRE_DATE DATE,
SALARY      NUMBER(8,2));
```

```
CREATE table MGR_HISTORY
(EMPLOYEE_ID    NUMBER(6),
MANAGER_ID     NUMBER(6),
SALARY      NUMBER(8,2));
```

Conditional INSERT ALL: Example



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

For all employees in the employees tables, if the employee was hired before 1995, insert that employee record into the employee history. If the employee earns a sales commission, insert the record information into the `EMP_SALES` table. The SQL statement follows in the next slide.

The definition of the `EMP_SALES` and `EMP_HISTORY` tables is as follows:

```
CREATE table EMP_SALES
(EMPLOYEE_ID    NUMBER(6),
COMMISSION_PCT NUMBER(2,2),
SALARY         NUMBER(8,2));

CREATE table EMP_HISTORY
(EMPLOYEE_ID    NUMBER(6),
HIRE_DATE DATE,
SALARY        NUMBER(8,2));
```

Conditional INSERT ALL

```
INSERT ALL
  WHEN HIREDATE < '01-JAN-03' THEN
    INTO emp_history VALUES (EMPID, HIREDATE, SAL)
  WHEN COMM IS NOT NULL THEN
    INTO emp_sales VALUES (EMPID, COMM, SAL)
  SELECT employee_id EMPID, hire_date HIREDATE,
         salary SAL, commission_pct COMM
  FROM employees;
```

43 rows inserted.

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The example in the slide is similar to the example in the previous slide because it inserts rows into both the `EMP_HISTORY` and the `EMP_SALES` tables. The `SELECT` statement retrieves details such as employee ID, hire date, salary, and commission percentage for all employees from the `EMPLOYEES` table. Details such as employee ID, hire date, and salary are inserted into the `EMP_HISTORY` table. Details such as employee ID, commission percentage, and salary are inserted into the `EMP_SALES` table.

This `INSERT` statement is referred to as a conditional `INSERT ALL` because a further restriction is applied to the rows that are retrieved by the `SELECT` statement. From the rows that are retrieved by the `SELECT` statement, only those rows in which the hire date was before 2003 are inserted in the `EMP_HISTORY` table. Similarly, only those rows where the value of commission percentage is not null are inserted in the `EMP_SALES` table.

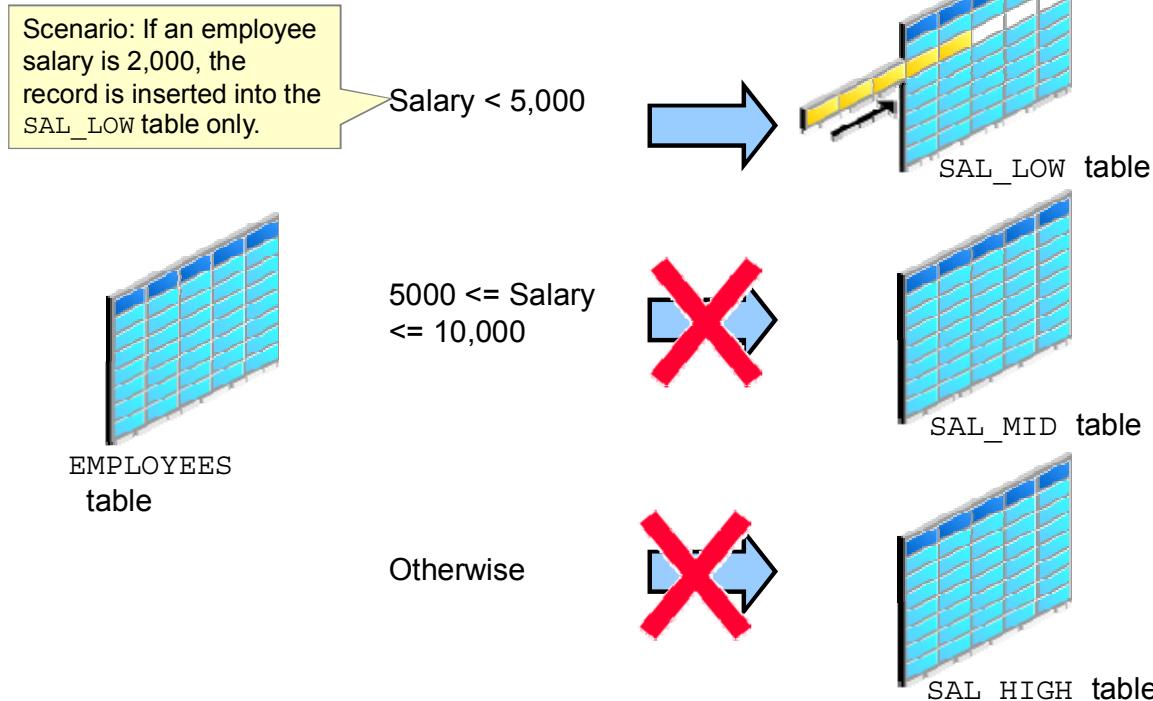
```
SELECT count(*) FROM emp_history;
```

	COUNT(*)
1	8

```
SELECT count(*) FROM emp_sales;
```

	COUNT(*)
1	35

Conditional INSERT FIRST: Example



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

For all employees in the EMPLOYEES table, insert the employee information into the first target table that meets the condition. In the example, if an employee has a salary of 2,000, the record is inserted into the SAL_LOW table only. The SQL statement follows in the next slide.

Conditional INSERT FIRST

```
INSERT FIRST  
WHEN salary < 5000 THEN  
    INTO sal_low VALUES (employee_id, last_name, salary)  
WHEN salary between 5000 and 10000 THEN  
    INTO sal_mid VALUES (employee_id, last_name, salary)  
ELSE  
    INTO sal_high VALUES (employee_id, last_name, salary)  
SELECT employee_id, last_name, salary  
FROM employees;
```

107 rows inserted.



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The SELECT statement retrieves details such as employee ID, last name, and salary for every employee in the EMPLOYEES table. For each employee record, it is inserted into the very first target table that meets the condition.

This INSERT statement is referred to as a conditional INSERT FIRST. The WHEN salary < 5000 condition is evaluated first. If this first WHEN clause evaluates to true, the Oracle server executes the corresponding INTO clause and inserts the record into the SAL_LOW table. It skips subsequent WHEN clauses for this row.

If the row that does not satisfy the first WHEN condition (WHEN salary < 5000), the next condition (WHEN salary between 5000 and 10000) is evaluated. If this condition evaluates to true, the record is inserted into the SAL_MID table, and the last condition is skipped.

If neither the first condition (WHEN salary < 5000) nor the second condition (WHEN salary between 5000 and 10000) is evaluated to true, the Oracle server executes the corresponding INTO clause for the ELSE clause.

A total of 20 rows were inserted:

```
SELECT count(*) low FROM sal_low;
```

	LOW
1	49

```
SELECT count(*) mid FROM sal_mid;
```

	MID
1	43

```
SELECT count(*) high FROM sal_high;
```

	HIGH
1	15

The definition of the SAL_LOW, SAL_MID, and SAL_HIGH tables are as follows:

```
CREATE TABLE sal_low
  ( employee_id number(6,0),
    last_name varchar2(25),
    salary number(8,2));
CREATE TABLE sal_mid
  ( employee_id number(6,0),
    last_name varchar2(25),
    salary number(8,2));
CREATE TABLE sal_high
  ( employee_id number(6,0),
    last_name varchar2(25),
    salary number(8,2));
```

Pivoting INSERT

Convert the set of sales records from the nonrelational database table to relational format.

Emp_ID	Week_ID	MON	TUES	WED	THUR	FRI
176	6	2000	3000	4000	5000	6000



Employee_ID	WEEK	SALES
176	6	2000
176	6	3000
176	6	4000
176	6	5000
176	6	6000

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Pivoting is an operation in which you must build a transformation such that each record from any input stream, such as a nonrelational database table, is converted into multiple records for a more relational database table environment.

Suppose you receive a set of sales records from a nonrelational database table:

SALES_SOURCE_DATA, in the following format:

```
EMPLOYEE_ID, WEEK_ID, SALES_MON, SALES_TUE, SALES_WED,
SALES_THUR, SALES_FRI
```

You want to store these records in the SALES_INFO table in a more typical relational format:

EMPLOYEE_ID, WEEK, SALES

To solve this problem, you must build a transformation such that each record from the original nonrelational database table, SALES_SOURCE_DATA, is converted into five records for the data warehouse's SALES_INFO table. This operation is commonly referred to as *pivoting*.

The solution of this problem follows in the next slide.

The definition of the SALES_INFO table is as follows:

```
CREATE TABLE SALES_INFO
(employee_id    NUMBER(6),
WEEKNUMBER(2),
SALES      NUMBER(8,2));
```

Pivoting INSERT

```
INSERT ALL
  INTO sales_info VALUES (employee_id,week_id,sales_MON)
  INTO sales_info VALUES (employee_id,week_id,sales_TUE)
  INTO sales_info VALUES (employee_id,week_id,sales_WED)
  INTO sales_info VALUES (employee_id,week_id,sales_THUR)
  INTO sales_info VALUES (employee_id,week_id, sales_FRI)
    SELECT EMPLOYEE_ID, week_id, sales_MON, sales_TUE,
           sales_WED, sales_THUR,sales_FRI
      FROM sales_source_data;
```

5 rows inserted

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In the example in the slide, the sales data is received from the nonrelational database table SALES_SOURCE_DATA, which is the details of the sales performed by a sales representative on each day of a week, for a week with a particular week ID. The definition of the SALES_INFO table follows on the next page.

```
DESC SALES_SOURCE_DATA
```

Name	Null	Type
EMPLOYEE_ID		NUMBER(6)
WEEK_ID		NUMBER(2)
SALES_MON		NUMBER(8,2)
SALES_TUE		NUMBER(8,2)
SALES_WED		NUMBER(8,2)
SALES_THUR		NUMBER(8,2)
SALES_FRI		NUMBER(8,2)

```
SELECT * FROM SALES_SOURCE_DATA;
```

	EMPLOYEE_ID	WEEK_ID	SALES_MON	SALES_TUE	SALES_WED	SALES_THUR	SALES_FRI
1	178	6	1750	2200	1500	1500	3000

```
DESC SALES_INFO
```

Name	Null	Type
EMPLOYEE_ID		NUMBER(6)
WEEK		NUMBER(2)
SALES		NUMBER(8,2)

```
SELECT * FROM sales_info;
```

	EMPLOYEE_ID	WEEK	SALES
1	178	6	1750
2	178	6	2200
3	178	6	1500
4	178	6	1500
5	178	6	3000

Observe that by using a pivoting `INSERT`, one row from the `SALES_SOURCE_DATA` table is converted into five records for the relational table, `SALES_INFO`.

The definition and the of the `SALES_SOURCE_DATA` table with the `INSERT` statement is as follows:

```
CREATE TABLE SALES_SOURCE_DATA
(employee_id    NUMBER(6),
WEEK_ID        NUMBER(2),
SALES_MON      NUMBER(8, 2),
SALES_TUE      NUMBER(8, 2),
SALES_WED      NUMBER(8, 2),
SALES_THUR     NUMBER(8, 2),
SALES_FRI      NUMBER(8, 2));
INSERT INTO sales_source_data
VALUES (178, 6, 1750, 2200, 1500, 1500, 3000);
```

MERGE Statement

- Provides the ability to conditionally update, insert, or delete data into a database table
- Performs an UPDATE if the row exists, and an INSERT if it is a new row:
 - Avoids separate updates
 - Increases performance and ease of use
 - Is useful in data warehousing applications



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The Oracle server supports the MERGE statement for INSERT, UPDATE, and DELETE operations. Using this statement, you can update, insert, or delete a row conditionally into a table, thus avoiding multiple DML statements. The decision whether to update, insert, or delete into the target table is based on a condition in the ON clause.

You must have the INSERT and UPDATE object privileges on the target table and the SELECT object privilege on the source table. To specify the DELETE clause of merge_update_clause, you must also have the DELETE object privilege on the target table.

The MERGE statement is deterministic. You cannot update the same row of the target table multiple times in the same MERGE statement.

An alternative approach is to use PL/SQL loops and multiple DML statements. The MERGE statement, however, is easy to use and more simply expressed as a single SQL statement.

The MERGE statement is suitable in a number of data warehousing applications. For example, in a data warehousing application, you may need to work with data coming from multiple sources, some of which may be duplicates. With the MERGE statement, you can conditionally add or modify rows.

MERGE Statement Syntax

You can conditionally insert, update, or delete rows in a table by using the MERGE statement.

```
MERGE INTO table_name table_alias
  USING (table/view/sub_query) alias
  ON (join condition)
  WHEN MATCHED THEN
    UPDATE SET
      col1 = col1_val,
      col2 = col2_val
  WHEN NOT MATCHED THEN
    INSERT (column_list)
    VALUES (column_values);
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You can update existing rows and insert new rows conditionally by using the MERGE statement. Using the MERGE statement, you can delete obsolete rows at the same time as you update rows in a table. To do this, you include a DELETE clause with its own WHERE clause in the syntax of the MERGE statement.

In the syntax:

INTO clause	Specifies the target table you are updating or inserting into
USING clause	Identifies the source of the data to be updated or inserted; can be a table, view, or subquery
ON clause	The condition on which the MERGE operation either updates or inserts
WHEN MATCHED	Instructs the server how to respond to the results of the join condition
WHEN NOT MATCHED	

For more information, see the *Oracle Database SQL Reference Guide*.

Merging Rows: Example

Insert or update rows in the COPY_EMP3 table to match the EMPLOYEES table.

```
MERGE INTO copy_emp3 c
USING (SELECT * FROM EMPLOYEES ) e
ON (c.employee_id = e.employee_id)
WHEN MATCHED THEN
UPDATE SET
  c.first_name = e.first_name,
  c.last_name = e.last_name,
  ...
DELETE WHERE (e.COMMISSION_PCT IS NOT NULL)
WHEN NOT MATCHED THEN
INSERT VALUES(e.employee_id, e.first_name, e.last_name,
  e.email, e.phone_number, e.hire_date, e.job_id,
  e.salary, e.commission_pct, e.manager_id,
  e.department_id);
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

```
MERGE INTO copy_emp3 c
USING (SELECT * FROM EMPLOYEES ) e
ON (c.employee_id = e.employee_id)
WHEN MATCHED THEN
UPDATE SET
  c.first_name = e.first_name,
  c.last_name = e.last_name,
  c.email = e.email,
  c.phone_number = e.phone_number,
  c.hire_date = e.hire_date,
  c.job_id = e.job_id,
  c.salary = e.salary*2,
  c.commission_pct = e.commission_pct,
  c.manager_id = e.manager_id,
  c.department_id = e.department_id
```

```
DELETE WHERE (E.COMMISSION_PCT IS NOT NULL)
WHEN NOT MATCHED THEN
INSERT VALUES(e.employee_id, e.first_name, e.last_name,
e.email, e.phone_number, e.hire_date, e.job_id,
e.salary, e.commission_pct, e.manager_id,
e.department_id);
```

The COPY_EMP3 table is created by using the following code:

```
CREATE TABLE COPY_EMP3 AS SELECT * FROM EMPLOYEES
WHERE SALARY>10000;
```

Then query the COPY_EMP3 table.

