SQL Fundamentals

Student Guide
X95174GC10
Edition 1.0 | May 2016

Learn more from Oracle University at **oracle.com/education/**

Authors

Apoorva Srinivas

Puja Singh

Technical Contributors and Reviewers

Nancy Greenberg

Suresh Rajan

Satyajit Ranganathan

Gururaj Bs

Editors

Chandrika Kennedy

Vijayalakshmi Narasimhan

Raj Kumar

Graphic Editors

Kavya Bellur

Prakash Dharmalingam

Maheshwari Krishnamurthy

Publishers

Pavithran Adka
Asief Baig

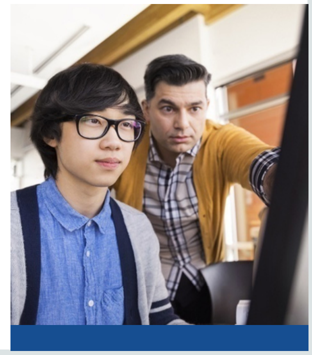
Contents


Lesson 1: Introduction
ORACLE
Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Lesson Objectives

After completing this lesson, you should be able to:

- Define the goals of the course
- Describe the course roadmap
- List the Oracle Database documentation and additional resources



ORACLE

In this lesson, you gain an understanding of the overall objectives of the course. You learn about the roadmap followed for the course. You also learn where to find the documentation for Oracle Database 12*c*, Oracle Cloud, and SQL Developer for reference.

Lesson Agenda

- Overview of course objectives
- Overview of course roadmap
- Introduction to Oracle Cloud
- Oracle Database 12*c* SQL documentation and additional resources

ORACLE

Course Objectives

After completing this course, you should be able to:

- Identify the major components of Oracle Database
- Retrieve row and column data from tables with the `SELECT` statement
- Create reports of sorted and restricted data
- Employ SQL functions to generate and retrieve customized data
- Run complex queries to retrieve data from multiple tables
- Run data manipulation language (DML) statements to update data in Oracle Database
- Run data definition language (DDL) statements to create and manage schema objects
- Manage users with different levels of access privileges
- Use data dictionary views

This course offers you an introduction to the Oracle Database technology. In this class, you learn the basic concepts of relational databases and the powerful SQL programming language. This course provides the essential SQL skills that enable you to write queries against single and multiple tables, manipulate data in tables, create database objects, query metadata, manage users, and use data dictionary views.

Lesson Agenda

- Overview of course objectives
- Overview of course roadmap
- Introduction to Oracle Cloud
- Oracle Database 12*c* SQL documentation and additional resources

Course Roadmap

- Lesson 1: Course Overview
- **Unit 1: Relational Database and SQL Overview**
  - Lesson 2: Relational Database Overview
  - Lesson 3: Database Storage Structures
  - Lesson 4: Introduction to SQL
- Unit 2: Retrieving and Sorting Data
- Unit 3: Joins, Subqueries, and Set Operators
- Unit 4: DML and DDL
- Unit 5: Managing Relational Database
- Unit 6: Advance Queries and Database Management System

ORACLE

In Unit 1, you will learn about databases and database concepts. You will be introduced to relational databases, data storage concepts, and SQL.

Course Roadmap

- Lesson 1: Course Overview
- Unit 1: Relational Database and SQL Overview
- **Unit 2: Retrieving and Sorting Data**
  - Lesson 5: Retrieving Data Using SQL `SELECT` Statement
  - Lesson 6: Restricting and Sorting Data
  - Lesson 7: Using Single-Row Functions
  - Lesson 8: Using Conversion Functions
  - Lesson 9: Using Conditional Expressions
- Unit 3: Joins, Subqueries, and Set Operators
- Unit 4: DML and DDL
- Unit 5: Managing Relational Database
- Unit 6: Advance Queries and Database Management System

In Unit 2, you will dive into the concepts of SQL. You will learn to use the SQL `SELECT` statement to retrieve data from database tables, and restrict and sort the retrieved data. You will also learn about single-row functions, conversion functions, and conditional expressions in SQL.

Course Roadmap

- Lesson 1: Course Overview
- Unit 1: Relational Database and SQL Overview
- Unit 2: Retrieving and Sorting Data
- Unit 3: Joins, Subqueries, and Set Operators
  - Lesson 10: Reporting Aggregated Data Using the Group Functions
  - Lesson 11: Retrieving Data from Multiple Tables Using Joins
  - Lesson 12: Using the Set Operators
  - Lesson 13: Using Subqueries to Solve Queries
- Unit 4: DML and DDL
- Unit 5: Managing Relational Database
- Unit 6: Advance Queries and Database Management System

In Unit 3, you will learn about using joins, subqueries, and set operators. You will learn to write compound queries in SQL to generate customized reports using group functions, joins, and subqueries.

Course Roadmap

- Lesson 1: Course Overview
- Unit 1: Relational Database and SQL Overview
- Unit 2: Retrieving and Sorting Data
- Unit 3: Joins, Subqueries, and Set Operators
- **Unit 4: DML and DDL**
  - Lesson 14: Introduction to Data Manipulation Language
  - Lesson 15: Introduction to Data Definition Language
  - Lesson 16: Managing Tables using DML Statements
- Unit 5: Managing Relational Database
- Unit 6: Advance Queries and Database Management System

ORACLE

In Unit 4, you will learn about Data Manipulation Language (DML) and Data Definition Language (DDL). Using DML statements, you will learn to update and manage data in the tables. Using DDL statements, you will learn to create tables, remove tables, etc.

Course Roadmap

- Lesson 1: Course Overview
- Unit 1: Relational Database and SQL Overview
- Unit 2: Retrieving and Sorting Data
- Unit 3: Joins, Subqueries, and Set Operators
- Unit 4: DML and DDL
- **Unit 5: Managing Relational Database**
  - Lesson 17: Introduction to Data Dictionary Views
  - Lesson 18: Creating Views
  - Lesson 19: Creating Sequences, Synonyms and Indexes
  - Lesson 20: Managing Constraints, Temporary Tables and External Tables
- Unit 6: Advance Queries and Database Management System

ORACLE

In Unit 5, you will be introduced to views. You will also learn to query data dictionary views. You will learn to create sequences, synonyms and indexes. You will also learn to manage constraints and tables.

Course Roadmap

- Lesson 1: Course Overview
- Unit 1: Relational Database and SQL Overview
- Unit 2: Retrieving and Sorting Data
- Unit 3: Joins, Subqueries, and Set Operators
- Unit 4: DML and DDL
- Unit 5: Managing Relational Database
- Unit 6: Advance Queries and Database Management System
  - Lesson 21: Using Advanced Subqueries
  - Lesson 22: Manipulating Data by Using Advanced Subqueries
  - Lesson 23: Controlling User Access
  - Lesson 24: Advanced Data Manipulation
  - Lesson 25: Managing Multiple Timezones

ORACLE

In Unit 6, you will be introduced to some advanced features of SQL. You will learn to write advanced subqueries. You will learn to create users and manage users. You will also learn about managing multiple timezones.

Lesson Agenda

- Overview of course objectives
- Overview of course roadmap
- Introduction to Oracle Cloud
- Oracle Database 12*c* SQL documentation and additional resources

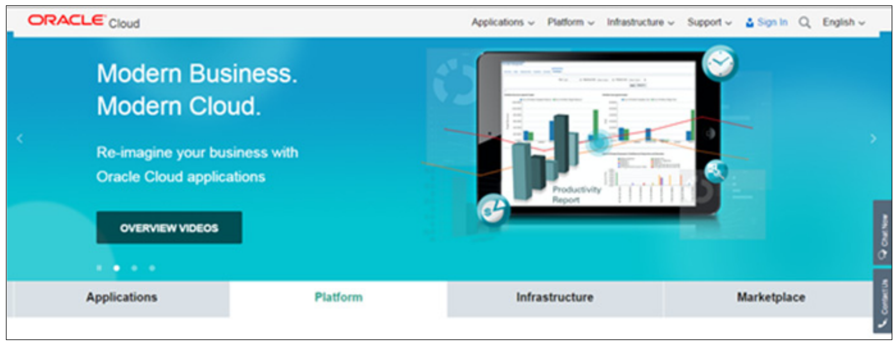
ORACLE

Introduction to Oracle Cloud

Oracle Cloud is an enterprise cloud for business. Oracle Public Cloud consists of many different services that share some common characteristics:

- On-demand self-service
- Resource pooling
- Rapid elasticity
- Measured service
- Broad network access

www.cloud.oracle.com



Oracle Cloud is an enterprise cloud for business. It provides an integrated collection of application and platform cloud services that are based on best-in-class products and open Java and SQL standards.

As a result, the applications and databases that are deployed in Oracle Cloud are portable and can be easily moved to or from a private cloud or an on-premise (local machine) environment.

- All Cloud Services can be provisioned through a self-service interface. Users can get their Cloud Services delivered on an integrated development and deployment platform with tools to rapidly extend and create new services. Oracle Cloud services are built on the Oracle Exalogic Elastic Cloud and Oracle Exadata Database Machine, which together offer a platform that delivers extreme performance, redundancy, and scalability. The top two benefits of cloud computing are speed and cost.

The five essential characteristics are:

- **On-demand self-service:** Provisioning, monitoring, management control
- **Resource pooling:** Sharing and a level of abstraction between consumers and services
- **Rapid elasticity:** The ability to quickly scale up or down as needed
- **Measured service:** Metering utilization for either internal chargeback (private cloud) or external billing (public cloud)
- **Broad network access:** Typically, access through a browser on any networked device

Oracle Cloud Services

Oracle Cloud provides the following three types of services:

- Software as a Service (SaaS)
- Platform as a Service (PaaS)
- Infrastructure as a Service (IaaS)

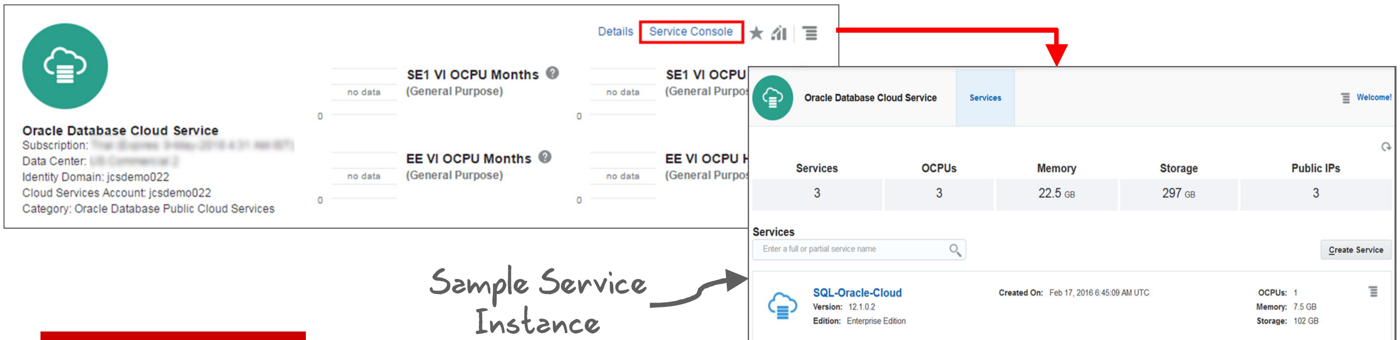
- **Software as a Service (SaaS)** generally refers to applications that are delivered to end users over the Internet. Oracle CRM On Demand is an example of a SaaS offering that provides both multitenant as well as single-tenant options, depending on the customer's preferences.
- **Platform as a Service (PaaS)** generally refers to an application development and deployment platform that is delivered as a service to developers, enabling them to quickly build and deploy a SaaS application to end users. The platform typically includes databases, middleware, and development tools, all delivered as a service via the Internet.
- **Infrastructure as a Service (IaaS)** refers to computing hardware (servers, storage, and network) delivered as a service. This typically includes the associated software as well as operating systems, virtualization, clustering, and so on. Examples of IaaS in the public cloud include Amazon's Elastic Compute Cloud (EC2) and Simple Storage Service (S3).

The database cloud is built within an enterprise's private cloud environment, as a PaaS model. The database cloud provides on-demand access to database services in a self-service, elastically scalable, and metered manner. The database cloud offers compelling advantages in cost, quality of service, and agility. A database can also be deployed within a virtual machine on an IaaS platform.

Database clouds can be rapidly deployed on Oracle Exadata, a pre-integrated and optimized hardware platform that supports both online transaction processing (OLTP) and data warehouse (DW) workloads.

Database on Oracle Cloud

- Oracle Database Cloud Service
  - It is implemented as a PaaS.
  - The service is identified by an identity domain.
  - You can sign in, and create a new Database Service Instance to start using the Cloud Database memory.



ORACLE

- Oracle Database Cloud Service is a service offered as a PaaS.
- When an Oracle Database Cloud Service is purchased or is requested as a trial version, you receive a welcome email from the Oracle Cloud team with the following details:
  - Service Identity Domain
  - User ID
  - Temporary password that must be changed upon first sign-in
- When you sign in to the service, you see the “My Services” page with options to manage the service and its users.
- Initially, there will not be any service instances. You will need to create a Database Service Instance by using the “Create Service Instance” wizard.
- Only after the Database Service Instance is created does memory get allocated from Oracle Cloud’s Database.
- Using the Public IP that is provided for the newly created Database Service Instance, you can create a new connection in SQL Developer and start accessing data.
- If you want to access the Database Service Instance by using SQL *Plus, you need to create an SSH tunnel by using the ssh utility on Linux.
- You can refer to Creating an SSH Tunnel Using the ssh Utility on Linux on the Oracle Help Center for more details.

Lesson Agenda

- Overview of course objectives
- Overview of course roadmap
- Introduction to Oracle Cloud
- Oracle Database 12*c* SQL documentation and additional resources

ORACLE

Oracle Database Documentation

- *Oracle Database Reference*
- *Oracle Database SQL Language Reference*
- *Oracle Database Concepts*
- *Oracle Database SQL Developer User's Guide*

ORACLE

Navigate to http://docs.oracle.com/en/database/database.html to access the Oracle Database 12*c* documentation library.

Additional Resources

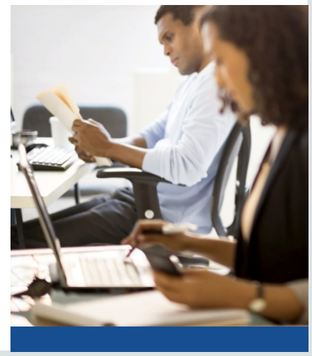
For additional information about Oracle Database 12*c*, refer to the following:

- *Oracle Database 12c: New Features eStudies*
- *Oracle Learning Library:*
  - http://www.oracle.com/goto/oll
- *Oracle Cloud:*
  - http://cloud.oracle.com
- Online SQL Developer Home Page, which is available at:
  - http://www.oracle.com/technology/products/database/sql_developer/index.html
- SQL Developer tutorial, which is available online at:
  - http://download.oracle.com/oll/tutorials/SQLDeveloper/index.htm

Summary

In this lesson, you should have learned about:

- The goals of the course
- The course roadmap used in this course
- The documentation and resources for reference



ORACLE

This lesson provided you an overview of the course objectives and the different units and lessons in the course. In the next lesson, you will learn about databases and database concepts.

Lesson 2: Relational Database Overview

Course Roadmap

- Lesson 1: Course Overview
- Unit 1: Relational Database and SQL Overview
  - Lesson 2: Relational Database Overview
  - Lesson 3: Database Storage Structures
  - Lesson 4: Introduction to SQL
- Unit 2: Retrieving and Sorting Data
- Unit 3: Joins, Subqueries, and Set Operators
- Unit 4: DML and DDL
- Unit 5: Managing Relational Database
- Unit 6: Advance Queries and Database Management System

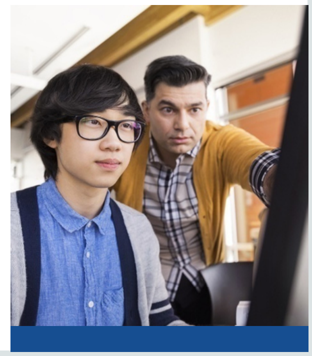
You are here

In Unit 1, you will learn about databases and database concepts. You will be introduced to relational databases, data storage concepts, and Structured Query Language (SQL).

Objectives

After completing this lesson, you should be able to:

- Define a database
- Describe the components of a database
- Explain the need of a database
- List the major transformations in database technology
- List the key concepts of a relational database
- Discuss the benefits of using a relational database



ORACLE

In this lesson, you gain an understanding of databases and database concepts. You also learn about the relational database management system (RDBMS). Additionally, you learn about the need and benefits of using a database.

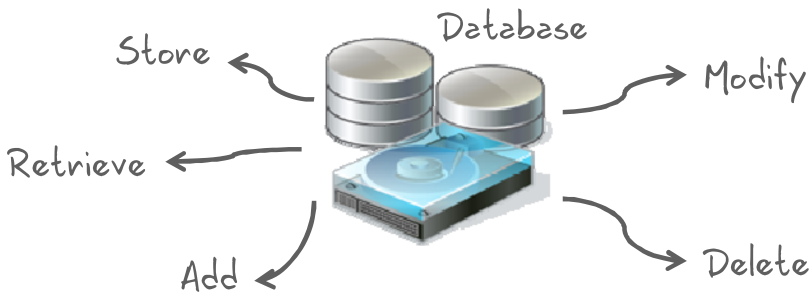
Lesson Agenda

- Introduction to Database
- Overview of Oracle Database 12*c*
- Overview of Relational Database management concepts and terminologies
- Overview of Database technologies

Database: Definition

A database:

- Is a centralized and structured set of data stored on a computer system
- Provides facilities for retrieving, adding, modifying, and deleting the data when required
- Provides facilities for transforming the retrieved data into useful information



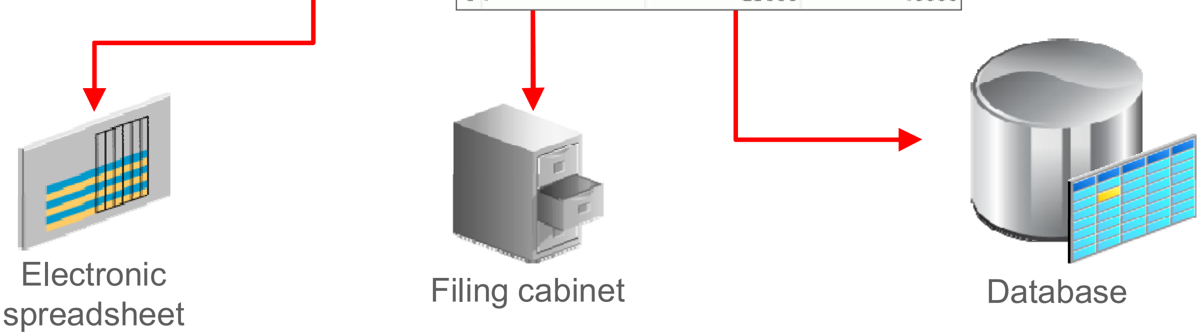
ORACLE

A database is a centralized and structured set of data stored on a computer system. It can be accessed in various ways. A database enables you to add, modify, and delete data. You can retrieve data and customize it into meaningful information.

Data Storage on Different Media

	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	10	Administration	200	1700
2	20	Marketing	201	1800
3	50	Shipping		
4	60	IT		
5	80	Sales		
6	90	Executive		
7	110	Accounting		
8	190	Contracting		

	GRADE_LEVEL	LOWEST_SAL	HIGHEST_SAL
1	A	1000	2999
2	B	3000	5999
3	C	6000	9999
4	D	10000	14999
5	E	15000	24999
6	F	25000	40000



Every organization has some information needs. A library keeps a list of members, books, due dates, and fines. A company needs to save information about its employees, departments, and salaries. These pieces of information are called *data*.

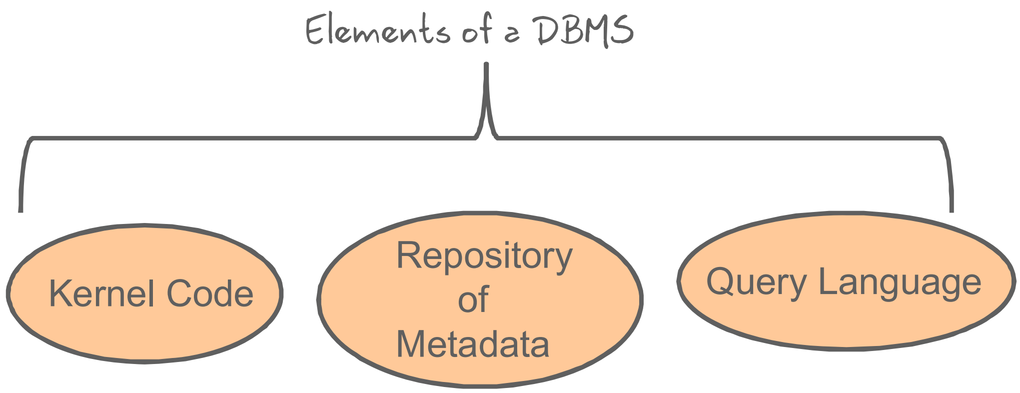
Organizations can store data in various media and in different formats, such as a hard copy document in a filing cabinet, or data stored in electronic spreadsheets or in databases.

A *database* is an organized collection of information.

To manage databases, you need a database management system (DBMS).

Database Management System (DBMS)

A DBMS is a software that controls the storage, organization, and retrieval of data.

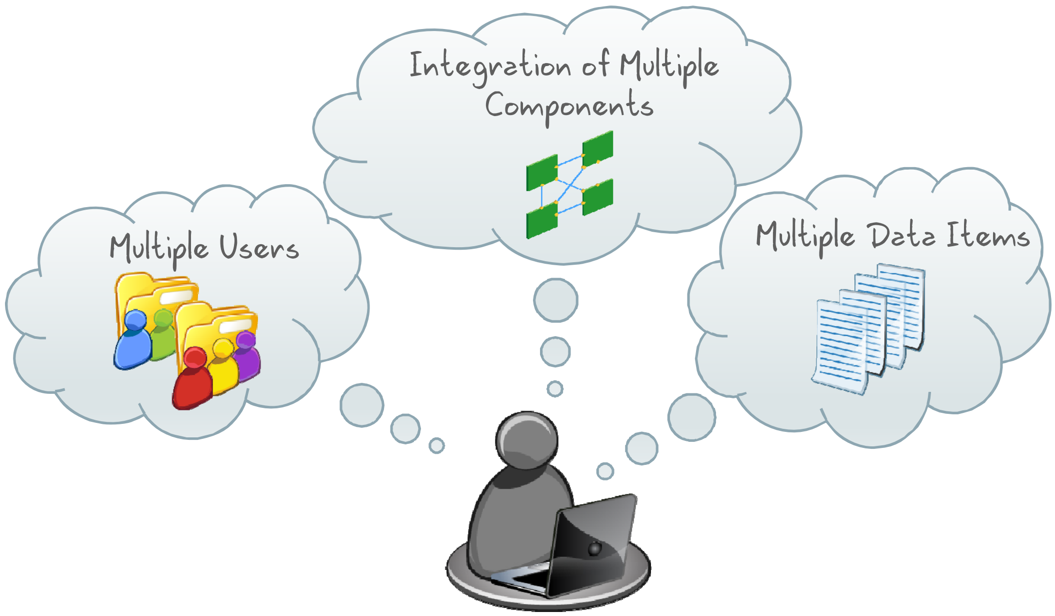


ORACLE

A DBMS has the following elements:

- The kernel code manages memory and storage for the DBMS.
- The repository of metadata is called a data dictionary.
- The query language enables applications to access the data.

Why Do I Need a Database Solution?



The basic component of a file in a file system is a data item. Examples of data items in the real world are last name, first name, street address, and employee ID.

A database is a more complex object. It is a collection of interrelated stored data that must meet the needs of many users. A database must also adhere to the business rules and processes of the organization.

Advantages of using a database rather than a simple file system are:

- Availability of data to a diverse group of users
- Integration of data for easier access and modification when performing complex transactions
- Data integrity and reduced data redundancy

Examples of Databases

ORACLE

Examples of areas where database applications are used:

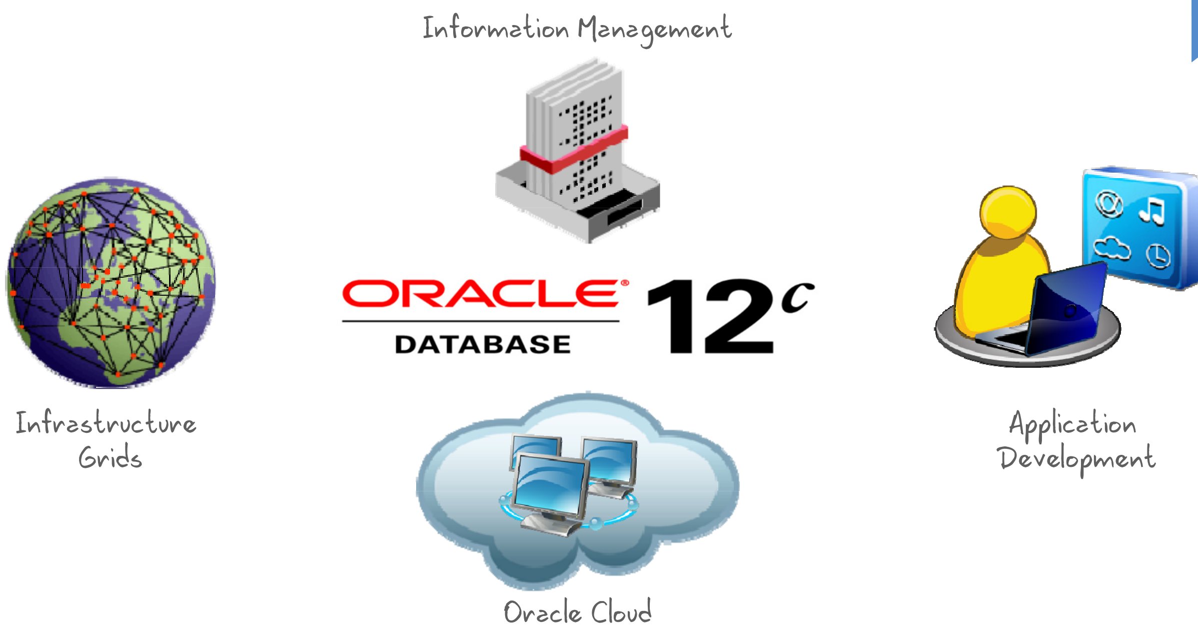
- Airlines and railways use online databases for reservations and for displaying information on the schedule.
- Banks use databases for storing information about customers, accounts, loans, and transactions.
- Schools and colleges use databases to maintain details about courses, students, and faculty.
- Telecommunication departments store information in their databases about the communication network, telephone numbers, call details, and monthly bills.
- Databases are used:
  - For keeping track of purchases on credit and debit cards, which helps generate monthly statements
  - For integrating heterogeneous information sources for business-related activities, such as online shopping, booking of holiday packages, and doctor consultations
  - In the healthcare industry to maintain and track patient healthcare details
  - In the area of digital publishing and digital libraries to manage and deliver textual and multimedia data
  - In finance and trading for storing information pertaining to sales, purchases of stocks and bonds, or online trading
  - At organizations for storing information about their employees, salaries, benefits, and taxes, and for generating paychecks

Lesson Agenda

- Introduction to Database
- Overview of Oracle Database 12*c*
- Overview of Relational Database management concepts and terminologies
- Overview of Database technologies

ORACLE

Oracle Database 12*c*: Focus Areas



Oracle Database 12*c* offers extensive features across the following focus areas:

- **Infrastructure Grids:** The Infrastructure Grid technology of Oracle enables pooling of low-cost servers and storage to form systems that deliver the highest quality of service in terms of manageability, high availability, and performance. Oracle Database 12*c* consolidates and extends the benefits of grid computing. Apart from taking full advantage of grid computing, Oracle Database 12*c* has unique change assurance features to manage changes in a controlled and cost-effective manner.
- **Information Management:** Oracle Database 12*c* extends the existing information management capabilities in content management, information integration, and information lifecycle management areas. Oracle provides content management of advanced data types such as Extensible Markup Language (XML), text, spatial, multimedia, medical imaging, and semantic technologies.
- **Application Development:** Oracle Database 12*c* has capabilities to use and manage all the major application development environments such as PL/SQL, Java/JDBC, .NET, Windows, PHP, SQL Developer, and Application Express.
- **Oracle Cloud:** The Oracle Cloud is an enterprise cloud for business. It provides an integrated collection of application and platform cloud services that are based on best-in-class products and open Java and SQL standards.

Oracle Database 12*c*

ORACLE
DATABASE

Manageability
High Availability
Performance
Security
Information Integration

Organizations need to support multiple terabytes of information for users who demand fast and secure access to business applications round the clock. The database systems must be reliable and must be able to recover quickly in the event of any kind of failure. Oracle Database 12*c* is designed along the following feature areas to help organizations manage infrastructure grids easily and deliver high-quality service:

- **Manageability:** By using some of the change assurance, management automation, and fault diagnostics features, the database administrators (DBAs) can increase their productivity, reduce costs, minimize errors, and maximize quality of service. Some of the useful features that promote better management are Database Replay facility, the SQL Performance Analyzer, the Automatic SQL Tuning facility, and Real-Time Database Operations Monitoring.

  Enterprise Manager Database Express 12*c* is a web-based tool for managing Oracle databases. Enterprise Manager Database Express greatly simplifies database performance diagnostics by consolidating the relevant database performance screens into a consolidated view called Database Performance Hub. DBAs get a single, consolidated view of the current real-time and historical view of the database performance across multiple dimensions such as database load, monitored SQL and PL/SQL, and Active Session History (ASH) on a single page for the selected time period.

- **High availability:** By using the high availability features, you can reduce the risk of down time and data loss. These features improve online operations and enable faster database upgrades.
- **Performance:** By using capabilities such as SecureFiles, compression for online transaction processing (OLTP), Real Application Clusters (RAC) optimizations, Result Caches, and so on, you can greatly improve the performance of your database. Oracle Database 12*c* enables organizations to manage large, scalable, transactional and data warehousing systems that deliver fast data access using low-cost modular storage.
- **Security:** Oracle Database 12*c* helps organizations protect their information with unique secure configurations, data encryption and masking, and sophisticated auditing capabilities. It delivers a secure and scalable platform for reliable and fast access to all types of information by using the industry-standard interfaces.
- **Information integration:** Oracle Database 12*c* has many features to better integrate data throughout the enterprise. It also supports advanced information lifecycle management capabilities. This helps you manage the changing data in your database.

Lesson Agenda

- Introduction to Database
- Overview of Oracle Database 12*c*
- Overview of Relational Database management concepts and terminologies
- Overview of Database technologies

ORACLE

Relational and Object Relational Database Management Systems

- Relational model and object relational model
- User-defined data types and objects
- Fully compatible with relational database
- Supports multimedia and large objects
- High-quality database server features

ORACLE

The Oracle server supports both the relational and the object relational database models.

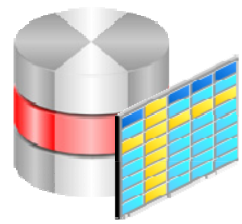
The Oracle server extends the data-modeling capabilities to support an object relational database model that provides object-oriented programming, complex data types, complex business objects, and full compatibility with the relational world.

It includes several features for improved performance and functionality of the OLTP applications, such as better sharing of runtime data structures, larger buffer caches, and deferrable constraints. Data warehouse applications benefit from enhancements such as parallel execution of insert, update, and delete operations; partitioning; and parallel-aware query optimization. The Oracle model supports client/server and web-based applications that are distributed and multitiered.

For more information about the relational and object relational model, refer to *Oracle Database Concepts for 12c Database*.

Relational Database Concept

- Dr. E. F. Codd proposed the relational model for database systems in 1970.
- It is the basis for the relational database management system (RDBMS).
- The relational model consists of the following:
  - Collection of objects or relations
  - Set of operators to act on the relations
  - Data integrity for accuracy and consistency



ORACLE

The principles of the relational model were first outlined by Dr. E. F. Codd in a June 1970 paper titled *A Relational Model of Data for Large Shared Data Banks*. In this paper, Dr. Codd proposed the relational model for database systems.

The common models used at that time were hierarchical and network, or even simple flat-file data structures. RDBMS soon became very popular, especially for its ease of use and flexibility in structure. In addition, a number of innovative vendors, such as Oracle, supplemented the RDBMS with a suite of powerful application development and user-interface products, thereby providing a total solution.

Components of the Relational Model

- Collections of objects or relations that store the data
- A set of operators that can act on the relations to produce other relations
- Data integrity for accuracy and consistency

Definition of a Relational Database

A relational database is a collection of relations or two-dimensional tables controlled by the Oracle server.

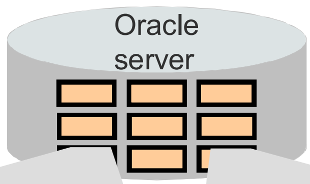


Table name: `EMPLOYEES`

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL
100	Steven	King	SKING
101	Neena	Kochhar	NKOCHHAR
102	Lex	De Haan	LDEHAAN

. . .

Table name: `DEPARTMENTS`

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID
10	Administration	200
20	Marketing	201
50	Shipping	124

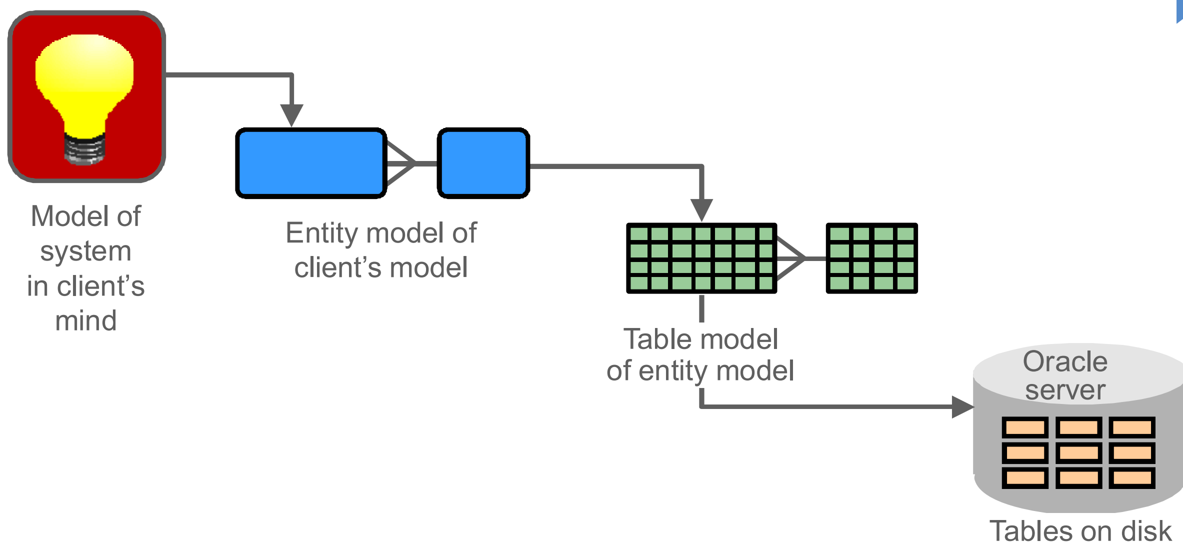
. . .

ORACLE

A relational database uses relations or two-dimensional tables to store information.

For example, you might want to store information about all the employees in your company. In a relational database, you create several tables to store different pieces of information about your employees, such as an employee table, a department table, and a salary table.

Data Models



Models are the cornerstone of design. Engineers build a model of a car to work out any details before putting it into production. In the same manner, system designers develop models to explore ideas and improve the understanding of database design.

Purpose of Models

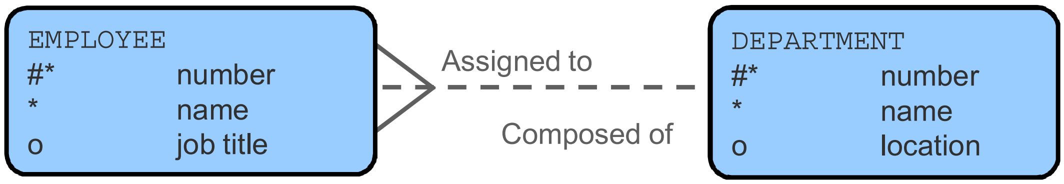
Models help to communicate the concepts that are in people's minds. They can be used to do the following:

- Communicate
- Categorize
- Describe
- Specify
- Investigate
- Evolve
- Analyze
- Imitate

The objective is to produce a model that fits a multitude of these uses, can be understood by an end user, and contains sufficient detail for a developer to build a database system.

Entity Relationship Model

- Create an entity relationship diagram from business specifications or narratives:



- Scenario:
  – “. . . Assign one or more employees to a department . . .”
  – “. . . Some departments do not yet have assigned employees . . .”

ORACLE

In an effective system, data is divided into discrete categories or entities. An entity relationship (ER) model is an illustration of the various entities in a business and the relationships among them. An ER model is derived from business specifications or narratives and built during the analysis phase of the system development life cycle. ER models separate the information required by a business from the activities performed within the business. Although businesses can change their activities, the type of information tends to remain constant. Therefore, the data structures also tend to be constant.

Benefits of ER Modeling

- Documents information for the organization in a clear, precise format
- Provides a clear picture of the scope of the information requirement
- Provides an easily understood pictorial map for database design
- Offers an effective framework for integrating multiple applications

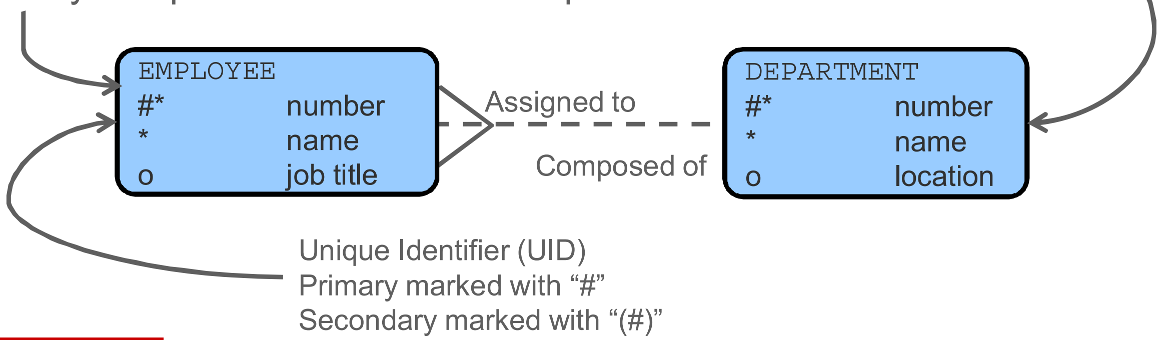
Key Components

- **Entity:** An aspect of significance about which information must be known. Examples are departments, employees, and orders.
- **Attribute:** Something that describes or qualifies an entity. For example, for the employee entity, the attributes would be the employee number, name, job title, hire date, department number, and so on. Each of the attributes is either required or optional. This state is called *optionality*.
- **Relationship:** A named association between entities showing optionality and degree. Examples are employees and departments, and orders and items.