```
SELECT employee_id, salary, commission_pct FROM COPY_EMP3;
```

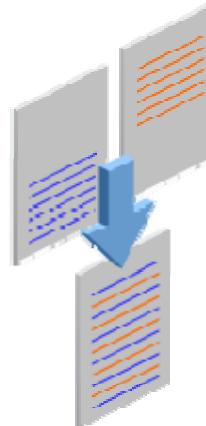
	EMPLOYEE_ID	SALARY	COMMISSION_PCT
1	100	24000	
2	101	17000	
3	102	17000	
4	108	12008	
5	114	11000	
6	145	14000	0.4
7	146	13500	0.3
8	147	12000	0.3
9	148	11000	0.3
10	149	10500	0.2
11	162	10500	0.25
12	168	11500	0.25
13	174	11000	0.3
14	201	13000	
15	205	12008	

Observe that there are fifteen employees with SALARY > 10000 and there are eight employees with COMMISSION_PCT.

The example in the slide matches the EMPLOYEE_ID in the COPY_EMP3 table to the EMPLOYEE_ID in the EMPLOYEES table. If a match is found, the row in the COPY_EMP3 table is updated to match the row in the EMPLOYEES table and the salary of the employee is doubled. If the match is not found, rows are inserted into the COPY_EMP3 table.

MERGE Improvements

- New conditional clauses and extensions to the standard MERGE statement
- An optional DELETE clause to the MERGE statement



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Oracle's data warehousing extraction, transformation, and loading (ETL) capabilities are improved with several MERGE enhancements.

The MERGE statement has two new improvements that allow you to:

- Specify the update operation or the insert operation, or both. These new conditional clauses and extensions to the standard MERGE statement make it easier to use and faster to execute:
 - Optional UPDATE or INSERT clauses
 - Conditional UPDATE for a MERGE
 - Compile time recognition and evaluation of ON clause predicates
- Delete rows from the target table during the update operation with an optional DELETE clause

MERGE Extensions

With the MERGE extensions, you have the option to:

- Omit the UPDATE or INSERT clauses

```
CREATE TABLE cust_archive AS
  SELECT * FROM customers
  WHERE 1=2;
MERGE INTO cust_archive ca
  USING customers c
  ON (ca.cust_id = c.cust_id)
WHEN MATCHED THEN
  UPDATE SET
    ca.cust_total = c.cust_total;
```

- Allow conditional updates
- Recognize special ON conditions
- Perform conditional inserts
- Use the DELETE clause



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You may want to have only UPDATE or INSERT operations on certain tables. To do so, you can omit the UPDATE or INSERT of a MERGE statement.

For example, there may be a change to the source table that you want to MERGE into the destination table. If the CUSTOMERS table changes as follows:

```
UPDATE customers
  SET cust_total = 'Excellent'
  WHERE cust_credit_limit > 10000;
```

You may want to reflect the updated CUST_TOTAL values in the destination (CUST_ARCHIVE) table.

In the example, the INSERT statement is omitted. You can have an INSERT statement present and omit the UPDATE statement too.

However, because there is no WHERE clause, 55000 rows are merged (which is the count of rows in the CUSTOMERS table).

Allowing Conditional Updates

Use the WHERE clause to conditionally update:

```
-- return environment back to original state
ROLLBACK;

MERGE INTO cust_archive  ca
  USING customers c
  ON (ca.cust_id = c.cust_id)
WHEN MATCHED THEN
  UPDATE SET
    ca.cust_total = c.cust_total
  WHERE c.cust_total = 'Excellent';

4805 rows merged.
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You can add a WHERE clause to the UPDATE operation to test a condition when it is desirable to skip the UPDATE operation.

For example, when merging information into a customer archive table, you may want to add information on excellent customers. The statement shown in the slide includes a WHERE clause that causes the UPDATE statement to occur only for excellent customers.

Using Conditional Updates

Create the sample tables and turn the timing on:

```
CREATE TABLE c1  
AS SELECT cust_id, cust_first_name, cust_last_name  
FROM customers;
```

```
CREATE TABLE c2  
AS SELECT cust_id, cust_first_name, cust_last_name  
FROM customers;
```

```
SET TIMING ON
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Using the conditional UPDATE, you can suppress all updates where the data has not changed. This minimizes the DML operations and associated REDO tasks and increases performance.

The example shown in the slide is in an extreme case, using the two tables shown.

The SQL*Plus SET TIMING command is turned on to show the differences in performance.

In the following slides, these two tables are used for conditional updates to show the impact on efficiency.

Using Conditional Updates

First example: Conditional update is not used.

```
MERGE INTO c1
  USING c2
  ON (c1.cust_id = c2.cust_id)
WHEN MATCHED THEN
  UPDATE SET c1.cust_first_name = c2.cust_first_name,
             c1.cust_last_name   = c2.cust_last_name
WHEN NOT MATCHED THEN
  INSERT VALUES(c2.cust_id, c2.cust_first_name,
                c2.cust_last_name);

55500 rows merged.
Elapsed: 00:00:00.453

-- return environment back to original state
ROLLBACK;
```

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

This first example does not use conditional updates: 55,500 rows are updated so that the first and last names of `c1` are set to `c2`. In the next slide, conditional updates are used and you see significant performance improvements.

Using Conditional Updates

Second example: Conditional update is used.

```
MERGE INTO c1
  USING c2
  ON (c1.cust_id = c2.cust_id)
WHEN MATCHED THEN
  UPDATE SET c1.cust_first_name = c2.cust_first_name,
             c1.cust_last_name = c2.cust_last_name
  WHERE (c1.cust_first_name <> c2.cust_first_name OR
         c1.cust_last_name <> c2.cust_last_name)
WHEN NOT MATCHED THEN
  INSERT VALUES(c2.cust_id, c2.cust_first_name,
                c2.cust_last_name);

0 rows merged.
Elapsed: 00:00:00.050

ROLLBACK;
```

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In the example shown in the slide, an optional WHERE clause is added to the UPDATE statement. The statement is checked to see whether there is a difference between the names in c1 and c2 tables. This results in no rows being merged. The elapsed time is much shorter because no DML operation is taking place and thus no REDO is generated.

Using the DELETE Clause to Cleanse Data

```
MERGE INTO cust_archive ca
  USING customers c
  ON (ca.cust_id = c.cust_id)
WHEN MATCHED THEN
  UPDATE SET
    ca.cust_total = c.cust_total
  WHERE c.cust_total = 'Excellent'
  DELETE WHERE (ca.cust_total = 'Customer total')
WHEN NOT MATCHED THEN
  INSERT VALUES(c.cust_id, c.cust_first_name,
    c.cust_last_name, c.cust_gender,
    c.cust_year_of_birth,
    ... c.cust_valid );

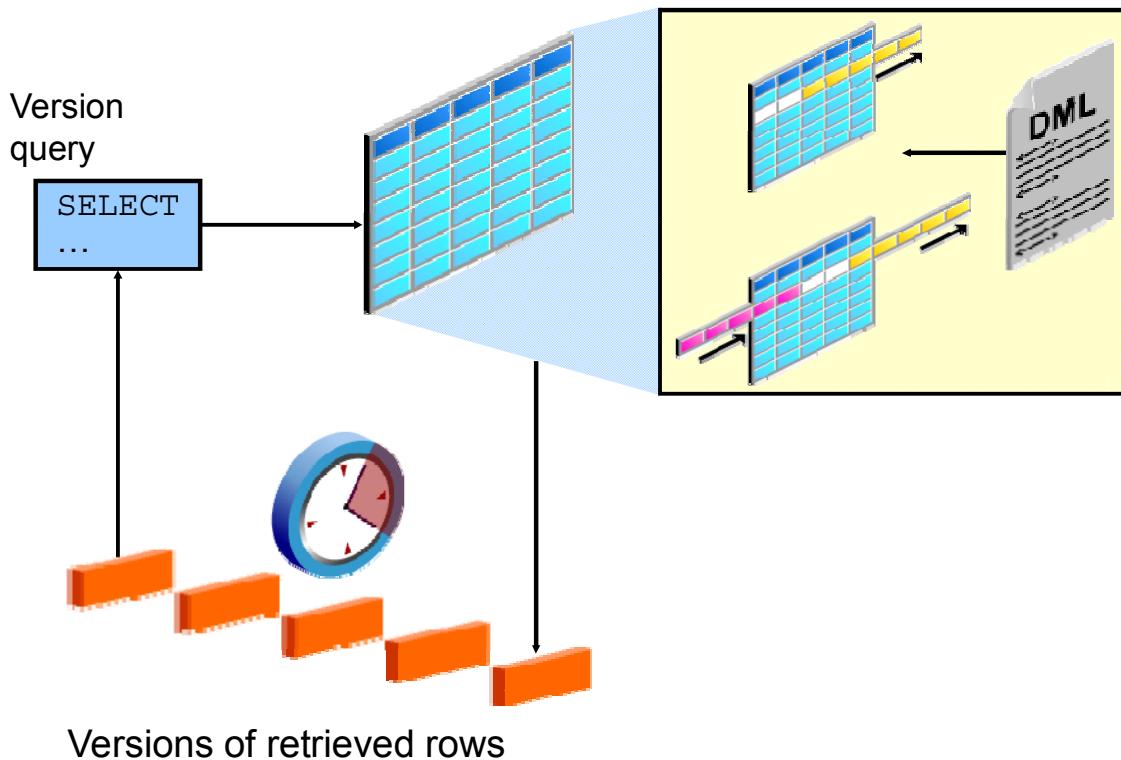
4805 rows merged.
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

To cleanse the tables while populating or updating them, you can use the optional `DELETE` clause in the `MERGE` statement. The statement shown uses the `DELETE` clause to remove rows whose product status is obsolete while performing the `UPDATE` operation. This results in cleansing of the data in the destination tables.