Entity Relationship Modeling Conventions

Entity:
- Singular, unique name
- Uppercase
- Soft box
- Synonym in parentheses

Attribute:
- Singular name
- Lowercase
- Mandatory marked with "*"
- Optional marked with "o"



Entities

To represent an entity in a model, use the following conventions:

- Singular, unique entity name
- Entity name in uppercase
- Soft box
- Optional synonym names in uppercase within parentheses: ( )

Attributes

To represent an attribute in a model, use the following conventions:

- Singular name in lowercase
- Asterisk (*) tag for mandatory attributes (that is, values that *must* be known)
- Letter "o" tag for optional attributes (that is, values that *may* be known)

Relationships

Each direction of the relationship contains:

- **A label:** For example, *taught by* or *assigned to*
- **An optionality:** Either *must be* or *maybe*
- **A degree:** Either *one and only one* or *one or more*

Symbol	Description
Dashed line	Optional element indicating “maybe”
Solid line	Mandatory element indicating “must be”
Crow’s foot	Degree element indicating “one or more”
Single line	Degree element indicating “one and only one”

Note: The term *cardinality* is a synonym for the term *degree*.

Each source entity {may be | must be} in relation {one and only one | one or more} with the destination entity.

Note: The convention is to read clockwise.

Unique Identifiers

A unique identifier (UID) is any combination of attributes or relationships, or both, that serves to distinguish occurrences of an entity. Each entity occurrence must be uniquely identifiable.

- Tag each attribute that is part of the UID with a hash sign “#”.
- Tag secondary UIDs with a hash sign in parentheses (#).

Relating Multiple Tables

- Each row of data in a table can be uniquely identified by a primary key.
- You can logically relate data from multiple tables using foreign keys.

Table name: `DEPARTMENTS`

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
50	Shipping	124	1500
60	IT	103	1400
80	Sales	149	2500
90	Executive	100	1700
110	Accounting	205	1700
190	Contracting	(null)	1700

Primary key

Table name: `EMPLOYEES`

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	DEPARTMENT_ID
100	Steven	King	90
101	Neena	Kochhar	90
102	Lex	De Haan	90
103	Alexander	Hunold	60
104	Bruce	Ernst	60
107	Diana	Lorentz	60
124	Kevin	Mourgos	50
141	Trenna	Rajs	50
142	Curtis	Davies	50

...

Primary key

Foreign key

ORACLE

Each table contains data that describes exactly one entity. For example, the `EMPLOYEES` table contains information about employees. Categories of data are listed across the top of each table, and individual cases are listed below. By using a table format, you can readily visualize, understand, and use information.

Because data about different entities is stored in different tables, you may need to combine two or more tables to answer a particular question. For example, you may want to know the location of the department where an employee works. In this scenario, you need information from the `EMPLOYEES` table (which contains data about employees) and the `DEPARTMENTS` table (which contains information about departments). With an RDBMS, you can relate the data in one table to the data in another table by using foreign keys. A foreign key is a column (or a set of columns) that refers to a primary key in the same table or another table.

You can use the ability to relate data in one table to data in another table to organize information in separate, manageable units. Employee data can be kept logically distinct from the department data by storing it in a separate table.

Guidelines for Primary Keys and Foreign Keys

- You cannot use duplicate values in a primary key.
- Primary keys generally cannot be changed.
- Foreign keys are based on data values and are purely logical (not physical) pointers.
- A foreign key value must match an existing primary key value or unique key value; otherwise, it must be null.
- A foreign key must reference either a primary key or a unique key column.

Relational Database Terminology

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	SALARY	COMMISSION_PCT	DEPARTMENT_ID
100	Steven	King	24000	(null	90
101	Neena	Kochhar	17000	(null	90
102	Lex	De Haan	17000	(null	90
103	Alexander	Hunold	9000	(null	60
104	Bruce	Ernst	6000	(null	60
107	Diana	Lorentz	4200	(null	60
124	Kevin	Mourgos	5800	(null	50
141	Trenna	Rajs	3500	(null	50
142	Curtis	Davies	3100	(null	50
143	Randall	Matos	2600	(null	50
144	Peter	Vargas	2500	(null	50
149	Eleni	Zlotkey	10500	0.2	80
174	Ellen	Abel	11000	0.3	80
176	Jonathon	Taylor	8600	0.2	80
178	Kimberely	Grant	7000	0.15	(null)
200	Jennifer	Whalen	4400	(null	10
201	Michael	Hartstein	13000	(null	20
202	Pat	Fay	6000	(null	20
205	Shelley	Higgins	12000	(null	110
206	William	Gietz	8300	(null	110

ORACLE

A relational database can contain one or many tables. A *table* is the basic storage structure of an RDBMS. A table holds all the data necessary about something in the real world, such as employees, invoices, or customers.

The slide shows the contents of the `EMPLOYEES` *table* or *relation*. The numbers indicate the following:

1. A single *row* (or *tuple*) representing all the data required for a particular employee. Each row in a table should be identified by a primary key, which permits no duplicate rows. The order of rows is insignificant; specify the row order when the data is retrieved.
2. A *column* or attribute containing the employee number. The employee number identifies a *unique* employee in the `EMPLOYEES` table. In this example, the employee number column is designated as the *primary key*. A primary key must contain a value and the value must be unique.
3. A column that is not a key value. A column represents one kind of data in a table; in this example, the data is the salaries of all the employees. Column order is insignificant when storing data; specify the column order when the data is retrieved.
4. A column containing the department number, which is also a *foreign key*. A foreign key is a column that defines how tables relate to each other. A foreign key refers to a primary key or a unique key in the same table or in another table. In the example, `DEPARTMENT_ID` uniquely identifies a department in the `DEPARTMENTS` table.
5. A *field* can be found at the intersection of a row and a column. There can be only one value in it.
6. A field may have no value in it. This is called a null value. In the `EMPLOYEES` table, only those employees who have the role of sales representative have a value in the `COMMISSION_PCT` (commission) field.

Advantages of a Relational Database

- Avoids duplication of data
- Ensures consistency of data that is stored as records
- Easier to modify data and data format
- Easier to insert and delete data
- Easier to maintain security of data

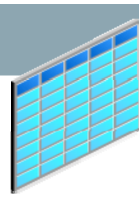
When you store data in tables, you can easily add, modify, and delete data, as well as maintain consistency of the stored information.

Lesson Agenda

- Introduction to Database
- Overview of Oracle Database 12*c*
- Overview of Relational Database management concepts and terminologies
- Overview of Database technologies
  - Difference between OLTP and OLAP
  - Difference between SQL Database and NoSQL Database
  - Overview of Multitenant architecture of Oracle Database 12*c*
  - Introduction to Oracle Cloud and Database Cloud Service

OLTP Versus OLAP

OLTP	OLAP
Works with operational data	Works with historical data
Is used for updating data	Is used for reporting data
Schema is normalized	Schema can be of any type (star, snowflake, constellation)
Simple queries are used	Complex queries are used
Retrieval of data is fast	Retrieval of data is slow



ORACLE

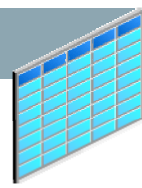
There are two types of information systems that help in managing and processing transaction-oriented and analytical-oriented applications. They are as follows:

Online Transaction Processing (OLTP) refers to a number of simple online transactions such as `INSERT`, `UPDATE`, or `DELETE`. The main function of OLTP is to retrieve, modify, or delete data at a high speed while maintaining data integrity in a centralized environment. A special characteristic of an OLTP database is the normalization of data. This helps in providing faster access and efficient performance of the database. This system is suitable to address online applications such as a banking transaction system.

Online Analytical Processing (OLAP) refers to relatively a small number of online transactions using complex queries. The main function of OLAP is to retrieve aggregated data from a set of historical or archived data stored in the database. Since the data is seldom changed, it can be used for analytical purposes. This system is suitable for applications such as business intelligence and data mining.

SQL Database Versus NoSQL Database

SQL Database	NoSQL Database
Is a relational database	Is a non-relational or distributed database
Stores data as records	Stores data as documents (JSON, key-value pair)
Schemas are predefined	Schemas are dynamic
Scaling is vertical	Scaling is horizontal
Uses SQL to query database	Uses APIs to query database

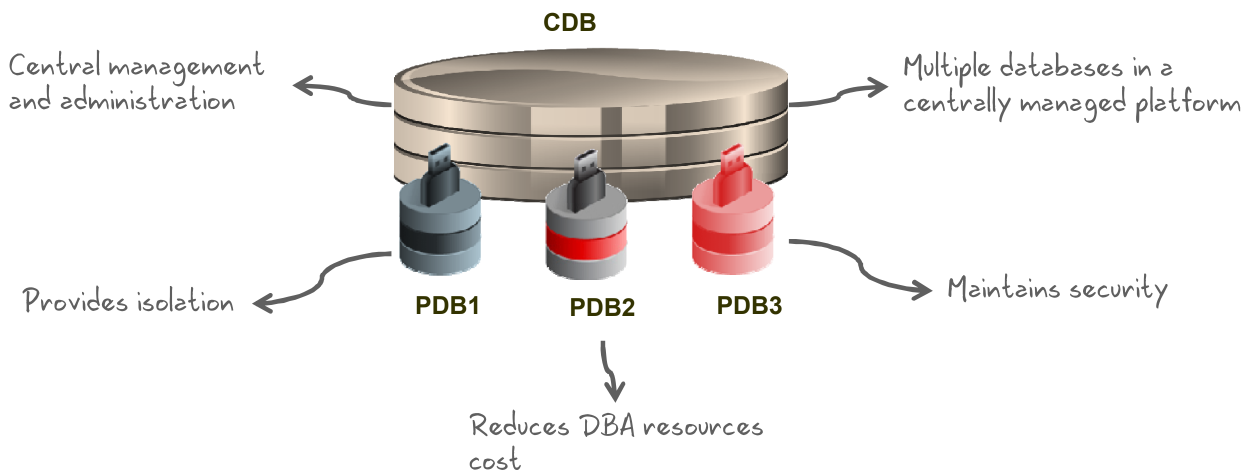


The different types of databases differ mainly in their architecture and how the data is stored. They also differ based on the applications they are used for.

SQL Database is a relational database whose data can be queried by using SQL. You can store data in the tables only if the tables and the field types are defined. Hence the design must be finalized before applying any business logic to manipulate data. All the tables in the database are normalized, thereby reducing redundancy. Complex queries can be written to retrieve customized reports. In order to scale up the database, you will need to increase the capacity of the server, which in turn increases the cost. SQL Database is ideal in cases where data requirements can be identified and data integrity is essential.

NoSQL Database is a scalable, distributed database. APIs with programming languages such as Java, Python, etc. are used to retrieve document-based data. Data can be modeled as relational database–style tables, JSON documents, or key-value pairs. Oracle NoSQL Database is a sharded (shared-nothing) system, which distributes the data uniformly across the multiple shards in the cluster. Within each shard, storage nodes are replicated to ensure high availability, rapid failover in the event of a node failure, and optimal load balancing of queries. NoSQL Database provides Java, C, Python, and node.js drivers and a REST API to simplify application development. NoSQL Database is integrated with a wide variety of related Oracle and open source applications in order to simplify and streamline the development and deployment of modern big data applications.

Multitenant Architecture



The **multitenant architecture** enables an Oracle database to function as a multitenant container database (CDB). A CDB includes zero, one, or many pluggable databases (PDBs).

A PDB is a portable collection of schemas, schema objects, and non-schema objects. In other words, a database that consolidates other databases is called a container database or CDB, and a database consolidated within a CDB is called a pluggable database or PDB.

All Oracle databases before Oracle Database 12*c* were non-CDBs. Oracle Database 12*c* supports both the new multitenant architecture and the old non-CDB architecture.

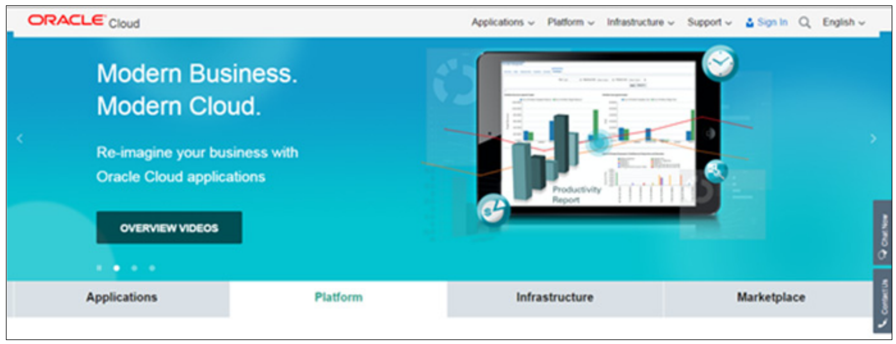
In this course, you will be connecting to a schema in a PDB to execute the SQL statements.

Introduction to Oracle Cloud

The Oracle Cloud is an enterprise cloud for business. The Oracle Public Cloud consists of many different services that share some common characteristics:

- On-demand self-service
- Resource pooling
- Rapid elasticity
- Measured service
- Broad network access

www.cloud.oracle.com



The Oracle Cloud is an enterprise cloud for business. It provides an integrated collection of application and platform cloud services that are based upon best-in-class products and open Java and SQL standards.

As a result, the applications and databases deployed in the Oracle Cloud are portable and can be easily moved to or from a private cloud or an on-premise (local machine) environment.

- All Cloud Services can be provisioned through a self-service interface. Users can get their Cloud Services delivered on an integrated development and deployment platform, with tools to rapidly extend and create new services. Oracle Cloud services are built on the Oracle Exalogic Elastic Cloud and Oracle Exadata Database Machine, together offering a platform that delivers extreme performance, redundancy, and scalability. The top two benefits of cloud computing are speed and cost.

The five essential characteristics are:

- **On-demand self-service:** Provisioning, monitoring, management control
- **Resource pooling:** Implies sharing and a level of abstraction between consumers and services
- **Rapid elasticity:** The ability to quickly scale up or down as needed
- **Measured service:** Metering utilization for either internal chargeback (private cloud) or external billing (public cloud)
- **Broad network access:** Typically means access through a browser on any networked device

Oracle Cloud Services

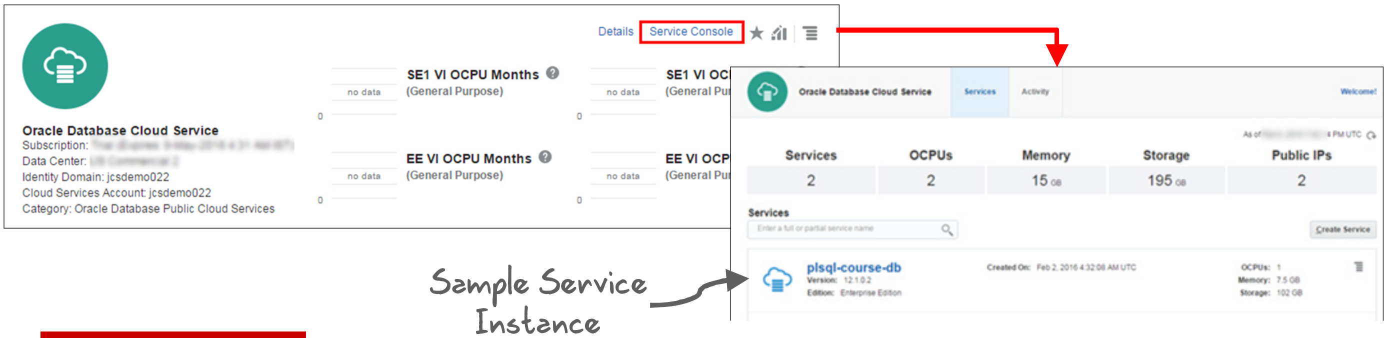
The Oracle Cloud provides the following three types of services:

- Software as a Service (SaaS)
- Platform as a Service (PaaS)
- Infrastructure as a Service (IaaS)

- **Software as a Service (SaaS)** generally refers to applications that are delivered to end users over the Internet. Oracle CRM On Demand is an example of a SaaS offering that provides both multitenant as well as single-tenant options, depending on the customer's preferences.
- **Platform as a Service (PaaS)** generally refers to an application development and deployment platform delivered as a service to developers, enabling them to quickly build and deploy a SaaS application to end users. The platform typically includes databases, middleware, and development tools, all delivered as a service via the Internet.
- **Infrastructure as a Service (IaaS)** refers to computing hardware (servers, storage, and network) delivered as a service. This typically includes the associated software as well as operating systems, virtualization, clustering, and so on. Examples of IaaS in the public cloud include Amazon's Elastic Compute Cloud (EC2) and Simple Storage Service (S3).
- The database cloud is built within an enterprise's private cloud environment, as a PaaS model. The database cloud provides on-demand access to database services in a self-service, elastically scalable, and metered manner. The database cloud offers compelling advantages in cost, quality of service, and agility. A database can also be deployed within a virtual machine in an IaaS platform.
- Database clouds can be rapidly deployed on Oracle Exadata, a pre-integrated and optimized hardware platform that supports both OLTP and Data Warehouse workloads.

Database on Oracle Cloud

- Oracle Cloud Database as a Service (DBaaS)
  - Implemented as a PaaS
  - Service identified by an identity domain
  - Sign in and create a new Database Service Instance to start using the Cloud Database memory



ORACLE®

- Oracle Cloud Database as a Service (DBaaS) is a service offered as PaaS.
- When an Oracle Cloud DBaaS is purchased or requested for a trial version, you will receive a welcome email from the Oracle Cloud team with following details:
  - Service Identity Domain
  - User ID
  - Temporary password, which has to be changed upon your first sign-in
- Once you sign in to the service, you will see the “My Services” page with all the options to manage the service and its users.
- Initially, there are no service instances and you need to create a Database Service Instance using the “Create Service Instance” wizard.
- Only upon the Database Service Instance creation does the memory get allocated from the Oracle Cloud’s Database.
- Using the Public IP provided for the newly created Database Service Instance, you can create a new connection in SQL Developer and start accessing the data.
- If you want to access the Database Service Instance using SQL *Plus, you need to create an SSH tunnel using the ssh utility on Linux.
- You can refer to http://www.oracle.com/webfolder/technetwork/tutorials/obe/cloud/dbaas/obe_dbaas_QS/oracle_database_cloud_service_dbaas_quick_start.html in Oracle Help Center for more details.

Quiz

Which of the following are true for a Primary Key?

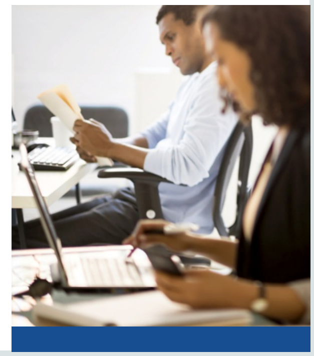
a. Can contain duplicate values
b. Has unique values in a table
c. Cannot be `NULL`
d. Can be changed to a different value in a table

Answer: b, c

Summary

In this lesson, you should have learned about:

- The features of Oracle Database 12*c*
- The theoretical and physical aspects of a relational database
- Oracle server's implementation of RDBMS and object relational database management system (ORDBMS)
- The major transformations in database technology
- The salient features of Oracle Cloud



ORACLE

Relational database management systems are composed of objects or relations. They are managed by operations and governed by data integrity constraints.

Oracle Corporation produces products and services to meet your RDBMS needs. The main products are the following:

- Oracle Database with which you store and manage information by using SQL
- Oracle Fusion Middleware with which you develop, deploy, and manage modular business services that can be integrated and reused
- Oracle Enterprise Manager Grid Control, which you use to manage and automate administrative tasks across sets of systems in a grid environment

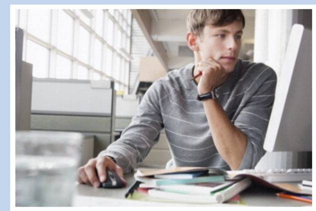
SQL

The Oracle server supports ANSI-standard SQL and contains extensions. SQL is the language that is used to communicate with the server to access, manipulate, and control data.

Practice 2: Overview

This practice covers the following topics:

- Identifying entities, attributes, and their corresponding tables, rows, and columns
- Identifying unique identifiers and their corresponding primary keys



ORACLE

In this practice, you perform the following:

- Identify entities, attributes, and their corresponding tables, rows, and columns.
- Identify unique identifiers and their corresponding primary keys.

Note the following location for the lab files:

```
/home/oracle/labs/sql1/labs
```

If you are asked to save any lab files, save them in this location.

In any practice, there may be exercises that are prefaced with the phrases “If you have time” or “If you want an extra challenge.” Work on these exercises only if you have completed all other exercises within the allocated time and would like a further challenge to your skills.

Perform the practices slowly and precisely. You can experiment with saving and running command files. If you have any questions at any time, ask your instructor.

Note: All written practices use Oracle SQL Developer as the development environment. Although it is recommended that you use Oracle SQL Developer, you can also use SQL*Plus that is available in this course.


Lesson 3: Database Storage Structures
ORACLE
Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Course Roadmap

- Lesson 1: Course Overview
- Unit 1: Relational Database and SQL Overview
  - Lesson 2: Relational Database Overview
  - Lesson 3: Database Storage Structures
  - Lesson 4: Introduction to SQL
- Unit 2: Retrieving and Sorting Data
- Unit 3: Joins, Subqueries, and Set Operators
- Unit 4: DML and DDL
- Unit 5: Managing Relational Database
- Unit 6: Advance Queries and Database Management System

You are here

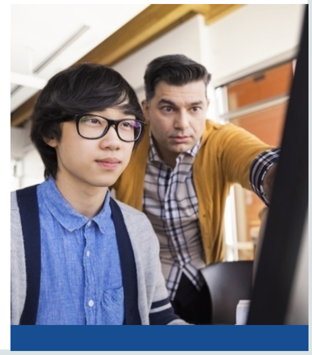
ORACLE

In Unit 1, you will learn about databases and database concepts. You will be introduced to relational databases, data storage concepts, and SQL.

Objectives

After completing this lesson, you should be able to:

- Understand database data storage
- Define logical structures
- Define physical storage structures
- Describe the structure of relational tables

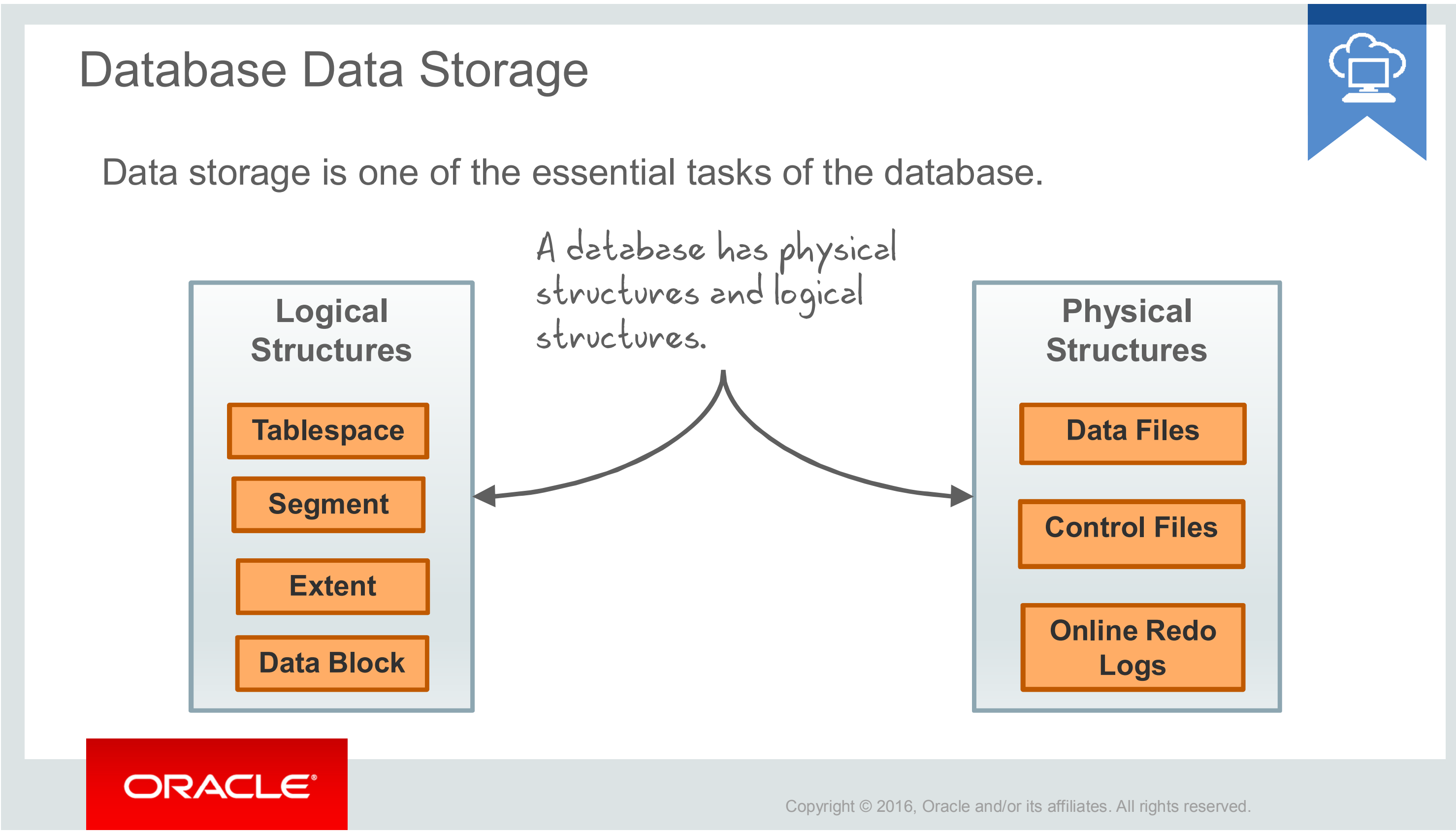


ORACLE

In this lesson, you will learn about database data storage. You will gain an understanding of logical structures and physical storage structures. You will also learn about relational table structure.

Lesson Agenda

- Overview of database data storage
- Introduction to logical structures
- Introduction to physical storage structures
- Structure of relational tables



A database can be considered from both a physical and a logical perspective. Physical data is data viewable at the operating system level. Logical data, such as a table, is meaningful only for the database. A SQL statement can list the tables in an Oracle database, but an operating system utility cannot.

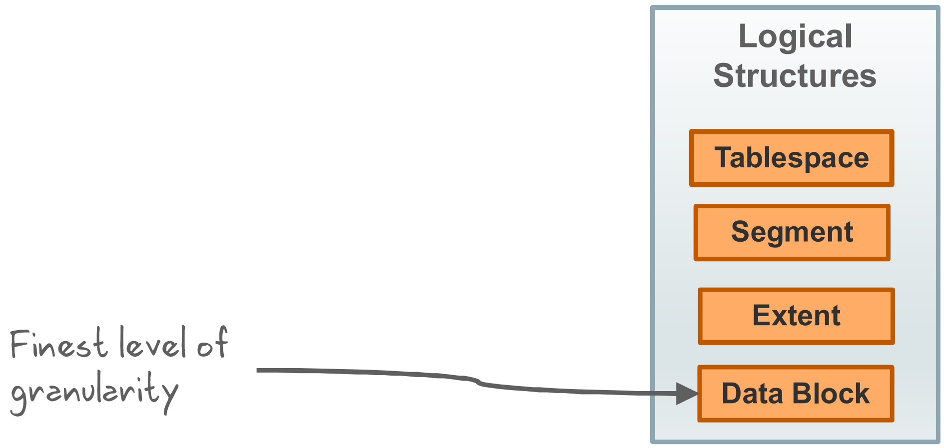
The physical storage of data can be managed without affecting access to logical storage structures because the physical and logical structures are separate.

Lesson Agenda

- Overview of database data storage
- Introduction to logical structures
- Introduction to physical storage structures
- Structure of relational tables

Introduction to Logical Structures

- Oracle Database allocates logical space for all data in the database.
- There are four allocation units of database space allocation:
  - Data Blocks
  - Extents
  - Segments
  - Tablespaces



ORACLE

Data blocks are the smallest units of storage that Oracle Database can use or allocate. At the finest level of granularity, Oracle Database stores data in data blocks. One logical data block corresponds to a specific number of bytes of physical disk space.

An extent is a set of logically contiguous data blocks allocated for storing a specific type of information.

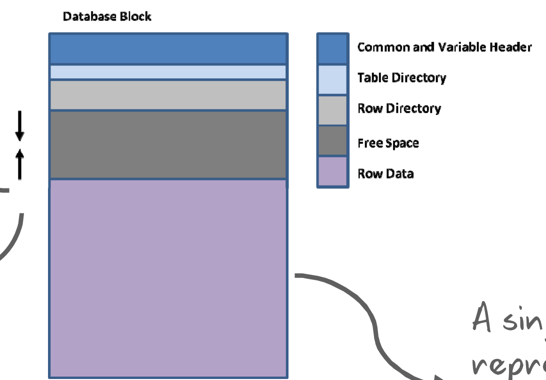
A segment is a set of extents allocated for a specific database object, such as a table. For example, the data for the Employees table is stored in its own data segment. Every database object that consumes storage consists of a single segment.

Each segment belongs to one and only one tablespace. Thus, all extents for a segment are stored in the same tablespace.

Data Blocks

A data block is the smallest logical storage unit of a database.

The size of a data block is generally a multiple of the operating system block size.



A single data block represents a specific number of bytes on the physical hard disk.

The Oracle Database manages the logical storage space in the data files of a database in units called data blocks, also called Oracle blocks or pages. A data block is the minimum unit of database I/O. At the physical level, database data is stored in disk files made up of operating system blocks. An operating system block is the minimum unit of data that the operating system can read or write. In contrast, an Oracle block is a logical storage structure whose size and structure are not known to the operating system.

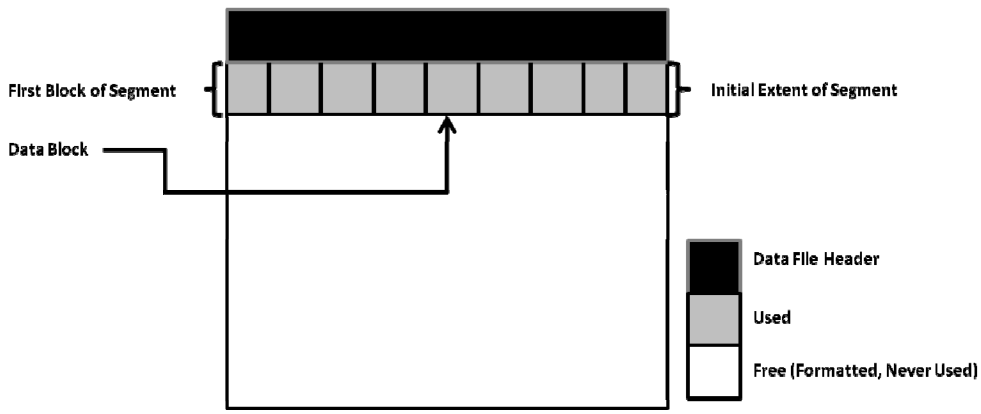
A data block consists of the following format:

- Header that holds generic information like block address and type of segment
- Table Directory that contains information about the table having rows in that block
- Row Directory that contains information about the actual row contained in that block
- Free Space that is the available free space in the data block
- Row Data that contains table or index data

The first three components of a data block (Header, Table Directory, and Row Directory) are collectively known as Overhead.

Extents

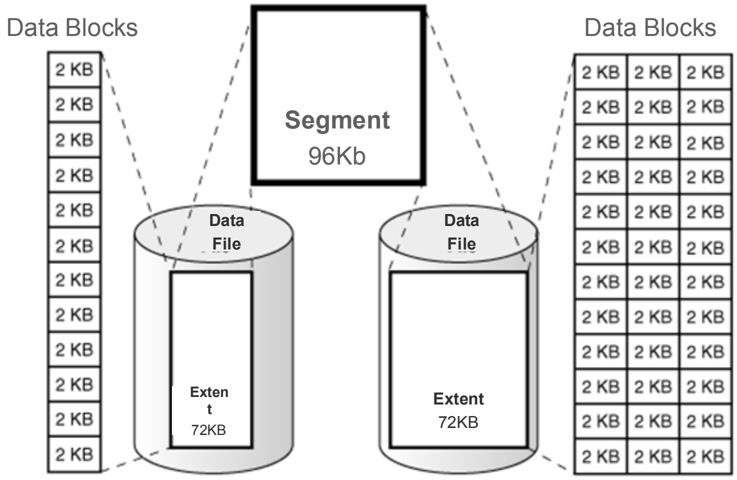
An extent is a logical unit of database storage space allocation made up of contiguous data blocks.



Data blocks in an extent are logically contiguous, but can be physically spread out on the disk. By default, the database allocates an initial extent for a data segment when the segment is created. The first data block of every segment contains a directory of the extents in the segment.

Segments

A segment is a set of extents that have been allocated for a specific type of data structure and that are stored in the same tablespace.



ORACLE

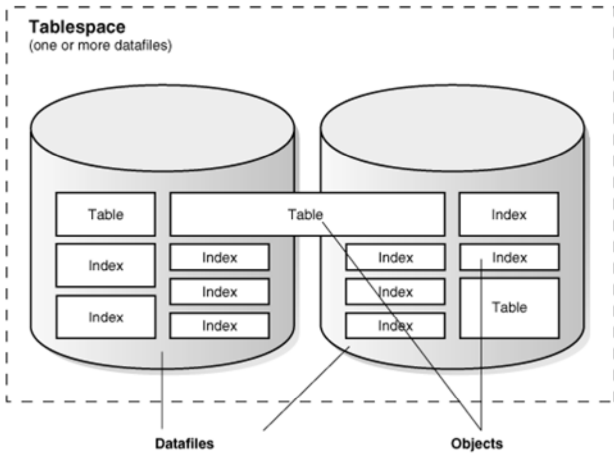
A segment is a set of extents that have been allocated for a specific type of data structure and that are stored in the same tablespace. For example, each table's data is stored in its own data segment, while each query's data is stored in a temporary segment. Oracle allocates space for segments in extents.

Oracle allocates another extent when the existing extents of a segment become full. The extents of a segment may or may not be contiguous on disk, because extents are allocated on an as-needed basis.

Segments reside in a physical structure called a Data File, which is covered in further topics.

Tablespaces

Oracle Database stores data logically in tablespaces and physically in data files associated with the corresponding tablespaces.



Tablespaces are the primary logical storage structures of any Oracle database. The usable data of an Oracle database is logically stored in the tablespaces and physically stored in the data files associated with the corresponding tablespaces.

An Oracle database consists of one or more logical storage units called tablespaces. The database's data is collectively stored in the database's tablespaces. Each tablespace in an Oracle database consists of one or more files called data files, which are physical structures that conform to the operating system in which an Oracle database is running.

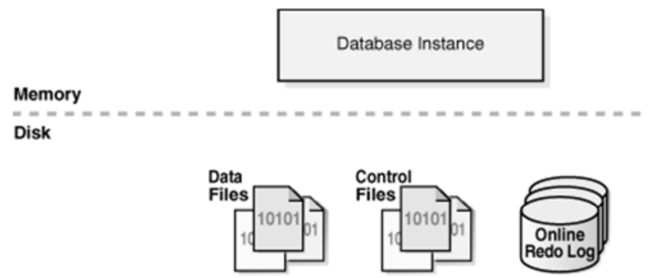
An index is associated with a tablespace and helps in retrieving data more quickly. Just as the index in a manual helps you locate information faster than if there were no index, a database index provides a faster access path to tablespace data.

Lesson Agenda

- Overview of database data storage
- Introduction to logical structures
- Introduction to physical storage structures
- Structure of relational tables

Introduction to Physical Storage Structures

- An Oracle Database is a set of files that stores Oracle data in persistent disk storage.
- The following database files are generated:
  – Data files and temp files
  – Control files
  – Online redo log files

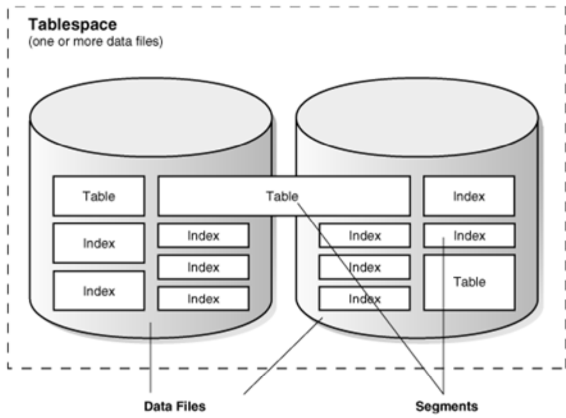


ORACLE

- A data file is a physical file on the disk that is created by Oracle Database and contains data structures, such as tables and indexes. A temp file is a data file that belongs to a temporary tablespace. The data is written to these files in an Oracle proprietary format that cannot be read by other programs.
- A control file is a root file that tracks the physical components of the database.
- The online redo log is a set of files containing records of changes made to data.
- A database instance is a set of memory structures that manages database files.

Data Files

- Oracle Database stores database data in data files.
- Every database must have at least one data file.

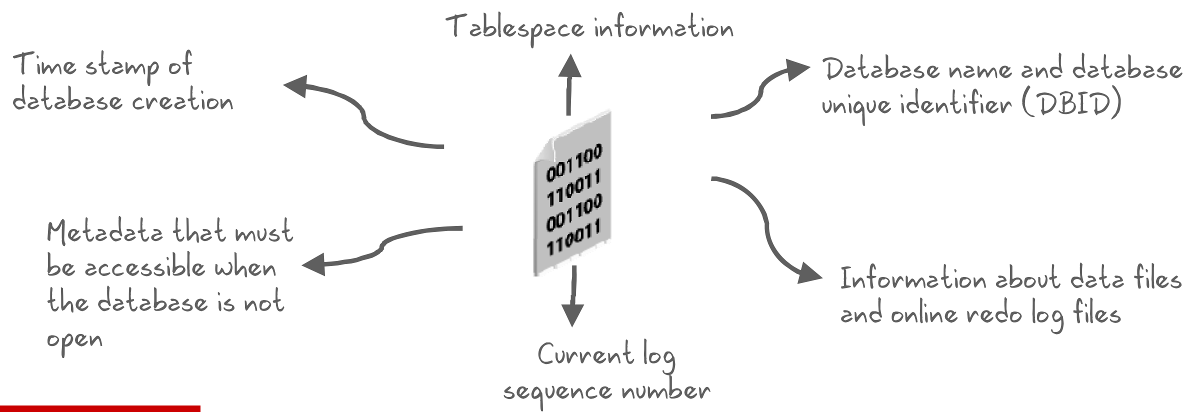


Oracle Database stores database data in data files. Every database must have at least one data file. Oracle Database allocates space for user data in tablespaces, which, like segments, are logical storage structures. Each segment belongs to only one tablespace. Oracle Database physically stores tablespace data in data files. Tablespaces and data files are closely related, but they have important differences:

- Each tablespace consists of one or more data files, which conform to the operating system in which the Oracle database is running.
- The data for a database is collectively stored in the data files located in each tablespace of the database.
- A segment can span one or more data files, but it cannot span multiple tablespaces.
- A database must have the SYSTEM and SYSAUX tablespaces. Oracle Database automatically allocates the first data files of any database for the SYSTEM tablespace during database creation.
- The SYSTEM tablespace contains the data dictionary, which is a set of tables that contains database metadata. Generally, a database also has an undo tablespace and a temporary tablespace (usually named TEMP).

Control Files

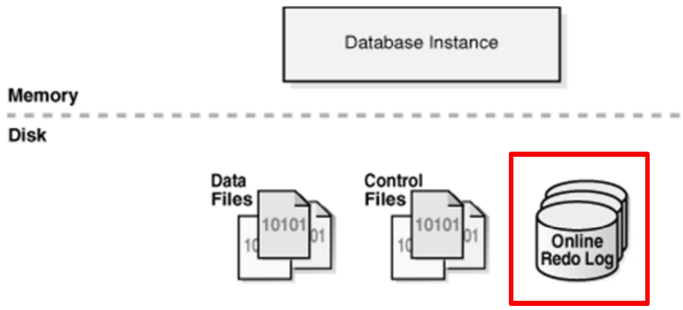
- The database control file is a small binary file associated with only one database.
- A control file contains the following information:



Each database has one unique control file, although it may maintain identical copies of it. The control file is the root file that Oracle Database uses to find database files and to manage the state of the database generally. The control file of an Oracle database is created at the same time as the database.