Tracking Changes in Data



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You may discover that somehow data in a table has been inappropriately changed. To research this, you can use multiple flashback queries to view row data at specific points in time. More efficiently, you can use the Flashback Version Query feature to view all changes to a row over a period of time. This feature enables you to append a `VERSIONS` clause to a `SELECT` statement that specifies a system change number (SCN) or the time stamp range within which you want to view changes to row values. The query also can return associated metadata, such as the transaction responsible for the change. Further, after you identify an erroneous transaction, you can use the Flashback Transaction Query feature to identify other changes that were done by the transaction. You then have the option of using the Flashback Table feature to restore the table to a state before the changes were made.

You can use a query on a table with a `VERSIONS` clause to produce all the versions of all the rows that exist or ever existed between the time the query was issued and the `undo_retention` seconds before the current time. `undo_retention` is an initialization parameter, which is an autotuned parameter. A query that includes a `VERSIONS` clause is referred to as a version query. The results of a version query behave as though the `WHERE` clause was applied to the versions of the rows. The version query returns versions of the rows only across transactions. The Oracle server assigns an SCN to identify the redo records for each committed transaction.

Flashback Version Query: Example

```
SELECT salary FROM employees3
WHERE employee_id = 107;
```

1

```
UPDATE employees3 SET salary = salary * 1.30
WHERE employee_id = 107;
```

2

```
COMMIT;
```

```
SELECT salary FROM employees3
VERSIONS BETWEEN SCN MINVALUE AND MAXVALUE
WHERE employee_id = 107;
```

3

1

	SALARY
1	4200

3

	SALARY
1	5460
2	4200

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In the example in the slide, the salary for employee 107 is retrieved (1). The salary for employee 107 is increased by 30 percent and this change is committed (2). The different versions of salary are displayed (3).

The VERSIONS clause does not change the plan of the query. For example, if you run a query on a table that uses the index access method, the same query on the same table with a VERSIONS clause continues to use the index access method. The versions of the rows returned by the version query are versions of the rows across transactions. The VERSIONS clause has no effect on the transactional behavior of a query. This means that a query on a table with a VERSIONS clause still inherits the query environment of the ongoing transaction.

The default VERSIONS clause can be specified as VERSIONS BETWEEN {SCN | TIMESTAMP} MINVALUE AND MAXVALUE.

The VERSIONS clause is a SQL extension only for queries. You can have DML and DDL operations that use a VERSIONS clause within subqueries. The row version query retrieves all the committed versions of the selected rows. Changes made by the current active transaction are not returned. The version query retrieves all incarnations of the rows. This essentially means that versions returned include deleted and subsequent reinserted versions of the rows.

The row access for a version query can be defined in one of the following two categories:

- **ROWID-based row access:** In case of ROWID-based access, all versions of the specified ROWID are returned irrespective of the row content. This essentially means that all versions of the slot in the block indicated by the ROWID are returned.
- **All other row access:** For all other row access, all versions of the rows are returned.

The definition of the employees3 table is as follows:

```
create table employees3 as select * from employees;  
SELECT salary FROM employees3  
    WHERE employee_id = 107;
```

VERSIONS BETWEEN Clause

```
SELECT versions_starttime "START_DATE",
       versions_endtime    "END_DATE",
       salary
  FROM employees
 WHERE last_name = 'Lorentz';
      VERSIONS BETWEEN SCN MINVALUE
                  AND MAXVALUE
```

START_DATE	END_DATE	SALARY
1 10-JUL-07 10.55.49.000000000 (null)		5460
2 (null)	10-JUL-07 10.55.49.000000000	4200



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You can use the VERSIONS BETWEEN clause to retrieve all the versions of the rows that exist or have ever existed between the time the query was issued and a point back in time.

If the undo retention time is less than the lower bound time or the SCN of the BETWEEN clause, the query retrieves versions up to the undo retention time only. The time interval of the BETWEEN clause can be specified as an SCN interval or a wall-clock interval. This time interval is closed at both the lower and the upper bounds.

In the example, Lorentz's salary changes are retrieved. The NULL value for END_DATE for the first version indicates that this was the existing version at the time of the query. The NULL value for START_DATE for the last version indicates that this version was created at a time before the undo retention time.

Summary

In this appendix, you should have learned how to:

- Use DML statements and control transactions
- Describe the features of multitable INSERTS
- Use the following types of multitable INSERTS:
 - Unconditional INSERT
 - Pivoting INSERT
 - Conditional INSERT ALL
 - Conditional INSERT FIRST
- Merge rows in a table
- Manipulate data by using subqueries
- Track the changes to data over a period of time



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In this appendix, you should have learned how to manipulate data in the Oracle database by using subqueries. You also should have learned about multitable INSERT statements, the MERGE statement, and tracking changes in the database.

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

Densifying Data and Performing Time Period Comparison

ORACLE®

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Objectives

After completing this appendix, you should be able to use time series to perform the following:

- Densification of data
- Period-to-period comparison for one time level
- Period-to-period comparison for multiple time levels



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Densifying Data with Partitioned Outer Joins

The outer join syntax provides high performance and enables you to fill in the gaps in sparse data:

Requirement	Consistent set of dimension values	To specify comparison calculations and format reports reliably, it is best to return a consistent set of dimension values in query results with “dense” data.
Reality	Sparse data	Data is normally stored in a sparse form (that is nonevents, such as snowshoes sold in Florida, are not stored).
Problem	Current SQL lacking	Complex and slow SQL is needed to add back rows for nonexistent cases into the query output.



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Data is normally stored in sparse form—that is, if no value exists for a given combination of dimensions, no row exists in the fact table. However, some calculations and report formatting can be performed most easily when there is a row for each combination of dimensions. This is called dense data. Dense data returns a consistent number of rows for each group of dimensions, which makes it simple to use the SQL analytic functions with physical offsets such as `LAG()` and `LEAD()`.

You can use a `PARTITION OUTER JOIN` clause to make sparse data dense.

Densifying Data with Partitioned Outer Joins

Solution?	You add a simple extension to the Outer Join syntax: Add the phrase "PARTITION BY (...)" to the join clause.
What does it do?	It is similar to a regular outer join, except that the outer join is applied to each partition.
Where to use it?	It is most frequently used to replace missing values along the time dimension, but it can be useful for any dimension.
Is it a standard?	It is accepted by ANSI and ISO for SQL standard.



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

To overcome the problem of sparsity, you can use a partitioned outer join to fill the gaps in a dimension. Such a join extends the conventional outer join syntax by applying the outer join to each logical partition defined in a query. The Oracle database logically partitions the rows in your query based on the expression that you specify in the PARTITION BY clause. The result of a partitioned outer join is a UNION of the outer joins of each of the groups in the logically partitioned table with the table on the other side of the join.