Online Redo Log Files

Oracle Database uses the online redo log only for recovery.



Every instance of an Oracle database has an associated redo log to protect the database in case of an instance failure. It consists of two or more pre-allocated files that store all changes made to the database as they occur.

The redo log for each database instance is also referred to as a redo thread.

Redo log files are filled with redo records. A redo record, also called a redo entry, is made up of a group of change vectors, each of which is a description of a change made to a single block in the database. For example, if you change a salary value in an employee table, you generate a redo record. This contains change vectors that describe changes to the data segment block for the table, the undo segment data block, and the transaction table of the undo segments.

Lesson Agenda

- Overview of database data storage
- Introduction to logical structures
- Introduction to physical storage structures
- Structure of relational tables

ORACLE

Relational Tables

A table is a simple structure where data is organized and stored.

Table: **EMPLOYEES**

Columns

Rows

EMPLOYEE_ID	LAST_NAME	FIRST_NAME	DEPARTMENT_ID	PAYROLL_ID	NICKNAME
100	SMITH	DANA	10	21215	Dana
310	ADAMS	TYLER	15	59877	Ty
210	CHEN	LAWRENCE	10	1101	Larry
405	GOMEZ	CARLOS	10	52	Chaz
378	LOUNGANI	NEIL	22	90386	Neil

Primary Key Column (PK)

Foreign Key Column (FK)

Unique Key Column (UK)

ORACLE

A relational database management system (RDBMS) stores data in tables. Each table is given a name by the user who creates the table. The user generally chooses a name that correlates to the data that will be stored in the table, for example, `STUDENTS, EMPLOYEES, and LOCATIONS.` When a table is created, the user also creates and names columns related to the specific characteristics that are stored for each record.

Tables have columns and rows. In the slide example, the `EMPLOYEES` table stores employee information. Each row describes an occurrence of an employee. Each column is used to store a specific type of value, such as employee number, last name, and first name.

The `EMPLOYEE_ID` column is a primary key. Every employee has a unique identification number. The value in the primary key column distinguishes each individual row. The `PAYROLL_ID` column is a unique key. This means that the system does not allow two rows with the same `payroll_id`.

The foreign key column refers to a row in another table. In this example, `department_id` refers to a row in the `DEPARTMENTS` table. You know that Dana Smith works in department 10. If you wanted to know more about Dana Smith's department, you would look for the `department_id = 10` row in the `DEPARTMENTS` table.

Quiz

More than one tablespace can be present in a database.

a. True

b. False

Answer: a

Quiz

A segment is a logical structure that can span multiple tablespaces.

a. True

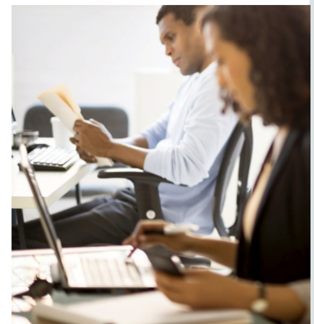
b. False

Answer: b

Summary

In this lesson, you should have learned how to:

- Discuss database data storage
- Define logical structures namely data blocks, extents, segments, and tablespaces
- Define physical storage structures namely data files, control files, and online redo log files
- Describe the structure of relational tables

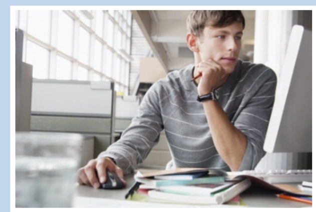


ORACLE

Practice 3: Overview

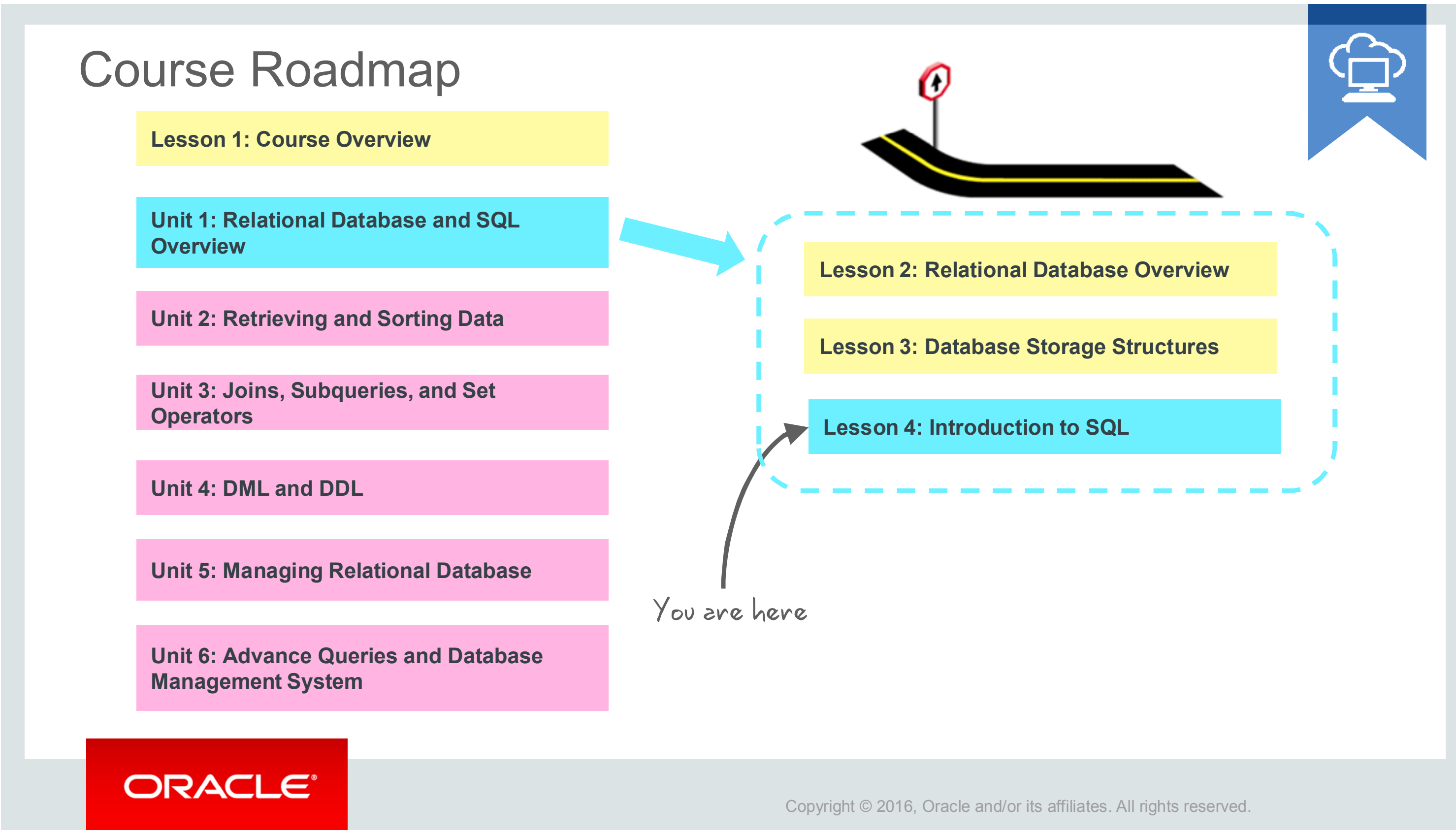
This practice covers database storage structures by using the following:

- Crossword puzzle
- Multiple-choice questions



In this practice, you will learn about the database storage structures by solving a crossword puzzle and answering multiple-choice questions.


Lesson 4: Introduction to SQL
ORACLE
Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

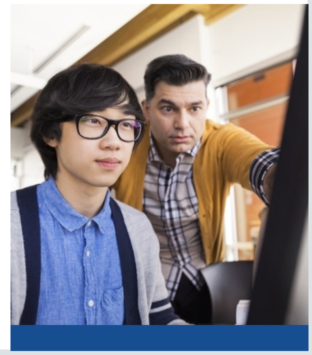


In Unit 1, you will learn about databases and database concepts. You will be introduced to relational databases, data storage concepts, and SQL.

Objectives

After completing this lesson, you should be able to:

- List the key concepts of SQL
- List the key concepts of PL/SQL
- Discuss the use case used in this course
- Describe the database schemas that are used in this course
- Identify the available user interface environments



This lesson introduces you to SQL and PL/SQL. You learn about the database schema and the tables that the course uses. The course also introduces you to tools such as SQL Developer.

Lesson Agenda

- Introduction to SQL
- Introduction to PL/SQL
- Overview of schemas and the use case used in this course
- Overview of the development environments available

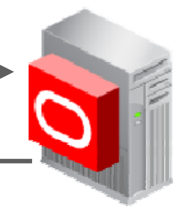
Using SQL to Query Your Database

Structured query language (SQL) is:

- The ANSI standard language for operating relational databases
- Efficient, easy to learn and use
- Functionally complete (With SQL, you can define, retrieve, and manipulate data in tables.)

```
SELECT department_name
FROM   departments;
```

DEPARTMENT_NAME
Administration
Marketing
Shipping
IT
Sales
Executive
Accounting
Contracting



Oracle server

In a relational database, you do not specify the access route to the tables, and you do not need to know how the data is arranged physically.

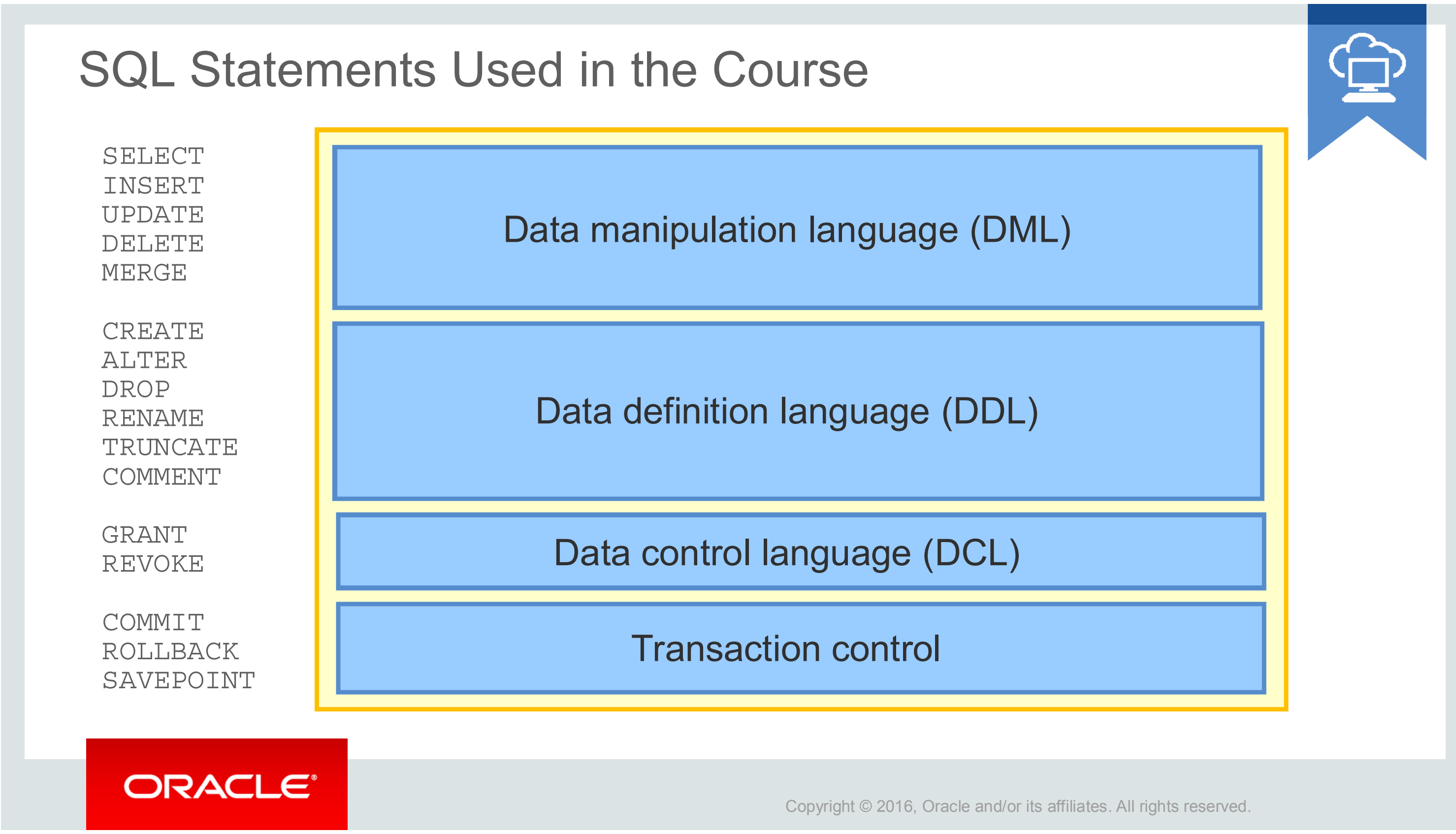
To access the database, you execute a SQL statement, which is the American National Standards Institute (ANSI) standard language for operating relational databases. SQL is also compliant to ISO Standard (SQL:1999).

SQL is a set of statements with which all programs and users access data in an Oracle Database. Application programs and Oracle tools often allow users access to the database without using SQL directly, but these applications, in turn, must use SQL when executing the user's request.

SQL provides statements for a variety of tasks, including:

- Querying data
- Inserting, updating, and deleting rows in a table
- Creating, replacing, altering, and dropping objects
- Controlling access to the database and its objects
- Guaranteeing database consistency and integrity

SQL unifies all of the preceding tasks in one consistent language and enables you to work with data at a logical level.



SQL Statements

SQL statements supported by Oracle comply with industry standards. Oracle Corporation ensures future compliance with evolving standards by actively involving key personnel in SQL standards committees. The industry-accepted committees are ANSI and International Standards Organization (ISO). Both ANSI and ISO have accepted SQL as the standard language for relational databases.

Statement	Description

`SELECT`
`INSERT`
`UPDATE`
`DELETE`
`MERGE`
	Retrieves data from the database, enters new rows, changes existing rows, and removes unwanted rows from tables in the database, respectively. Collectively known as *data manipulation language* (DML)

`CREATE`
`ALTER`
`DROP`
`RENAME`
`TRUNCATE`
`COMMENT`
	Sets up, changes, and removes data structures from tables. Collectively known as *data definition language* (DDL)

`GRANT` `REVOKE`	Provides or removes access rights to both the Oracle Database and the structures within it

`COMMIT`
`ROLLBACK`
`SAVEPOINT`
	Manages the changes made by DML statements. Changes to the data can be grouped together into logical transactions

Lesson Agenda

- Introduction to SQL
- Introduction to PL/SQL
- Overview of schemas and the use case used in this course
- Overview of the development environments available

ORACLE

Introduction to PL/SQL

PL/SQL:

- Stands for "Procedural Language extension to SQL"
- Is Oracle Corporation's standard data access language for relational databases
- Seamlessly integrates procedural constructs with SQL

ORACLE

SQL is the primary language used to access and modify data in relational databases. However, it has its own limitations.

Consider a problem statement: For every employee retrieved, check the department ID and salary. Depending on the department's performance and also the employee's salary, you may want to provide varying bonuses to the employees.

Looking at the problem, you know that you have to execute a SQL statement, collect the data, and apply logic to the data.

- One solution is to write a SQL statement for each department to give bonuses to the employees in that department. Remember that you also have to check the salary component before deciding the bonus amount. This makes it a little complicated.
- A more effective solution might include conditional statements. PL/SQL is designed to meet such requirements. It provides a programming extension to the already-existing SQL.

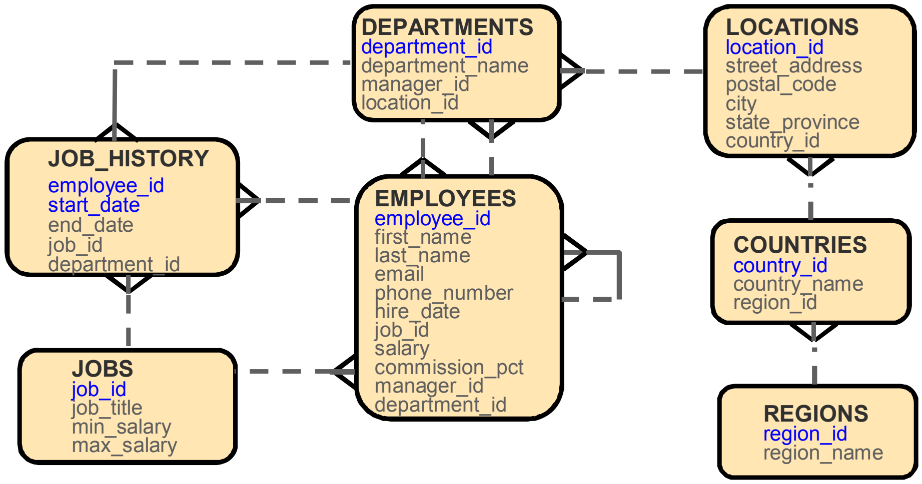
PL/SQL defines a block structure for writing code. Maintaining and debugging code is made easier with such a structure because you can easily understand the flow and execution of the program unit.

PL/SQL offers modern software engineering features such as data encapsulation, exception handling, information hiding, and object orientation. It brings state-of-the-art programming to the Oracle Server and toolset. PL/SQL provides all the procedural constructs that are available in any third-generation language (3GL).

Lesson Agenda

- Introduction to SQL
- Introduction to PL/SQL
- Overview of schemas and the use case used in this course
- Overview of the development environments available

Human Resources (HR) Schema for This Course



ORACLE

The Human Resources (`HR`) schema is part of the Oracle Sample Schemas that can be installed in an Oracle database. The practice sessions in this course use data from the `HR` schema.

The slide shows the Entity Relationship Diagram (ERD) of the HR schema. The fields marked in blue indicate a Primary Key.

Table Descriptions

- `REGIONS` contains rows that represent a region such as the Americas or Asia.
- `COUNTRIES` contains rows for countries, each of which is associated with a region.
- `LOCATIONS` contains the specific address of a specific office, warehouse, or production site of a company in a particular country.
- `DEPARTMENTS` shows details about the departments in which employees work. Each department may have a relationship representing the department manager in the `EMPLOYEES` table.
- `EMPLOYEES` contains details about each employee working for a department. Some employees may not be assigned to any department.
- `JOBS` contains the job types that can be held by each employee.
- `JOB_HISTORY` contains the job history of the employees. If an employee changes departments within a job or changes jobs within a department, a new row is inserted into this table with the old job information of the employee.