The Partitioned Outer Join Syntax

Two forms are available:

```
SELECT select_expression
FROM table_reference
PARTITION BY (expr [, expr]...)
RIGHT OUTER JOIN table_reference
```

```
SELECT select_expression
FROM table_reference
LEFT OUTER JOIN table_reference
PARTITION BY {expr [, expr]...})
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You can use either of the two forms available to perform a partitioned outer join.

Note: Execute the code examples using the SH schema.

Sparse Data Sample

NAME	YEAR	WEEK	SALES
Bounce	2000	20	80.00
Bounce	2000	21	4062.24
Bounce	2000	22	2043.16
Bounce	2000	23	2731.14
Bounce	2000	24	4419.36
Bounce	2000	27	2297.29
Bounce	2000	28	1443.13
Bounce	2000	29	1927.38
Bounce	2000	30	1927.38
Bounce	2001	20	1483.30
Bounce	2001	21	4184.49
Bounce	2001	22	2609.19
Bounce	2001	23	1416.95
Bounce	2001	24	3149.62
Bounce	2001	25	2645.98
Bounce	2001	27	2125.12
Bounce	2001	29	2467.92
Bounce	2001	30	2620.17
18 rows selected.			

Data is missing
for weeks 25
and 26 in 2000

Data is missing
for weeks 26
and 28 in 2001

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The query below produces the data shown in the slide. The query requests data for one product in a time range of 11 weeks in two different years. In this example, you would expect 22 rows of data (11 weeks each from two years) if the data was dense. However, you see only 18 rows because weeks 25 and 26 are missing in 2000 and weeks 26 and 28 are missing in 2001.

```
SELECT
    SUBSTR(p.Prod_Name,1,15) AS Product_Name,
    t.Calendar_Year AS Year,
    t.Calendar_Week_Number AS Week,
    SUM(Amount_Sold) AS Sales
FROM Sales s, Times t, Products p
WHERE s.Time_id = t.Time_id
AND s.Prod_id = p.Prod_id
AND p.Prod_name IN ('Bounce')
AND t.Calendar_Year IN (2000,2001)
AND t.Calendar_Week_Number BETWEEN 20 AND 30
GROUP BY p.Prod_Name, t.Calendar_Year, t.Calendar_Week_Number
ORDER BY week;
```

Densifying Data: Example

```
SELECT Product_Name, t.Year, t.Week, sales, NVL(Sales,0)
      dense_sales
FROM (
  SELECT SUBSTR(p.Prod_Name,1,15) Product_Name,
         t.Calendar_Year Year, t.Calendar_Week_Number Week,
         SUM(Amount_Sold) Sales
    FROM Sales s, Times t, Products p
   WHERE s.Time_id = t.Time_id
     AND s.Prod_id = p.Prod_id
     AND p.Prod_name IN ('Bounce')
     AND t.Calendar_Year IN (2000,2001)
     AND t.Calendar_Week_Number BETWEEN 20 AND 30
    GROUP BY p.Prod_Name, t.Calendar_Year, t.Calendar_Week_Number ) v
PARTITION BY (v.Product_Name)
RIGHT OUTER JOIN
  (SELECT DISTINCT Calendar_Week_Number Week, Calendar_Year Year
   FROM Times
   WHERE Calendar_Year in (2000, 2001)
     AND Calendar_Week_Number BETWEEN 20 AND 30 ) t
ON (v.week = t.week AND v.Year = t.Year)
ORDER BY t.year, t.week;
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

To fill in the gaps in the preceding sample data shown, you can use the partitioned outer join shown in the slide.

You can take the sparse data of the query and perform a partitioned outer join with a dense set of time data. In the query shown in the slide, the original query is aliased as v and the data retrieved from the times table is aliased as t.

Note that in the query above, a WHERE condition for weeks between 20 and 30 is placed in the inline view for the time dimension. This step reduces the number of rows handled by the outer join, thereby saving processing time.

Densifying Data: Example

PRODUCT_NAME	YEAR	WEEK	SALES	DENSE_SALES
Bounce	2000	20	801.00	801.00
Bounce	2000	21	4062.24	4062.24
Bounce	2000	22	2043.16	2043.16
...				
Bounce	2000	25		0.00
Bounce	2000	26		0.00
Bounce	2000	27	2297.29	2297.29
Bounce	2000	28	1443.13	1443.13
...				
Bounce	2001	22	2609.19	2609.19
Bounce	2001	23	1416.95	1416.95
Bounce	2001	24	3149.62	3149.62
Bounce	2001	25	2645.98	2645.98
Bounce	2001	26		0.00
Bounce	2001	27	2125.12	2125.12
Bounce	2001	28		0.00
Bounce	2001	29	2467.92	2467.92
Bounce	2001	30	2620.17	2620.17
22 rows selected.				

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Note that the partitioned outer join returns `NULL` values for nonpartitioning columns of the newly added rows (in this case, `YEAR`, `WEEK`, and `DENSE_SALES`). In this example, the `NULL` values for `DENSE_SALES` are replaced with 0 using the `NVL` function. However, your calculation requirement may be more demanding than replacing `NULL` values with 0. This is where the analytic functions (introduced in Oracle8i Release 2) and SQL `MODEL` clause (introduced in Oracle Database 10g) are useful.

In the following examples, you learn how to use partitioned outer joins in combination with analytic functions to create even more powerful analytic queries. For details about analytic functions, refer to the *Oracle Data Warehousing Guide*. It is important to understand the `LAST_VALUE` and `LAG` functions and the sliding window concept for the material that follows.

Repeating Data Values to Fill Gaps

The following is a sample `INVENTORY` table:

TIME_ID	PRODUCT	QUANT
01-APR-01	bottle	10
06-APR-01	bottle	8
01-APR-01	can	15
04-APR-01	can	11

- Inventory tables are naturally sparse.
- A change in quantity results in a row.
- The value to show is the most recent non-NULL number.



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Inventory tables are naturally sparse. Similar to sales tables, they need to only store a row for a product when there is an event. For a sales table, the event is a sale and for the inventory table, the event is a change in quantity available for a product. There are two requirements to create dense data for an `INVENTORY` table:

1. The `TIME` dimension must be made dense.
2. The `QUANT` column should hold the most recent non-NULL value in the series.

If you make the inventory's time dimension dense, you want to see a quantity value for each day. The value to output is the most recent non-NULL value. A partitioned outer join can be used to satisfy the first requirement. However, a partitioned outer join alone is not sufficient to satisfy the second requirement too. The analytic functions can be used to satisfy the second requirement. You can create and populate the `INVENTORY` table as follows:

```
CREATE TABLE inventory
( time_id      DATE,
  product       VARCHAR2(10),
  quant         NUMBER);

INSERT INTO inventory VALUES ('01-APR-01', 'bottle', 10);
INSERT INTO inventory VALUES ('06-APR-01', 'bottle', 8);
INSERT INTO inventory VALUES ('01-APR-01', 'can', 15);
INSERT INTO inventory VALUES ('04-APR-01', 'can', 11);
```

Repeating Data Values to Fill Gaps

1. Use a partitioned outer join.
2. Use the LAST_VALUE analytic function.

```

WITH v1 AS
  (SELECT time_id
   FROM times
   WHERE times.time_id BETWEEN TO_DATE('01/04/01', 'DD/MM/YY')
                           AND TO_DATE('07/04/01', 'DD/MM/YY'))
  SELECT product, time_id, quant quantity,
         LAST_VALUE(quant IGNORE NULLS)
        OVER (PARTITION BY product ORDER BY time_id) ②
         repeated quantity
  FROM
    (SELECT product, v1.time_id, quant
     FROM inventory PARTITION BY (product)
     RIGHT OUTER JOIN v1
      ON (v1.time_id = inventory.time_id)) ①
  ORDER BY 1, 2;

```

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The query shown in the slide performs the operations that are needed to densify the inventory data. The WITH clause is used to choose a one-week subset of the time dimension.

Using the WITH clause, you define a query block before using it in the query. This enables you to reuse the same query block in a SELECT statement when it occurs more than once within a complex query.

In step 1, the inner query computes a partitioned outer join on time within each product. The inner query densifies the data on the time dimension (that is, the query results will now show all the days for each product). However, the QUANTITY measure column will have NULLs for the newly added rows (see the output of the QUANTITY column in the output in the next slide).

In step 2, the outer query computes the LAST_VALUE analytic function. The expression partitions the data by product and orders each partitioned data on the time dimension column (TIME_ID). For each row, the expression finds the last non-NUL value in its window (due to the IGNORE NULLS option). This function returns the desired output for the QUANTITY column. The results are shown in the next slide.