Tables Used in the Course

EMPLOYEES

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY
1	100	Steven	King	SKING	515.123.4567	17-JUN-03	AD_PRES	24000
2	101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-SEP-05	AD_VP	17000
3	102	Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-01	AD_VP	17000
4	103	Alexander	Hunold	AHUNOLD	590.423.4567	03-JAN-06	AC_MGR	12008
5	104	Bruce	Ernst	BERNST	590.423.4568	21-MAY-07	IT_PROG	6000
6	107	Diana	Lorentz	DLORENTZ	590.423.5567	07-FEB-07	IT_PROG	4200
7	124	Kevin	Mourgos	KMOURGOS	650.123.5234	16-NOV-07	ST_MAN	5800
8	141	Trenna	Rajs	TRAJS	650.121.8009	17-OCT-03	ST_CLERK	3500
9	142	Curtis	Davies	CDAVIES	650.121.2994	29-JAN-05	ST_CLERK	3100
10	143	Randall	Matos	RMATOS	650.121.2874	15-MAR-06	ST_CLERK	2600
11	144	Peter	Vargas	PVARGAS	650.121.2004	09-JUL-06	ST_CLERK	2500
12	149	Eleni	Zlotkey	EZLOTKEY	011.44.1344.429018	29-JAN-08	SA_MAN	10500
13	174	Ellen	Abel	EABEL	011.44.1644.429267	11-MAY-04	SA_REP	11000
14	176	Jonathon	Taylor	JTAYLOR	011.44.1644.429265	24-MAR-06	SA_REP	8600
15	178	Kimberely	Grant	KGRANT	011.44.1644.429263	24-MAY-07	SA_REP	7000
16	200	Jennifer	Whalen	JWHALEN	515.123.4444	17-SEP-03	AD_ASST	4400
17	201	Michael	Hartstein	MHARTSTE	515.123.5555	17-FEB-04	MK_MAN	13000
18	202	Pat	Fay	PFAY	603.123.6666	17-AUG-05	MK_REP	6000
19	205	Shelley	Higgins	SHIGGINS	515.123.8080	07-JUN-02	AC_MGR	12008
20	206	William	Gietz	WGIETZ	515.123.8181	07-JUN-02	AC_ACCOUNT	8300

JOB_GRADES

	GRADE_LEVEL	LOWEST_SAL	HIGHEST_SAL
1	A	1000	2999
2	B	3000	5999
3	C	6000	9999
4	D	10000	14999
5	E	15000	24999
6	F	25000	40000

DEPARTMENTS

	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	10	Administration	200	1700
2	20	Marketing	201	1800
3	50	Shipping	124	1500
4	60	IT	103	1400
5	80	Sales	149	2500
6	90	Executive	100	1700
7	110	Accounting	205	1700
8	190	Contracting	(null)	1700

ORACLE

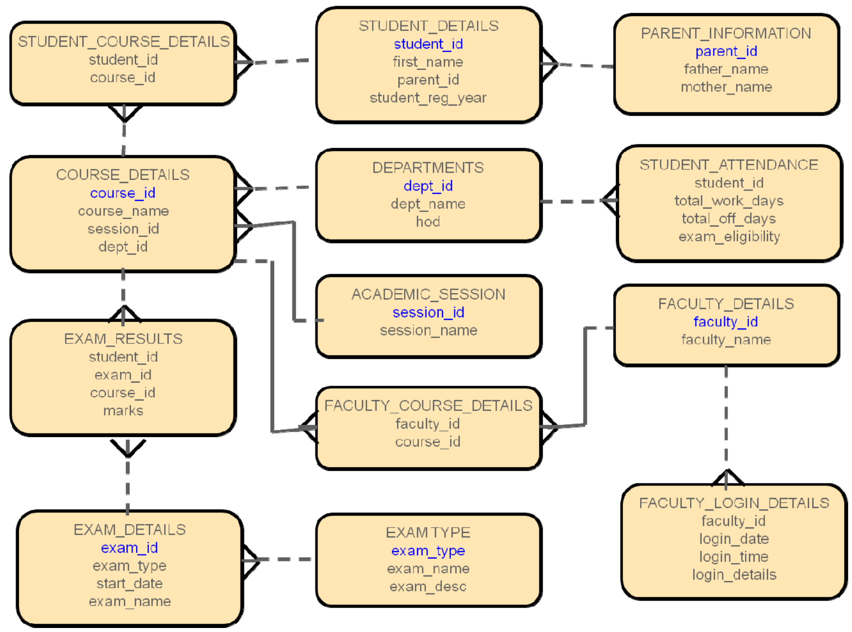
The following main tables are used in this course:

- EMPLOYEES table: Gives details of all the employees
- DEPARTMENTS table: Gives details of all the departments
- JOB_GRADES table: Gives details of salaries for various grades

Apart from these tables, you will use the other tables listed in the previous slide such as the LOCATIONS table and the JOB_HISTORY table.

Note: The structure and data for all the tables are provided in Appendix A.

Academic (AD) Schema



The Academic (AD) schema is designed for this course so that all the practices in this course use data from the AD schema. The slide shows the ERD of the AD schema. The fields marked in blue indicate a Primary Key.

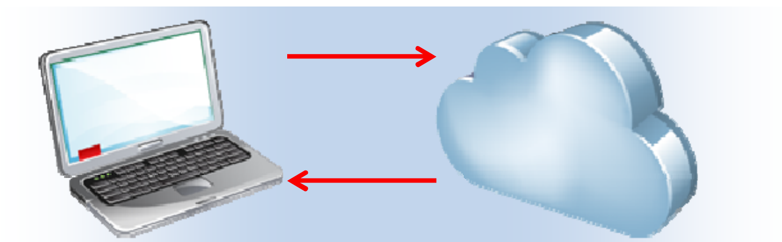
Table Descriptions

- ACADEMIC_SESSION contains rows that define various academic sessions such as Spring, Fall, and Summer.
- DEPARTMENTS shows details about departments, which offer courses to students in a given academic session.
- COURSE_DETAILS contains information about all the courses, each of which is associated with an academic session and a department.
- STUDENT_DETAILS contains the details about each student enrolled in the school for an academic session.
- PARENT_INFORMATION maintains information about the parents of the students enrolled in the school.
- STUDENT_COURSE_DETAILS contains details about the courses that each student has enrolled for in an academic session.
- STUDENT_ATTENDANCE contains rows to maintain the attendance details and exam eligibility of the students.
- FACULTY_DETAILS contains details about each faculty member working in the school.
- FACULTY_COURSE_DETAILS contains details about the courses taught by various faculty members.
- FACULTY_LOGIN_DETAILS contains rows to maintain the login information of faculty members.

- `EXAM_TYPE` contains rows to define all the exam types, such as ESSAY exams, LAB exams, and so on.
- `EXAM_DETAILS` contains details about the various exams conducted as part of academic sessions.
- `EXAM_RESULTS` contains data to maintain the results of students for all the exams they appear in.

Class Account Information

- Cloned HR account IDs are set up in the Database Cloud service for each of you.
- You are assigned one account ID for the lab practices.
- On your local machine, you should have SQL Developer installed so that you can access the Oracle Database Cloud service.



Course Environment

- You need to have these installed locally:
  - SQL Developer 4.1.3
  - Java Platform (JDK)
  - Internet Browser (Mozilla Firefox/Internet Explorer)
- On Oracle Cloud:
  - Oracle Database 12*c* on Database as a Service (DBaaS)

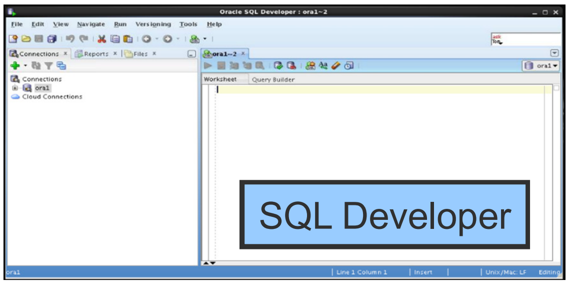
Lesson Agenda

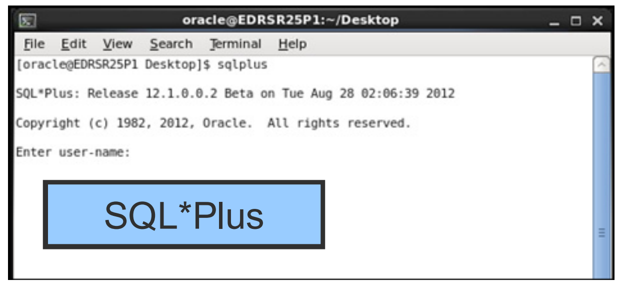
- Introduction to SQL
- Introduction to PL/SQL
- Overview of schemas and the use case used in this course
- Overview of the development environments available

SQL Development Environments

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

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





Oracle provides several tools that can be used to write PL/SQL code. Some of the development tools that are available for use in this course are:

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

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

What Is Oracle SQL Developer?

- Oracle SQL Developer is a free 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.
- You use SQL Developer in this course.

SQL Developer

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 maintain stored procedures, test SQL statements, and view optimizer plans.

SQL Developer, 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.

Specifications of SQL Developer

- Is shipped along with Oracle Database 12*c* Release 1
- Is developed in Java
- Supports Windows, Linux, and Mac OS X platforms
- Enables default connectivity using the Java Database Connectivity (JDBC) Thin driver
- Connects to Oracle Database version 9.2.0.1 and later
- Connects to Oracle Database on Cloud also

ORACLE

Oracle SQL Developer is shipped along with Oracle Database 12*c* 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 JDBC Thin driver, and therefore, no Oracle Home is required. SQL Developer does not require an installer and you need to simply unzip the downloaded file. With SQL Developer, users can connect to Oracle Databases 9.2.0.1 and later, and all Oracle database editions, including Express Edition.

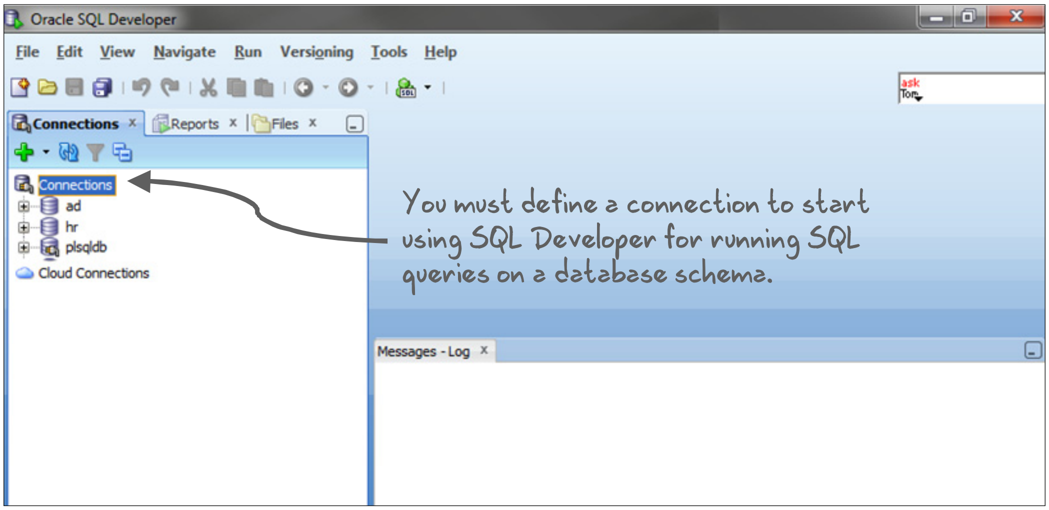
Note

- For Oracle Database 12*c* Release 1, you will have to download and install SQL Developer. SQL Developer can be downloaded free from the following link:

  http://www.oracle.com/technetwork/developer-tools/sql-developer/downloads/index.html
- For instructions on how to install SQL Developer, see the website at:

  http://www.oracle.com/technetwork/developer-tools/sql-developer/overview/index.html

SQL Developer 4.1.3 Interface



The SQL Developer interface contains three main navigation tabs:

- **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, as well as 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:** Contains options to invoke 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
  - Database on Oracle Cloud
- SQL Developer automatically imports any connections defined in the `tnsnames.ora` file on your system.
- Each additional database connection created is listed in the Connections Navigator hierarchy.

ORACLE

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.

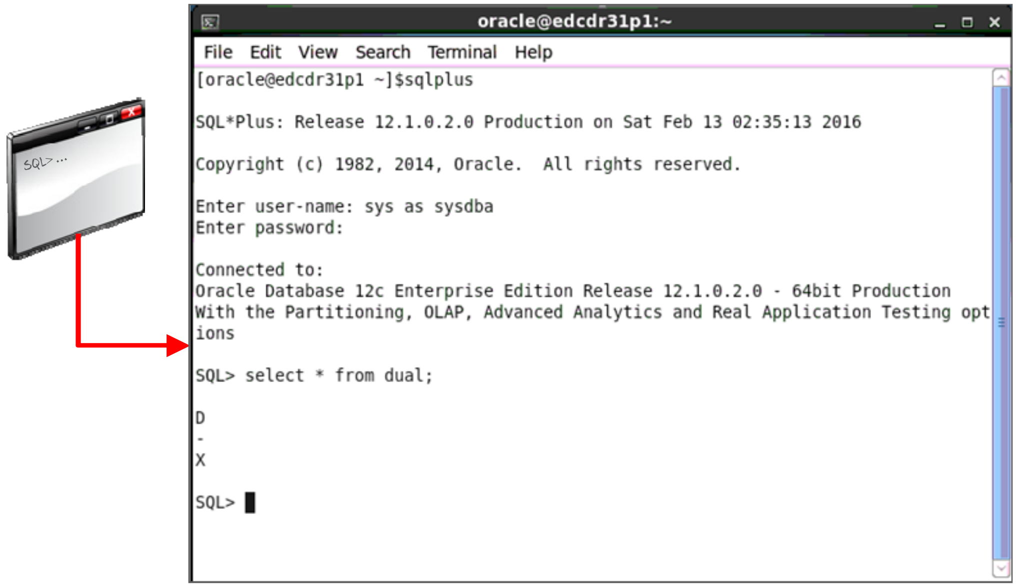
By default, the `tnsnames.ora` file is located in the `$ORACLE_HOME/network/admin` directory, but it can also be in the directory specified by the `TNS_ADMIN` environment variable or registry value. When you start SQL Developer and open the Database Connections dialog box, SQL Developer automatically 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 them.

You can create additional connections as different users to the same database or to connect to different databases.

Coding SQL in SQL*Plus



ORACLE

Oracle SQL*Plus is a command-line interface that enables you to submit SQL statements and receive the results in an application or a command window.

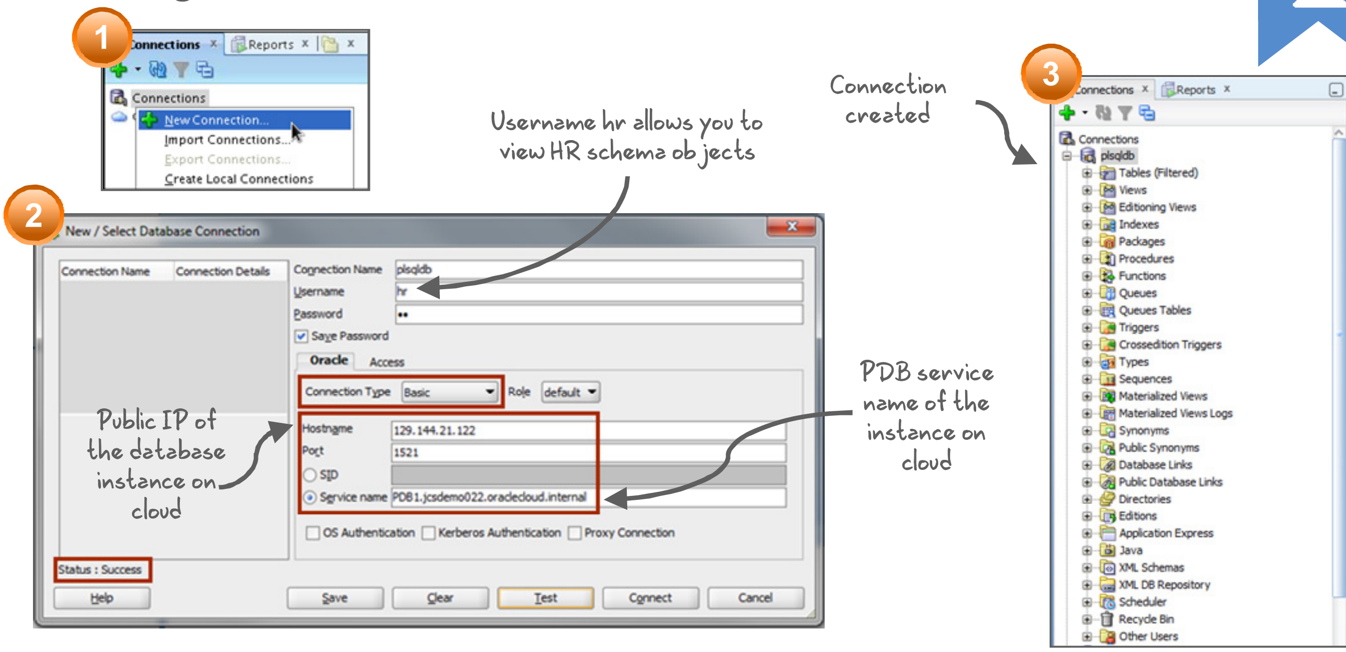
SQL*Plus is:

- Shipped with the database
- Installed on a client and on the database server system
- Accessed by using an icon or the command line

Note

- In a Linux environment, you can launch SQL*Plus by establishing a connection to Oracle Cloud Database instance and creating a SSH tunnel for port using the ssh utility on Linux.
- For more details on accessing SQL *Plus using the ssh utility, refer to Creating an SSH Tunnel Using the ssh Utility on Linux on Oracle Help Center.

Creating a Connection to Database on Oracle Cloud



ORACLE

To create a connection to the database on the cloud, perform the following steps:

1. On the Connections tab, 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. Select the connection type as **Basic**.
    c. Enter the connection details as follows:
        i) **Hostname**: Public IP listed in the Oracle Cloud DBaaS for the database instance
        ii) **Port**: 1521
        iii) **Service name**: PDB service name of the instance on the cloud in the format given below:
        ```
        <PDB name of the instance>.<Identity Domain
        Name>.oraclecloud.internal
        ```
    d. Click Test to ensure that the connection has been set correctly.
    e. 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—for example, dependencies, details, and statistics.

Quiz

Which four of the following options are DDL statements?

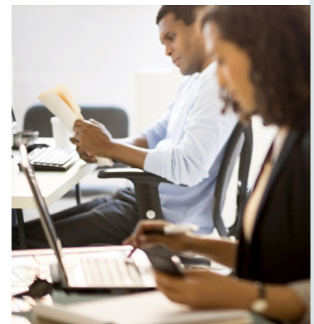
a. `CREATE`
b. `ALTER`
c. `TRUNCATE`
d. `DELETE`
e. `UPDATE`
f. `DROP`

Answer: a, b, c, f

Summary

In this lesson, you should have learned how to:

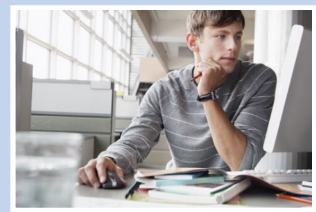
- Discuss the key concepts of SQL
- Describe the database schemas that are used in the course
- Identify the available user interface environments that can be used in this course
- Describe the salient features of Oracle Cloud
- Reference the available documentation and other resources



Practice 4: Overview

This practice covers the following topics:

- Starting SQL Developer
- Creating a new database connection
- Browsing the Academic (AD) schema tables
- Setting a SQL Developer preference



ORACLE

In this practice, you use SQL Developer to execute SQL statements to examine data in the `AD` schema. You also create a simple anonymous block.

Note: All written practices use SQL Developer as the development environment.

Lesson 5: Retrieving Data Using the SQL SELECT Statement

Course Roadmap

- Lesson 1: Course Overview
- Unit 1: Relational Database and SQL Overview
- Unit 2: Retrieving and Sorting Data
- Unit 3: Joins, Subqueries, and Set Operators
- Unit 4: DML and DDL
- Unit 5: Managing Relational Database
- Unit 6: Advance Queries and Database Management System

You are here

- Lesson 5: Retrieving Data Using SQL `SELECT` Statement
- Lesson 6: Restricting and Sorting Data
- Lesson 7: Using Single-Row Functions
- Lesson 8: Using Conversion Functions
- Lesson 9: Using Conditional Expressions

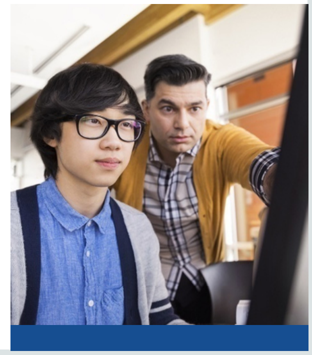
ORACLE

In Unit 2, you will dive into the concepts of SQL. You will learn to use the SQL `SELECT` statement to retrieve data from database tables and restrict and sort the retrieved data. You will also learn about single-row functions, conversion functions, and conditional expressions in SQL.