Repeating Data Values to Fill Gaps Results

The results from the query in the previous slide are:

PRODUCT	TIME_ID	QUANTITY	REPEATED_QUANTITY
bottle	01-APR-01	10	10
bottle	02-APR-01		10
bottle	03-APR-01		10
bottle	04-APR-01		10
bottle	05-APR-01		10
bottle	06-APR-01	8	8
bottle	07-APR-01		8
can	01-APR-01	15	15
can	02-APR-01		15
can	03-APR-01		15
can	04-APR-01	11	11
can	05-APR-01		11
can	06-APR-01		11
can	07-APR-01		11

14 rows selected.

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Computing Data Values to Fill Gaps

1. Use a partitioned outer join.
2. Use the AVG analytic function.

```

WITH v AS
  (SELECT p.prod_name, calendar_month_desc,
         SUM(quantity_sold) units, SUM(amount_sold) sales
    FROM sales s, products p, times t
   WHERE s.prod_id in (122, 136) AND calendar_year = 2000
     AND t.time_id = s.time_id AND p.prod_id = s.prod_id
   GROUP BY p.prod_name, calendar_month_desc)
SELECT v.prod_name, calendar_month_desc, units, sales,
       NVL(units, AVG(units) OVER (PARTITION BY v.prod_name)) )
       AS computed_units,
       NVL(sales, AVG(sales) OVER (PARTITION BY v.prod_name)) )
       AS computed_sales
FROM
  (SELECT DISTINCT calendar_month_desc
    FROM times
   WHERE calendar_year = 2000) t
  LEFT OUTER JOIN v PARTITION BY (prod_name)
  USING (calendar_month_desc);

```

2

1

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Here is another example that combines a partitioned outer join with analytic functions. You may want to obtain the monthly totals for product IDs 122 and 136 (64 MB memory card and DVD-R discs) for the year 2000. For the months that are missing, you want to assume the sales to be the average of the other months when products were sold. This query can be computed by using the following two steps:

1. The inner query uses the partitioned outer join to fill in the missing months.
2. The outer query uses the AVG analytic function to compute average sales and units. The NVL function replaces the NULLs produced by the partitioned outer join with the AVG value computed using the analytic function.

Computing Data Values to Fill Gaps

PROD_NAME	CALENDAR_UNITS	SALES	COMPUTED_UNITS	COMPUTED_SALES
...				
64MB Memory Card 2000-05	47	1738.24	47	1738.24
64MB Memory Card 2000-06	20	739.40	20	739.40
64MB Memory Card 2000-07			73	2686.79
64MB Memory Card 2000-08			73	2686.79
64MB Memory Card 2000-09			73	2686.79
64MB Memory Card 2000-10			73	2686.79
64MB Memory Card 2000-11			73	2686.79
64MB Memory Card 2000-12			73	2686.79
DVD-R Discs, 4.7 2000-01	167	3683.50	167	3683.50
DVD-R Discs, 4.7 2000-02	152	3362.24	152	3362.24
...				
DVD-R Discs, 4.7 2000-06	145	3192.21	145	3192.21
DVD-R Discs, 4.7 2000-07			124	2737.71
DVD-R Discs, 4.7 2000-08			124	2737.91
DVD-R Discs, 4.7 2000-09	1	18.91	1	18.91
DVD-R Discs, 4.7 2000-10			124	2737.71
DVD-R Discs, 4.7 2000-11			124	2737.71
DVD-R Discs, 4.7 2000-12	8	161.84	8	161.84
24 rows selected.				

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

These are the results from the partitioned outer join shown in the previous slide. The rows that are highlighted are the rows generated from the partitioned outer join for the missing months.

Time Series Calculations on Densified Data

```
SELECT Product_Name, t.Year, t.Week, Sales,
       NVL(Sales,0) Current_sales, SUM(Sales) OVER
        (PARTITION BY Product_Name, t.year ORDER BY t.week)
        AS Cumulative_sales
FROM (SELECT SUBSTR(p.Prod_Name,1,15) Product_Name,
             t.Calendar_Year Year, t.Calendar_Week_Number Week,
             SUM(Amount_Sold) Sales
      FROM Sales s, Times t, Products p
     WHERE s.Time_id = t.Time_id AND s.Prod_id = p.Prod_id AND
           p.Prod_name IN ('Bounce') AND t.Calendar_Year IN (2000,2001)
           AND t.Calendar_Week_Number BETWEEN 20 AND 30
      GROUP BY p.Prod_Name, t.Calendar_Year, t.Calendar_Week_Number) v
PARTITION BY (v.Product_Name)
RIGHT OUTER JOIN
  (SELECT DISTINCT
            Calendar_Week_Number Week, Calendar_Year Year
       FROM Times
      WHERE Calendar_Year in (2000, 2001) AND
            Calendar_Week_Number BETWEEN 20 AND 30) t
ON (v.week = t.week AND v.Year = t.Year)
ORDER BY t.year, t.week;
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Time series calculations are easier when data is dense along the time dimension. Dense data has a consistent number of rows for each time period, which in turn makes it simple to use analytic window functions with physical offsets.

The example shown calculates weekly year-to-date sales alongside the weekly sales. The NULL values that the partitioned outer join inserts in making the time series dense are handled in the usual way; the SUM function treats them as zeroes.

Time Series Calculations: Results

PRODUCT_NAME	YEAR	WEEK	SALES	CURRENT_SALES	CUMULATIVE_SALES
Bounce	2000	20	801.00	801.00	801.00
Bounce	2000	21	4062.24	4062.24	4863.24
Bounce	2000	22	2043.16	2043.16	6906.40
Bounce	2000	23	2731.14	2731.14	9637.54
Bounce	2000	24	4419.36	4419.36	14056.90
Bounce	2000	25		0.00	14056.90
Bounce	2000	26		0.00	14056.90
Bounce	2000	27	2297.29	2297.29	16354.19
Bounce	2000	28	1443.13	1443.13	17797.32
...					
Bounce	2001	23	1416.95	1416.95	9693.93
Bounce	2001	24	3149.62	3149.62	12843.55
Bounce	2001	25	2645.98	2645.98	15489.53
Bounce	2001	26		0.00	15489.53
Bounce	2001	27	2125.12	2125.12	17614.65
Bounce	2001	28		0.00	17614.65
Bounce	2001	29	2467.92	2467.92	20082.57
Bounce	2001	30	2620.17	2620.17	22702.74
22 rows selected.					

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

These are the results from the partitioned outer join shown in the previous slide. The highlighted rows show the NULL values that the partitioned outer join inserted. In the CURRENT_SALES column, the NVL function converts NULL to zero.

Period-to-Period Comparison of One Time Level

Calculating a year-over-year sales comparison at the week level results in the following:

PROD	YEAR	WEEK	SALES	WEEKLY_YTD_SALES	
				WEEKLY_YTD_SALES	PRIOR_YEAR
Y_Box	2001	30	7877.45	7877.45	0.00
Y_Box	2001	31	13082.46	20959.91	1537.35
Y_Box	2001	32	11569.02	32528.93	9531.57
Y_Box	2001	33	38081.97	70610.90	39048.69
Y_Box	2001	34	33109.65	103720.55	69100.79
Y_Box	2001	35	0.00	103720.55	71265.35
Y_Box	2001	36	4169.30	107889.85	81156.29
Y_Box	2001	37	24616.85	132506.70	95433.09
Y_Box	2001	38	37739.65	170246.35	107726.96
Y_Box	2001	39	284.95	170531.30	118817.40
Y_Box	2001	40	10868.44	181399.74	120969.69
11 rows selected.					



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Suppose you are interested in calculating a year-over-year Year-To-Date (YTD) sales comparison at the week level for the Y_Box product for the years 2001 versus 2002. This is done in three steps:

1. Using a partitioned outer join, data can be densified along the time dimension (in this case, at the week level).
2. The `SUM()` analytic function can be used to compute Year-To-Date Totals at the week level.
3. The `LAG` analytic function can be used to reach over and obtain the Year-To-Date Totals for the same week of the prior year. Because data has been densified on the time dimension (at the week level), the `LAG` function easily gets the sales for the prior year even though it uses physical offsets. See the query on the following page for details. To improve readability of the query and focus on the partitioned outer join, a `WITH` clause is used to start the query.

```

WITH v AS (
    SELECT p.Prod_Name Product_Name, t.Calendar_Year Year,
           t.Calendar_Week_Number Week, SUM(Amount_Sold) Sales
      FROM Sales s, Times t, Products p
     WHERE s.Time_id = t.Time_id AND s.Prod_id = p.Prod_id
       AND p.Prod_name in ('Y Box')
       AND t.Calendar_Year in (2000,2001)
       AND t.Calendar_Week_Number BETWEEN 30 AND 40
    GROUP BY p.Prod_Name, t.Calendar_Year,
             t.Calendar_Week_Number)
SELECT substr(Product_Name,1,12) Prod, Year, Week, Sales,
        Weekly_ytd_sales, Weekly_ytd_sales_prior_year
   FROM (SELECT --Start of year_over_year sales
            Product_Name, Year, Week, Sales,
            Weekly_ytd_sales, LAG(Weekly_ytd_sales, 1)
          OVER
         (PARTITION BY Product_Name, Week
            ORDER BY Year) Weekly_ytd_sales_prior_year
   FROM (SELECT --Start of dense_sales
            v.Product_Name Product_Name, t.Year Year,
            t.Week Week, NVL(v.Sales,0) Sales,
            SUM(NVL(v.Sales,0))
          OVER
         (PARTITION BY v.Product_Name, t.Year
            ORDER BY t.week) weekly_ytd_sales
      FROM v
            PARTITION BY (v.Product_Name)
      RIGHT OUTER JOIN (
            SELECT DISTINCT Calendar_Week_Number Week,
                           Calendar_Year Year
              FROM Times
             WHERE Calendar_Year IN (2000, 2001) ) t
            ON (v.week = t.week AND v.Year = t.Year)
      ) dense_sales
   ) year_over_year_sales
  WHERE Year = 2001
    AND Week BETWEEN 30 AND 40
  ORDER BY 1, 2, 3;

```

In the FROM clause of the DENSE_SALES inline view, a partitioned outer join of aggregate view v and the time view t is used to fill gaps in the sales data along the time dimension. The output of the partitioned outer join is then processed by the SUM ... OVER analytic function to compute the weekly YTD sales (the weekly_ytd_sales column).

Thus, the DENSE_SALES view computes the YTD sales data for each week, including those missing in the aggregate view.

The `YEAR_OVER_YEAR_SALES` inline view then computes the weekly YTD sales of the prior year using the `LAG` function. The `LAG` function, labeled `weekly_ytd_sales_prior_year`, specifies a `PARTITION BY` clause that pairs rows for the same week of the years 2000 and 2001 into a single partition. An offset of 1 is passed to the `LAG` function to get the weekly YTD sales for the prior year.

The outermost query block selects data from `YEAR_OVER_YEAR_SALES` with the condition `year = 2001`, and thus the query returns, for each product, its weekly YTD sales in the specified weeks of the years 2001 and 2000.

Period-to-Period Comparison for Multiple Time Levels: Example

Goal:

Create a single query with comparisons at the day, week, month, quarter, and year level.

Approach:

1. View of the TIME dimension
2. View of SALES aggregated over TIME and PRODUCT
3. Materialized view of SALES aggregated over TIME and PRODUCT (for performance)



ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

It is useful to handle multiple time levels in a single query. For instance, you can compare sales versus the prior period at the year, quarter, month, and day levels.

One approach is to use the three steps mentioned as follows. The goal is a single query with comparisons at the day, week, month, quarter, and year level.

1. You create a view of the TIME dimension to use as an edge of the cube. The time edge will be partition outer joined to the sparse data in the CUBE_PROD_TIME view to densify the data.
2. You create a view called CUBE_PROD_TIME that holds a hierarchical cube of sales aggregated across TIMES and PRODUCTS.
3. For optimal performance, you create a materialized view called MV_PROD_TIME. This materialized view matches the view defined in step 1.

Note: To learn about the hierarchical cube concept, refer to the *Data Warehousing Reference Guide*.

View of the TIME Dimension

```
CREATE OR REPLACE VIEW edge_time AS
SELECT
  (CASE WHEN ((GROUPING(calendar_year)=0 )AND
              (GROUPING(calendar_quarter_desc)=1 ))
        THEN (TO_CHAR(calendar_year) || '_0')
        WHEN ((GROUPING(calendar_quarter_desc)=0) AND
              (GROUPING(calendar_month_desc)=1 ))
        THEN (TO_CHAR(calendar_quarter_desc) || '_1')
        WHEN ((GROUPING(calendar_month_desc)=0 )AND
              (GROUPING(time_id)=1 ))
        THEN (TO_CHAR(calendar_month_desc) || '_2')
        ELSE (TO_CHAR(time_id) || '_3') END) Hierarchical_Time,
  calendar_year yr, calendar_quarter_number qtr_num,
  calendar_quarter_desc qtr, calendar_month_number
  mon_num, calendar_month_desc mon,
  time_id - TRUNC(time_id, 'YEAR') + 1 day_num,
  time_id day,
  GROUPING_ID(calendar_year, calendar_quarter_desc,
              calendar_month_desc, time_id) gid_t
FROM TIMES
GROUP BY ROLLUP (calendar_year,
                  (calendar_quarter_desc, calendar_quarter_number),
                  (calendar_month_desc, calendar_month_number), time_id);
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The CREATE VIEW statement shown in the slide defines a view of the TIME table that is used in a partitioned outer join with sales data.

Note that the HIERARCHICAL_TIME column contains string representations of time from all levels of the time hierarchy. This is analogous to creating a primary key in a single column across all levels of time. The CASE expression used for the HIERARCHICAL_TIME column appends a marker (_0, _1, ...) to each date string to denote the time level of the value. A _0 represents the year level, _1 represents quarters, _2 represents months, and _3 represents day.

Note that the GROUP BY ROLLUP clause is a rollup hierarchy for the TIME dimension.

View of the SALES Cube

```

CREATE OR REPLACE VIEW cube_prod_time
AS SELECT
(CASE
    WHEN ((GROUPING(calendar_year)=0) AND (GROUPING(calendar_quarter_desc)=1))
    THEN (TO_CHAR(calendar_year) || '_0')
    WHEN ((GROUPING(calendar_quarter_desc)=0) AND
          (GROUPING(calendar_month_desc)=1))
    THEN (TO_CHAR(calendar_quarter_desc) || '_1')
    WHEN ((GROUPING(calendar_month_desc)=0) AND (GROUPING(t.time_id)=1))
    THEN (TO_CHAR(calendar_month_desc) || '_2')
    ELSE (TO_CHAR(t.time_id) || '_3') END) Hierarchical_Time,
    calendar_year year, calendar_quarter_desc quarter,
    calendar_month_desc month, t.time_id day, prod_category cat, prod_subcategory
    subcat, p.prod_id prod,
    GROUPING_ID(prod_category, prod_subcategory, p.prod_id, calendar_year,
                calendar_quarter_desc, calendar_month_desc, t.time_id) gid,
    GROUPING_ID(prod_category, prod_subcategory, p.prod_id) gid_p,
    GROUPING_ID(calendar_year, calendar_quarter_desc, calendar_month_desc, t.time_id)
        gid_t,
    SUM(amount_sold) s_sold, COUNT(amount_sold) c_sold, COUNT(*) cnt
FROM SALES s, TIMES t, PRODUCTS p
WHERE s.time_id = t.time_id AND
      p.prod_name IN ('Bounce', 'Y Box') AND
      s.prod_id = p.prod_id
GROUP BY ROLLUP(calendar_year, calendar_quarter_desc, calendar_month_desc,
                 t.time_id), ROLLUP(prod_category, prod_subcategory, p.prod_id);

```

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The view in the slide creates a sparsely populated hierarchical cube of sales data.

Note that the HIERARCHICAL_TIME column contains string representations of time from all levels of the time hierarchy. The CASE expression used for the Hierarchical_Time column appends a marker (_0, _1, ...) to each date string to denote the time level of the value. A _0 represents the year level, _1 represents quarters, _2 represents months, and _3 represents day. Note that the GROUP_BY clause is a concatenated ROLLUP that specifies the rollup hierarchy for the TIME and PRODUCT dimensions. The GROUP_BY clause determines the hierarchical cube contents. Note that the GROUPING_ID function is used to create three sets of groups:

- gid covers the full product and time hierarchy.
- gid_p covers the product hierarchy.
- gid_t covers the time hierarchy.