Objectives

After completing this lesson, you should be able to:

- List the capabilities of SQL `SELECT` statements
- Execute a basic `SELECT` statement



To extract data from the database, you need to use the SQL `SELECT` statement. However, you may need to restrict the columns that are displayed. This lesson describes the `SELECT` statement that is needed to perform these actions. Further, you may want to create `SELECT` statements that can be used more than once.

Lesson Agenda

- Capabilities of SQL `SELECT` statements
- Arithmetic expressions and `NULL` values in the `SELECT` statement
- Column aliases
- Use of concatenation operator, literal character strings, alternative quote operator, and the `DISTINCT` keyword
- `DESCRIBE` command

ORACLE

Basic `SELECT` Statement

SELECT identifies the columns to be displayed.

```
SELECT   *|{[DISTINCT] column [alias],...}
FROM     table;
```

FROM identifies the table containing those columns.

In its simplest form, a `SELECT` statement must include the following:

- A `SELECT` clause, which specifies the columns to be displayed
- A `FROM` clause, which identifies the table containing the columns that are listed in the `SELECT` clause

In the syntax:

`SELECT`	Is a keyword to select one or more columns
`*`	Selects all columns
`DISTINCT`	Suppresses duplicates
`column|expression`	Selects the named column or the expression
`alias`	Gives different headings to the selected columns
`FROM` *`table`*	Specifies the table containing the columns

Note: Throughout this course, the words *keyword*, *clause*, and *statement* are used as follows:

- A *keyword* refers to an individual SQL element—for example, `SELECT` and `FROM` are keywords.
- A *clause* is a part of a SQL statement—for example, `SELECT employee_id, last_name`, and so on.
- A *statement* is a combination of two or more clauses—for example, `SELECT * FROM employees`.

Selecting All Columns

```
SELECT *
FROM   departments;
```

	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	10	Administration	200	1700
2	20	Marketing	201	1800
3	50	Shipping	124	1500
4	60	IT	103	1400
5	80	Sales	149	2500
6	90	Executive	100	1700
7	110	Accounting	205	1700
8	190	Contracting	(null)	1700

ORACLE

You can display all columns of data in a table by using the SELECT keyword followed by an asterisk (*). In the example in the slide, the DEPARTMENTS table contains four columns: DEPARTMENT_ID, DEPARTMENT_NAME, MANAGER_ID, and LOCATION_ID. The table contains eight rows, one for each department.

You can also display all columns in the table by listing them after the SELECT keyword. For example, the following SQL statement (like the example in the slide) displays all columns and all rows of the DEPARTMENTS table:

```
SELECT   department_id, department_name, manager_id, location_id
FROM     departments;
```

Note: In SQL Developer, you can enter your SQL statement in a SQL Worksheet and click the “Execute Statement” icon or press [F9] to execute the statement. The output displayed on the Results tabbed page appears as shown in the slide.

Selecting Specific Columns

```
SELECT department_id, location_id
FROM   departments;
```

	DEPARTMENT_ID	LOCATION_ID
1	10	1700
2	20	1800
3	50	1500
4	60	1400
5	80	2500
6	90	1700
7	110	1700
8	190	1700

ORACLE

You can use the SELECT statement to display specific columns of the table by specifying the column names, separated by commas. The example in the slide displays all the department numbers and location numbers from the DEPARTMENTS table.

In the SELECT clause, specify the columns that you want in the order in which you want them to appear in the output. For example, to display location before department number (from left to right), you use the following statement:

```
SELECT location_id, department_id
FROM   departments;
```

Writing SQL Statements

- SQL statements are not case-sensitive.
- SQL statements can be entered on one or more lines.
- Keywords cannot be abbreviated or split across lines.
- Clauses are usually placed on separate lines.
- Indents are used to enhance readability.
- In SQL Developer, SQL statements can be optionally terminated by a semicolon (;). Semicolons are required when you execute multiple SQL statements.
- In SQL*Plus, you are required to end each SQL statement with a semicolon (;).

ORACLE

Writing SQL Statements

By using the following simple rules and guidelines, you can construct valid statements that are easy to read and edit:

- SQL statements are not case-sensitive (unless indicated).
- SQL statements can be entered on one or many lines.
- Keywords cannot be split across lines or abbreviated.
- Clauses are usually placed on separate lines for readability and ease of editing.
- Indents should be used to make code more readable.
- Keywords typically are entered in uppercase; all other words, such as table names and columns names, are entered in lowercase.

Executing SQL Statements

In SQL Developer, click the Run Script icon or press [F5] to run the command or commands in the SQL Worksheet. You can also click the Execute Statement icon or press [F9] to run a SQL statement in the SQL Worksheet. The Execute Statement icon executes the statement at the cursor in the Enter SQL Statement box while the Run Script icon executes all the statements in the Enter SQL Statement box. The Execute Statement icon displays the output of the query on the Results tabbed page, whereas the Run Script icon shows the output on the Script Output tabbed page.

Column Heading Defaults for Output

- SQL Developer:
  - Default heading alignment: Left-aligned
  - Default heading display: Uppercase

```
SELECT last_name, hire_date, salary
FROM   employees;
```

Uppercase

	LAST_NAME	HIRE_DATE	SALARY
1	King	17-JUN-11	24000
2	Kochhar	21-SEP-13	17000
3	De Haan	13-JAN-09	17000
4	Hunold	03-JAN-14	9000
5	Ernst	21-MAY-15	6000
6	Lorentz	07-FEB-15	4200
7	Mourgos	16-NOV-15	5800
8	Rajs	17-OCT-11	3500
9	Davies	29-JAN-13	3100
10	Matos	15-MAR-14	2600

Left-aligned

ORACLE

In SQL Developer, column headings are displayed in uppercase and are left-aligned. You can run the following command to observe the output:

```
SELECT last_name, hire_date, salary
FROM   employees;
```

You can override the column heading display with an alias. Column aliases are covered later in this lesson.

Lesson Agenda

- Capabilities of SQL `SELECT` statements
- Arithmetic expressions and `NULL` values in the `SELECT` statement
- Column aliases
- Use of concatenation operator, literal character strings, alternative quote operator, and the `DISTINCT` keyword
- `DESCRIBE` command

ORACLE

Arithmetic Expressions

Create expressions with Number and Date data by using arithmetic operators.

Operator	Description
+	Add
-	Subtract
*	Multiply
/	Divide

ORACLE

You may need to modify the way in which data is displayed, or you may want to perform calculations or look at what-if scenarios. All these are possible by using arithmetic expressions. An arithmetic expression can contain column names, constant numeric values, and the arithmetic operators.

Arithmetic Operators

The slide lists the arithmetic operators that are available in SQL. You can use arithmetic operators in any clause of a SQL statement (except the `FROM` clause).

Note: With the `DATE` and `TIMESTAMP` data types, you can use the addition and subtraction operators only.

Using Arithmetic Operators

```
SELECT last_name, salary, salary + 300
FROM   employees;
```

	LAST_NAME	SALARY	SALARY+300
1	King	24000	24300
2	Kochhar	17000	17300
3	De Haan	17000	17300
4	Hunold	9000	9300
5	Ernst	6000	6300
6	Lorentz	4200	4500
7	Mourgos	5800	6100
8	Rajs	3500	3800
9	Davies	3100	3400
10	Matos	2600	2900

...

The example in the slide uses the addition operator to calculate a salary increase of $300 for all employees. The slide also displays a `SALARY+300` column in the output.

Note that the resultant calculated column, `SALARY+300`, is not a new column in the `EMPLOYEES` table; it is for display only. By default, the name of a new column comes from the calculation that generated it—in this case, `salary+300`.

Note: The Oracle server ignores blank spaces before and after the arithmetic operator.

Rules of Precedence

- Multiplication and division occur before addition and subtraction.
- Operators of the same priority are evaluated from left to right.
- Parentheses are used to override the default precedence or to clarify the statement.

Operator Precedence

```
SELECT last_name, salary, 12*salary+100
FROM   employees;
```

1

	LAST_NAME	SALARY	12*SALARY+100
1	King	24000	288100
2	Kochhar	17000	204100
3	De Haan	17000	204100
4	Hunold	9000	108100

...

```
SELECT last_name, salary, 12*(salary+100)
FROM   employees;
```

2

	LAST_NAME	SALARY	12*(SALARY+100)
1	King	24000	289200
2	Kochhar	17000	205200
3	De Haan	17000	205200
4	Hunold	9000	109200

...

ORACLE

The first example in the slide displays the last name, salary, and annual compensation of employees. It calculates the annual compensation by multiplying the monthly salary with 12, plus a one-time bonus of $100. Note that multiplication is performed before addition.

Note: Use parentheses to reinforce the standard order of precedence and to improve clarity. For example, the expression in the slide can be written as `(12*salary)+100` with no change in the result.

Using Parentheses

You can override the rules of precedence by using parentheses to specify the desired order in which the operators are to be executed.

The second example in the slide displays the last name, salary, and annual compensation of employees. It calculates the annual compensation as follows: adding a monthly bonus of $100 to the monthly salary, and then multiplying that subtotal with 12. Because of the parentheses, addition takes priority over multiplication.

Defining a Null Value

```
SELECT last_name, job_id, salary, commission_pct
FROM   employees;
```

	LAST_NAME	JOB_ID	SALARY	COMMISSION_PCT
1	King	AD_PRES	24000	(null)
2	Kochhar	AD_VP	17000	(null)
3	De Haan	AD_VP	17000	(null)
...				
12	Zlotkey	SA_MAN	10500	0.2
13	Abel	SA_REP	11000	0.3
14	Taylor	SA_REP	8600	0.2
15	Grant	SA_REP	7000	0.15
...				
18	Fay	MK_REP	6000	(null)
19	Higgins	AC_MGR	12008	(null)
20	Gietz	AC_ACCOUNT	8300	(null)

Null is a value that is unavailable, unassigned, unknown, or inapplicable.

Null is not the same as zero or a blank space.

ORACLE

If a row lacks a data value for a particular column, that value is said to be NULL or to contain a null.

Columns with NULL value can be selected in a SELECT query and can be the part of an arithmetic expression. Any arithmetic expression using NULL values results into NULL.

Columns of any data type can contain nulls. However, some constraints (NOT NULL and PRIMARY KEY) prevent nulls from being used in a column.

In the slide example, notice that only a sales manager or sales representative can earn a commission in the COMMISSION_PCT column of the EMPLOYEES table. Other employees are not entitled to earn commissions. A null represents that fact.

Note: By default, SQL Developer uses the literal (null) to identify null values. However, you can set it to something more relevant to you. To do so, select Preferences from the Tools menu. In the Preferences dialog box, expand the Database node. Click Advanced Parameters and in the right pane, for “Display Null value As,” enter the appropriate value.

Lesson Agenda

- Capabilities of SQL `SELECT` statements
- Arithmetic expressions and `NULL` values in the `SELECT` statement
- Column aliases
- Use of concatenation operator, literal character strings, alternative quote operator, and the `DISTINCT` keyword
- `DESCRIBE` command

Defining a Column Alias

A column alias:

- Renames a column heading
- Is useful with calculations
- Immediately follows the column name (there can also be the optional `AS` keyword between the column name and the alias)
- Requires double quotation marks if it contains spaces or special characters, or if it is case-sensitive

When displaying the result of a query, SQL Developer normally uses the name of the selected column as the column heading. This heading may not be descriptive and, therefore, may be difficult to understand. You can change a column heading by using a column alias.

Specify the alias after the column in the `SELECT` list using blank space as a separator. By default, alias headings appear in uppercase. If the alias contains spaces or special characters (such as `-`, `!`, _), or if it is case-sensitive, enclose the alias in double quotation marks (`" "`).

Using Column Aliases

```
SELECT last_name AS name, commission_pct comm
FROM   employees;
```

	NAME	COMM
1	King	(null)
2	Kochhar	(null)
3	De Haan	(null)
4	Hunold	(null)

...

```
SELECT last_name "Name" , salary*12 "Annual Salary"
FROM   employees;
```

	Name	Annual Salary
1	King	288000
2	Kochhar	204000
3	De Haan	204000
4	Hunold	108000

...

The first example displays the names and the commission percentages of all the employees. Note that the optional `AS` keyword has been used before the column alias name. The result of the query is the same whether the `AS` keyword is used or not. Also, note that the SQL statement has the column aliases, `name` and `comm`, in lowercase, whereas the result of the query displays the column headings in uppercase. As mentioned in the preceding slide, column headings appear in uppercase by default.

The second example displays the last names and annual salaries of all the employees. Because `Annual Salary` contains a space, it has been enclosed in double quotation marks. Note that the column heading in the output is exactly the same as the column alias.

Lesson Agenda

- Capabilities of SQL `SELECT` statements
- Arithmetic expressions and `NULL` values in the `SELECT` statement
- Column aliases
- Use of concatenation operator, literal character strings, alternative quote operator, and the `DISTINCT` keyword
- `DESCRIBE` command

ORACLE

Concatenation Operator

A concatenation operator:

- Links columns or character strings to other columns
- Is represented by two vertical bars (||)
- Creates a resultant column that is a character expression

```
SELECT   last_name||job_id AS "Employees"
FROM     employees;
```

	Employees
1	AbelSA_REP
2	DaviesST_CLERK
3	De HaanAD_VP
4	ErnstIT_PROG
5	FayMK_REP
6	GietzAC_ACCOUNT
7	GrantSA_REP
8	HartsteinMK_MAN

...

ORACLE

You can link columns to other columns, arithmetic expressions, or constant values to create a character expression by using the concatenation operator (||). Columns on either side of the operator are combined to make a single output column.

In the example, `LAST_NAME` and `JOB_ID` are concatenated, and given the alias `Employees`. Note that the last name of the employee and the job code are combined to make a single output column.

The `AS` keyword before the alias name makes the `SELECT` clause easier to read.

Null Values with the Concatenation Operator

If you concatenate a null value with a character string, the result is a character string. `LAST_NAME || NULL` results in `LAST_NAME`.

Literal Character Strings

- A literal is a character, a number, or a date that is included in the `SELECT` statement.
- Date and character literal values must be enclosed within single quotation marks.
- Each character string is output once for each row returned.

A literal is a character, a number, or a date that is included in the `SELECT` list. It is not a column name or a column alias. It is printed for each row returned. Literal strings of free-format text can be included in the query result and are treated the same as a column in the `SELECT` list.

The date and character literals *must* be enclosed within single quotation marks (`''`); number literals need not be enclosed in a similar manner.

Using Literal Character Strings

```
SELECT last_name ||' is a '||job_id
       AS "Employee Details"
FROM   employees;
```

	Employee Details
1	Abel is a SA_REP
2	Davies is a ST_CLERK
3	De Haan is a AD_VP
4	Ernst is a IT_PROG
5	Fay is a MK_REP
6	Gietz is a AC_ACCOUNT
7	Grant is a SA_REP
8	Hartstein is a MK_MAN
9	Higgins is a AC_MGR
10	Hunold is a IT_PROG
11	King is a AD_PRES

...

ORACLE

The example in the slide displays the last names and job codes of all employees. The column has the heading Employee Details. Note the spaces between the single quotation marks in the `SELECT` statement. The spaces improve the readability of the output.

In the following example, the last name and salary for each employee are concatenated with a literal, to give the returned rows more meaning:

```
SELECT last_name ||': 1 Month salary = '||salary Monthly
FROM   employees;
```

Alternative Quote (q) Operator

- Specify your own quotation mark delimiter.
- Select any delimiter.
- Increase readability and usability.

```
SELECT department_name || q'[ Department's Manager Id: ]'
       || manager_id
       AS "Department and Manager"
FROM departments;
```

	Department and Manager
1	Administration Department's Manager Id: 200
2	Marketing Department's Manager Id: 201
3	Shipping Department's Manager Id: 124
4	IT Department's Manager Id: 103
5	Sales Department's Manager Id: 149
6	Executive Department's Manager Id: 100
7	Accounting Department's Manager Id: 205
8	Contracting Department's Manager Id:

ORACLE

Many SQL statements use character literals in expressions or conditions. If the literal itself contains a single quotation mark, you can use the quote (q) operator and select your own quotation mark delimiter.

You can choose any convenient delimiter, single-byte or multibyte, or any of the following character pairs: [ ], { }, ( ), or < >.

In the example shown, the string contains a single quotation mark, which is normally interpreted as a delimiter of a character string. By using the q operator, however, brackets [ ] are used as the quotation mark delimiters. The string between the brackets delimiters is interpreted as a literal character string.

Using the DISTINCT keyword

The default display of queries is all rows, including duplicate rows.

1

```
SELECT department_id
FROM   employees;
```

	DEPARTMENT_ID
1	90
2	90
3	90
4	60
5	60
6	60
7	50
8	50

...

2

```
SELECT DISTINCT department_id
FROM   employees;
```

	DEPARTMENT_ID
1	(null)
2	90
3	20
4	110
5	50
6	80
7	60
8	10

ORACLE

Unless you indicate otherwise, SQL displays the results of a query without eliminating the duplicate rows. The first example in the slide displays all the department numbers from the EMPLOYEES table. Note that the department numbers are repeated.

To eliminate duplicate rows in the result, include the DISTINCT keyword in the SELECT clause immediately after the SELECT keyword. In the second example in the slide, the EMPLOYEES table actually contains 20 rows, but there are only seven unique department numbers in the table.

Using DISTINCT with Multiple Columns

You can specify multiple columns after the DISTINCT qualifier.

```
SELECT DISTINCT department_id, job_id
FROM   employees;
```

	DEPARTMENT_ID	JOB_ID
1	110	AC_ACCOUNT
2	90	AD_VP
3	50	ST_CLERK
4	80	SA_REP
5	50	ST_MAN
6	80	SA_MAN
7	110	AC_MGR

...

The result is a distinct combination of columns.

You can specify multiple columns after the DISTINCT qualifier. The DISTINCT qualifier affects all the selected columns, and the result is every distinct combination of the columns.

Syntax

```
SELECT   DISTINCT col1, col2, ….,coln
FROM     table;
```

where

`col1, col2, ….,coln`: The combination of columns that are to be displayed distinctly

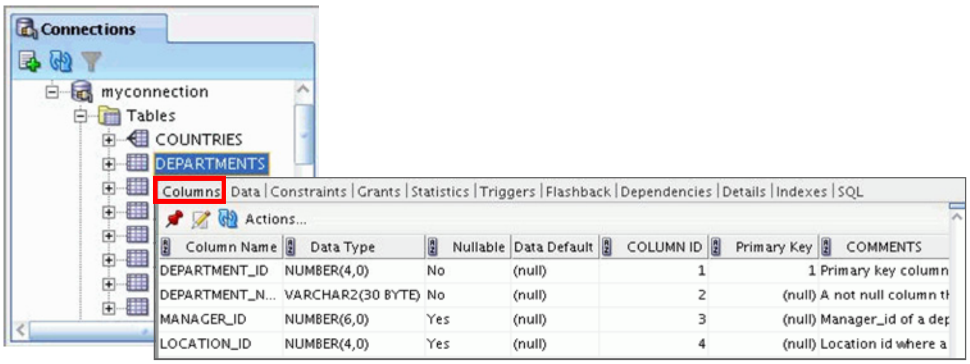
Lesson Agenda

- Capabilities of SQL `SELECT` statements
- Arithmetic expressions and `NULL` values in the `SELECT` statement
- Column aliases
- Use of concatenation operator, literal character strings, alternative quote operator, and the `DISTINCT` keyword
- `DESCRIBE` command

Displaying Table Structure

- Use the `DESCRIBE` command to display the structure of a table.
- Alternatively, select the table in the Connections tree and use the Columns tab to view the table structure.

```
DESC[RIBE] tablename
```



You can display the structure of a table by using the `DESCRIBE` command. The command displays the column names and the data types, and it shows you whether a column *must* contain data (that is, whether the column has a `NOT NULL` constraint).