In this example, the gid_t is used in the partitioned outer join.

Materialized View of SALES

```
CREATE MATERIALIZED VIEW mv_prod_time
REFRESH COMPLETE ON DEMAND AS
SELECT (CASE
    WHEN ((GROUPING(calendar_year)=0 ) AND
          (GROUPING(calendar_quarter_desc)=1 ))
    THEN (TO_CHAR(calendar_year) || '_0')
    WHEN ((GROUPING(calendar_quarter_desc)=0 ) AND
          (GROUPING(calendar_month_desc)=1 ))
    THEN (TO_CHAR(calendar_quarter_desc) || '_1')
    WHEN ((GROUPING(calendar_month_desc)=0 ) AND
          (GROUPING(t.time_id)=1 ))
    THEN (TO_CHAR(calendar_month_desc) || '_2')
    ELSE (TO_CHAR(t.time_id) || '_3') END) Hierarchical_Time,
       calendar_year year, calendar_quarter_desc quarter,
       calendar_month_desc month, t.time_id day, prod_category cat,
       prod_subcategory subcat, p.prod_id prod,
       GROUPING_ID(prod_category, prod_subcategory, p.prod_id, calendar_year,
                  calendar_quarter_desc, calendar_month_desc,t.time_id) gid,
       GROUPING_ID(prod_category, prod_subcategory, p.prod_id) gid_p,
       GROUPING_ID(calendar_year, calendar_quarter_desc, calendar_month_desc, t.time_id)
          gid_t, SUM(amount_sold) s_sold,
       COUNT(amount_sold) c_sold, COUNT(*) cnt
FROM SALES s, TIMES t, PRODUCTS p
WHERE s.time_id = t.time_id AND p.prod_name in ('Bounce', 'Y Box') AND
      s.prod_id = p.prod_id
GROUP BY
       ROLLUP(calendar_year, calendar_quarter_desc, calendar_month_desc, t.time_id),
       ROLLUP(prod_category, prod_subcategory, p.prod_id);
```



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

The MV_PROD_TIME materialized view is created with the same definition as the CUBE_PROD_TIME view.

Note: The materialized view is created purely for performance.

Period-to-Period Query

```

SELECT substr(prod,1,4) prod, substr(Hierarchical_Time,1,12) ht,
       sales, sales_prior_period, sales - sales_prior_period
       variance_prior_period, sales_same_period_prior_year,
       sales - sales same period prior year variance same period p year
FROM ( SELECT cat, subcat, prod, gid_p, gid_t, Hierarchical_Time, yr, qtr,
            mon, day, sales,
            LAG(sales, 1) OVER
                (PARTITION BY gid_p, cat, subcat, prod, gid_t
                 ORDER BY yr, qtr, mon, day) sales_prior_period,
            LAG(sales, 1 ) OVER
                (PARTITION BY gid_p, cat, subcat, prod, gid_t,
                           qtr_num, mon_num, day_num
                           ORDER BY yr) sales same period prior year
      FROM (
        SELECT c.gid, c.cat, c.subcat, c.prod, c.gid_p, t.gid_t, t.yr,
               t.qtr, t.qtr_num, t.mon, t.mon_num, t.day, t.day_num,
               t.Hierarchical_Time, NVL(s_sold,0) sales
        FROM cube prod time c
        PARTITION BY (gid_p, cat, subcat, prod)
        RIGHT OUTER JOIN edge_time t ON ( c.gid_t = t.gid_t
                                         AND c.Hierarchical Time = t.Hierarchical Time)
        ) dense_cube_prod_time
      ) -- side by side current,prior and prior year
WHERE prod IN (139) AND gid_p=0 AND
      ( (mon IN ('2001-08') AND gid_t IN (0, 1)) OR -- day and month data
        (qtr IN ('2001-03') AND gid_t IN (3)) )          -- quarter level data
ORDER BY day;

```

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

You now have the required elements for the comparison query. You can obtain period-to-period comparison calculations at all time levels. It requires applying analytic functions to a hierarchical cube with dense data along the time dimension. Some of the calculations that you can achieve for each time level are the following:

- Sum of sales for the previous period at all levels of time
- Variance in sales over the previous period
- Sum of sales in the same period a year ago at all levels of time
- Variance in sales over the same period last year

The query in the slide performs all four of these calculations. It uses a partitioned outer join of the CUBE_PROD_TIME and EDGE_TIME views to create an inline view of dense data called DENSE_CUBE_PROD_TIME (1). The query then uses the LAG function in the same way as the earlier single-level example. The outer WHERE clause specifies time at three levels: the days of August 2001, the entire month, and the entire third quarter of 2001.

Note that the last two rows of the results contain the month-level and quarter-level aggregations.

The first `LAG` function (`sales_prior_period`) orders the rows on the time dimension columns (2), `YR`, `QTR`, `MON`, and `DAY`, and gets the sales value of the prior period by passing an offset of 1. The second `LAG` function (`sales_same_period_prior_year`) partitions the data on additional columns `QTR_YR`, `MON_NUM`, and `DAY_NUM` and orders it on `YR` so that, with an offset of 1, it can compute the year-ago sales for the same period (3). The outermost `SELECT` clause computes the variances.

Period-to-Period Query Results

PROD HT	SALES	variance		sales_same		variance	
		sales_prior	period	prior	period	period_prior	_same_period
139 01-AUG-01_3	0.00	0.00		0.00		0.00	0.00
139 02-AUG-01_3	1347.53	0.00		1347.53		0.00	1347.53
...							
139 15-AUG-01_3	38.49	0.00		38.49		1104.55	-1066.06
139 16-AUG-01_3	0.00	38.49		-38.49		0.00	0.00
139 17-AUG-01_3	77.17	0.00		77.17		1052.03	-974.86
139 18-AUG-01_3	2467.54	77.17		2390.37		0.00	2467.54
139 19-AUG-01_3	0.00	2467.54		-2467.54		127.08	-127.08
139 20-AUG-01_3	0.00	0.00		0.00		0.00	0.00
139 21-AUG-01_3	0.00	0.00		0.00		0.00	0.00
139 22-AUG-01_3	0.00	0.00		0.00		0.00	0.00
139 23-AUG-01_3	1371.43	0.00		1371.43		0.00	1371.43
139 24-AUG-01_3	153.96	1371.43		-1217.47		2091.30	-1937.34
139 25-AUG-01_3	0.00	153.96		-153.96		0.00	0.00
139 26-AUG-01_3	0.00	0.00		0.00		0.00	0.00
139 27-AUG-01_3	1235.48	0.00		1235.48		0.00	1235.48
139 28-AUG-01_3	173.30	1235.48		-1062.18		2075.64	-1902.34
139 29-AUG-01_3	0.00	173.30		-173.30		0.00	0.00
139 30-AUG-01_3	0.00	0.00		0.00		0.00	0.00
139 31-AUG-01_3	0.00	0.00		0.00		0.00	0.00
139 2001-08_2	8347.43	7213.21		1134.22		8368.98	-21.55
139 2001-03_1	24356.80	28862.14		-4505.34		24168.99	187.81
33 rows selected.							

ORACLE

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

Note

To make the results easier to read, the SQL*Plus column headings are adjusted with the following commands. The commands fold the column headings to reduce line length:

```
col sales_prior_period heading 'sales_prior|_period'
col variance_prior_period heading 'variance|_prior|_period'
col sales_same_period_prior_year -
heading 'sales_same|_period_prior|_year'
col variance_same_period_p_year -
heading 'variance|_same_period|_prior_year'
```

Summary

In this appendix, you should have learned how to use time series to perform the following:

- Densification of data
- Period-to-period comparison on one time level
- Period-to-period comparison for multiple time levels



Copyright © 2013, Oracle and/or its affiliates. All rights reserved.

In this appendix, you learned about the enhancements to the `MERGE` statement. You learned that the `MERGE` statement can have either an `UPDATE` clause or an `INSERT` clause, or both. You also learned that you can perform conditional updates and inserts to reduce the number of rows processed. With the `DELETE` clause, you can cleanse the data by removing unnecessary rows.

You also learned about the enhancements on partitioned outer joins. These enhancements enable you to write statements that densify data and perform period-to-period analysis.