In the syntax, *`table name`* is the name of any existing table, view, or synonym that is accessible to the user.

Using the SQL Developer GUI interface, you can select the table in the Connections tree and use the Columns tab to view the table structure.

Note: `DESCRIBE` is a SQL*Plus command supported by SQL Developer. It is abbreviated as `DESC`.

Using the DESCRIBE Command

```
DESCRIBE employees
```

```
DESCRIBE Employees
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)
```


The example in the slide displays information about the structure of the EMPLOYEES table using the DESCRIBE command.

In the resulting display, *Null* indicates that the values for this column may be unknown. NOT NULL indicates that a column must contain data. *Type* displays the data type for a column.

The data types are described in the following table:

Data Type	Description
NUMBER(*p*,*s*)	Number value having a maximum number of digits *p*, with *s* digits to the right of the decimal point
VARCHAR2(*s*)	Variable-length character value of maximum size *s*
DATE	Date and time value between January 1, 4712 B.C. and December 31, 9999 A.D.

Quiz

Identify the `SELECT` statements that execute successfully.

a.
```
SELECT first_name, last_name, job_id, salary*12
 AS Yearly Sal
FROM   employees;
```

b.
```
SELECT first_name, last_name, job_id, salary*12
 "yearly sal"
FROM   employees;
```

c.
```
SELECT first_name, last_name, job_id, salary AS
 "yearly sal"
FROM   employees;
```

d.
```
SELECT first_name+last_name AS name, job_Id,
 salary*12 yearly sal
FROM   employees;
```

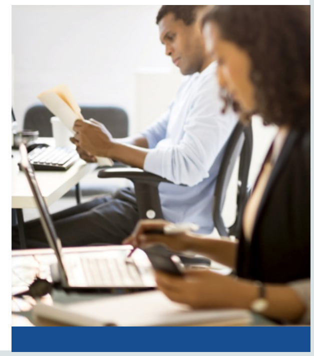
ORACLE

Answer: b, c

Summary

In this lesson, you should have learned how to write a SELECT statement that:

- Returns all rows and columns from a table
- Returns specified columns from a table
- Uses column aliases to display more descriptive column headings



ORACLE

In this lesson, you should have learned how to retrieve data from a database table with the SELECT statement.

```
SELECT  *|{[DISTINCT] column [alias],...}
FROM    table;
```

In the syntax:

SELECT	Is a keyword to select one or more columns
*	Selects all columns
DISTINCT	Suppresses duplicates
column|expression	Selects the named column or the expression
alias	Gives different headings to the selected columns
FROM *table*	Specifies the table containing the columns

Practice 5: Overview

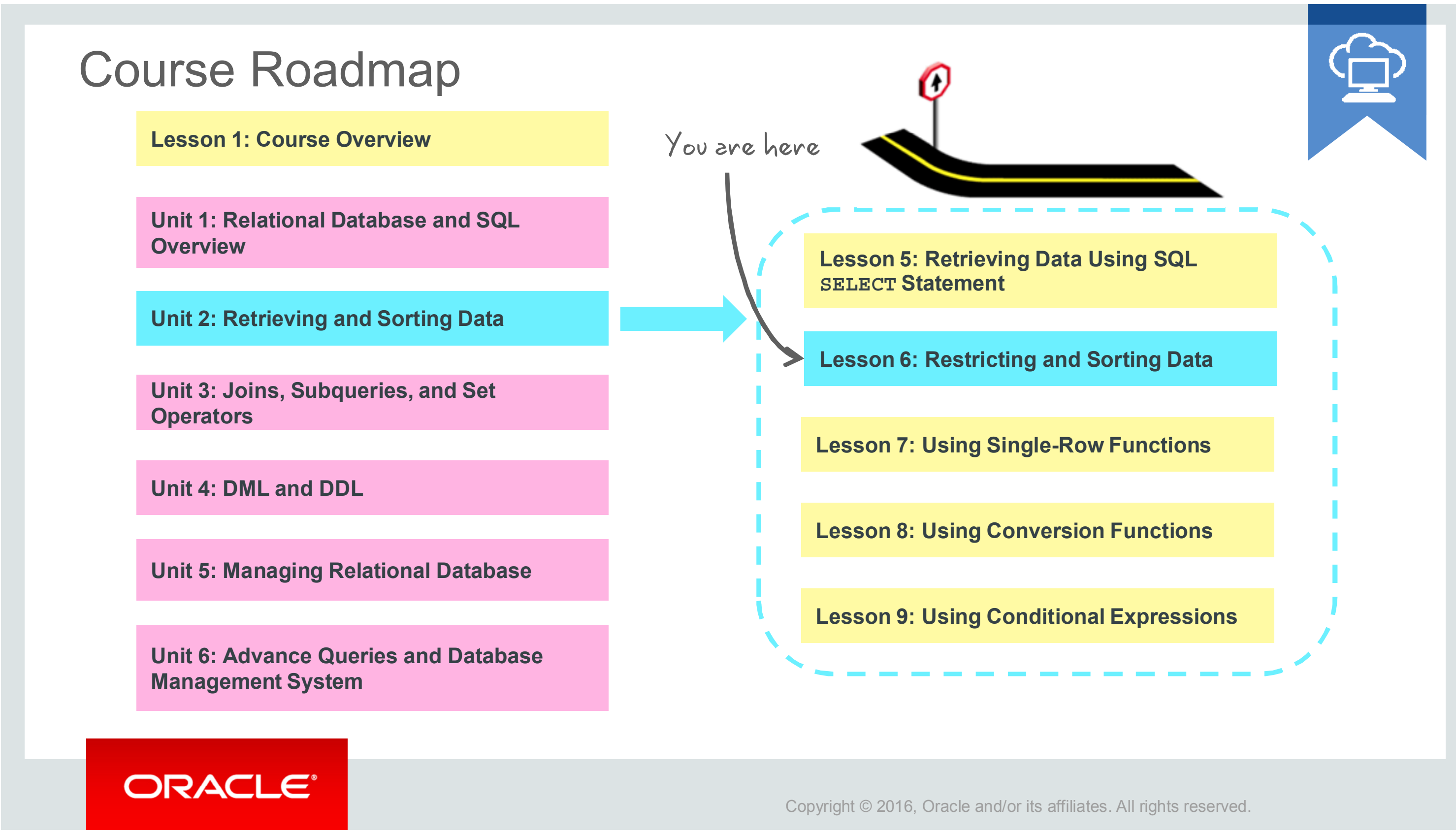
This practice covers the following topics:

- Selecting all data from different tables
- Describing the structure of tables
- Performing arithmetic calculations and specifying column names

ORACLE

In this practice, you write simple `SELECT` queries. The queries cover most of the `SELECT` clauses and operations that you learned in this lesson.


Lesson 6: Restricting and Sorting Data
ORACLE
Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

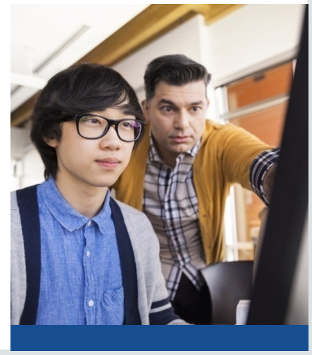


In Unit 2, you will dive into the concepts of SQL. You will learn to use the SQL SELECT statement to retrieve data from database tables and restrict and sort the retrieved data. You will also learn about single-row functions, conversion functions, and conditional expressions in SQL.

Objectives

After completing this lesson, you should be able to:

- Limit the rows that are retrieved by a query
- Sort the rows that are retrieved by a query
- Use ampersand substitution to restrict and sort output at run time



When retrieving data from the database, you may need to do the following:

- Restrict the rows of data that are displayed.
- Specify the order in which the rows are displayed.

This lesson explains the SQL statements that you use to perform the actions listed in the slide.

Lesson Agenda

- Limiting rows with:
  - `WHERE` clause
  - Comparison operators using `=`, `<=`, `BETWEEN`, `IN`, `LIKE`, and `NULL` conditions
  - Logical conditions using `AND`, `OR`, and `NOT` operators
- Rules of precedence for operators in an expression
- Sorting rows using the `ORDER BY` clause
- SQL row limiting clause in a query
- Substitution variables
- `DEFINE` and `VERIFY` commands

Limiting Rows by Using a Selection

EMPLOYEES

	EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
1	100	King	AD_PRES	90
2	101	Kochhar	AD_VP	90
3	102	De Haan	AD_VP	90
4	103	Hunold	IT_PROG	60
5	104	Ernst	IT_PROG	60
6	107	Lorentz	IT_PROG	60

. . .

"Retrieve all
employees in department 90"

	EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
1	100	King	AD_PRES	90
2	101	Kochhar	AD_VP	90
3	102	De Haan	AD_VP	90

ORACLE

In the example in the slide, assume that you want to display all the employees in department 90. The rows with a value of 90 in the `DEPARTMENT_ID` column are the only ones that are returned. This method of restriction is the basis of the `WHERE` clause in SQL.

Limiting Rows That Are Selected

```
SELECT   *|{[DISTINCT] column [alias],...}
FROM     table
[WHERE logical expression(s)];
```

Restrict the rows that are returned by using the WHERE clause.

The WHERE clause follows the FROM clause.

ORACLE

You can restrict the rows that are returned from the query by using the `WHERE` clause. The `WHERE` clause contains a condition that must be met and it directly follows the `FROM` clause. If the condition is true, the row meeting the condition is returned.

In the syntax:

`WHERE`	Restricts the query to rows that meet a condition
`logical expression`	Is composed of column names, constants, and a comparison operator. It specifies a combination of one or more expressions and Boolean operators, and returns a value of `TRUE`, `FALSE`, or `UNKNOWN`.

The `WHERE` clause can compare values in columns, literals, arithmetic expressions, or functions. It consists of three elements:

- Column name
- Comparison condition
- Column name, constant, or list of values

Using the WHERE Clause

```
SELECT employee_id, last_name, job_id, department_id
FROM   employees
WHERE  department_id = 90 ;
```

	EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
1	100	King	AD_PRES	90
2	101	Kochhar	AD_VP	90
3	102	De Haan	AD_VP	90

ORACLE

In the example, the SELECT statement retrieves the employee ID, last name, job ID, and department number of all employees who are in department 90.

Note: You cannot use column alias in the WHERE clause.

Character Strings and Dates

- Character strings and date values are enclosed within single quotation marks.
- Character values are case-sensitive and date values are format-sensitive.
- The default date display format is `DD-MON-RR`.

```
SELECT last_name, job_id, department_id
FROM   employees
WHERE  last_name = 'Whalen';
```

	LAST_NAME	JOB_ID	DEPARTMENT_ID
1	Whalen	AD_ASST	10

```
SELECT last_name
FROM   employees
WHERE  hire_date = '17-OCT-11' ;
```

	LAST_NAME
1	Rajs

ORACLE

Character strings and dates in the `WHERE` clause must be enclosed within single quotation marks (`''`). Number constants, however, need not be enclosed within single quotation marks.

All character searches are case-sensitive. In the following example, no rows are returned because the `EMPLOYEES` table stores all the last names in mixed case:

```
SELECT last_name, job_id, department_id
FROM   employees
WHERE  last_name = 'WHALEN';
```

Oracle databases store dates in an internal numeric format, representing the century, year, month, day, hours, minutes, and seconds. The default date display is in the `DD-MON-RR` format.

Note: For details about the RR format and about changing the default date format, see the lesson titled "Using Single-Row Functions to Customize Output." Also, you learn about the use of single-row functions such as `UPPER` and `LOWER` to override the case sensitivity in the same lesson.

Comparison Operators

Operator	Meaning
=	Equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
<>	Not equal to
`BETWEEN ...AND...`	Between two values (inclusive)
`IN(set)`	Match any of a list of values
`LIKE`	Match a character pattern
`IS NULL`	Is a null value

ORACLE

Comparison operators are used in conditions that compare one expression with another value or expression. They are used in the `WHERE` clause in the following format:

Syntax

... WHERE *expr operator* value

Example

```
... WHERE hire_date = '01-JAN-05'
... WHERE salary >= 6000
... WHERE last_name = 'Smith'
```

Remember, an alias cannot be used in the `WHERE` clause.

Note: The symbols `!=` and `^=` can also represent the *not equal* to condition.

Using Comparison Operators

```
SELECT last_name, salary
FROM   employees
WHERE  salary <= 3000 ;
```

	LAST_NAME	SALARY
1	Matos	2600
2	Vargas	2500

```
SELECT *
FROM   employees
WHERE  last_name = 'Ernst';
```

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID
1	104	Bruce	Ernst	BERNST	590.423.4568	21-MAY-15	IT_PROG	6000	(null)	103	60

ORACLE

In the first example in the slide, the SELECT statement retrieves the last name and salary from the EMPLOYEES table for any employee whose salary is less than or equal to $3,000. Note that there is an explicit value supplied to the WHERE clause. The explicit value of 3000 is compared to the salary value in the SALARY column of the EMPLOYEES table.

In the second code example, the SELECT statement retrieves all rows where the last name is Ernst. Because * is used in the SELECT statement, all fields from the EMPLOYEES table would appear in the result set.

Range Conditions Using the BETWEEN Operator

Use the BETWEEN operator to display rows based on a range of values.

```
SELECT last_name, salary
FROM   employees
WHERE  salary BETWEEN 2500 AND 3500 ;
```

Lower limit   Upper limit

	LAST_NAME	SALARY
1	Rajs	3500
2	Davies	3100
3	Matos	2600
4	Vargas	2500

ORACLE

You can display rows based on a range of values using the BETWEEN operator. The range that you specify contains a lower limit and an upper limit.

The SELECT statement in the slide returns rows from the EMPLOYEES table for any employee whose salary is between $2,500 and $3,500.

Values that are specified with the BETWEEN operator are inclusive. However, you must specify the lower limit first.

You can also use the BETWEEN operator on character values:

```
SELECT last_name FROM employees
WHERE last_name BETWEEN 'King' AND 'Whalen';
```

Using the IN Operator

Use the IN operator to test for values in a list.

```
SELECT employee_id, last_name, salary, manager_id
FROM   employees
WHERE  manager_id IN (100, 101, 201) ;
```

	EMPLOYEE_ID	LAST_NAME	SALARY	MANAGER_ID
1	101	Kochhar	17000	100
2	102	De Haan	17000	100
3	124	Mourgos	5800	100
4	149	Zlotkey	10500	100
5	201	Hartstein	13000	100
6	200	Whalen	4400	101
7	205	Higgins	12008	101
8	202	Fay	6000	201

ORACLE

To test for values in a specified set of values, use the `IN` operator. The condition defined using the `IN` operator is also known as the *membership condition*.

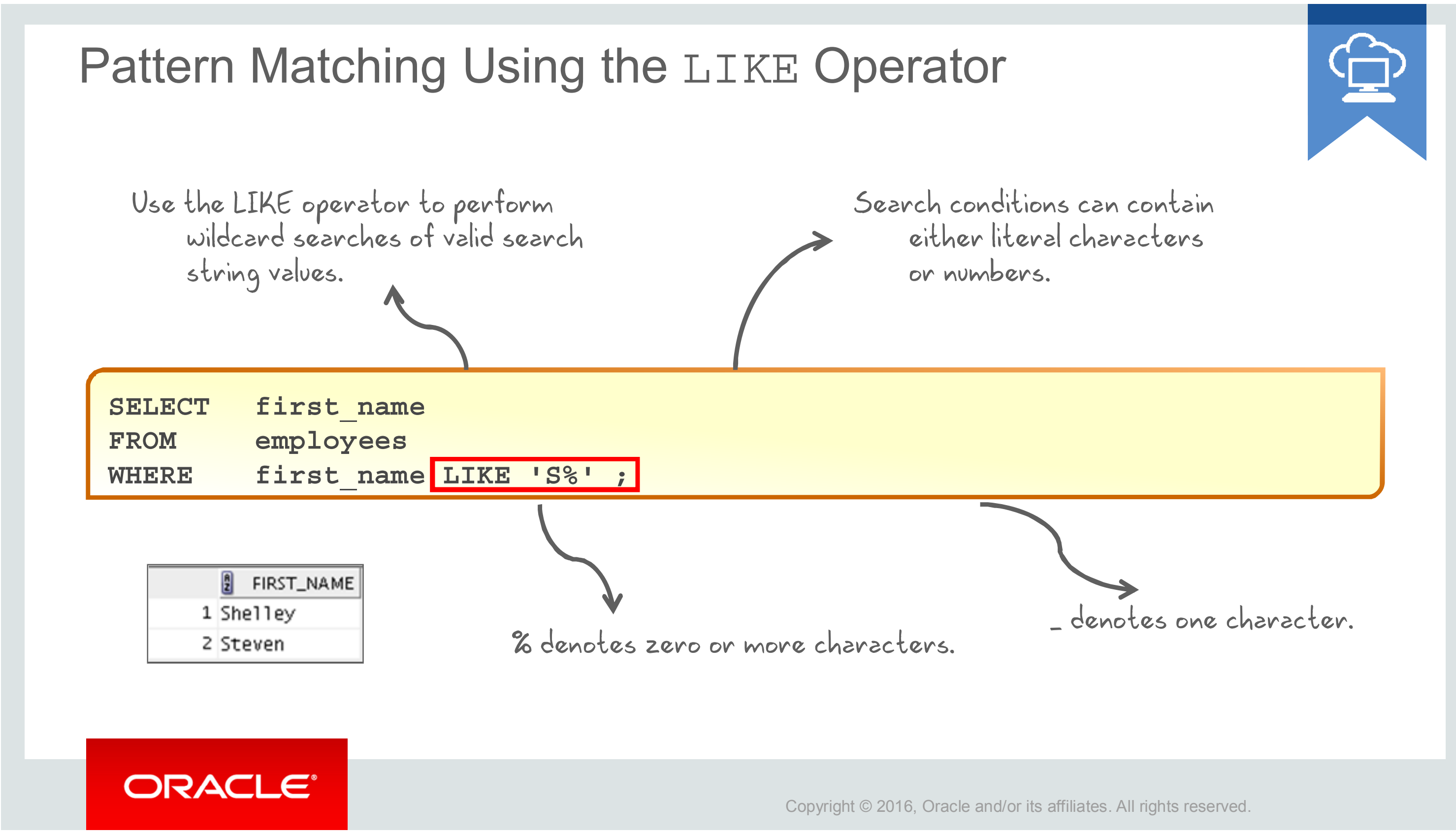
The example in the slide displays employee numbers, last names, salaries, and manager's employee numbers for all the employees whose manager's employee number is 100, 101, or 201.

Note: The set of values can be specified in any random order—for example, (201,100,101).

The `IN` operator can be used with any data type. The following example returns a row from the `EMPLOYEES` table, for any employee whose last name is included in the list of names in the `WHERE` clause:

```
SELECT  employee_id, manager_id, department_id
FROM    employees
WHERE   last_name IN ('Hartstein', 'Vargas');
```

If characters or dates are used in a list, they must be enclosed within single quotation marks (`''`).



You may not always know the exact value to search for. You can select rows that match a character pattern by using the `LIKE` operator. The character pattern–matching operation is referred to as a *wildcard* search. Two symbols can be used to construct the search string.

Symbol	Description
%	Represents any sequence of zero or more characters
_	Represents any single character

The `SELECT` statement in the slide returns the first name from the `EMPLOYEES` table for any employee whose first name begins with the letter "S." Note the uppercase "S." Consequently, names beginning with a lowercase "s" are not returned.

The `LIKE` operator can be used as a shortcut for some `BETWEEN` comparisons. The following example displays the last names and hire dates of all employees who joined between January, 2015 and December, 2015:

```
SELECT last_name, hire_date
FROM   employees
WHERE  hire_date LIKE '%15';
```

Combining Wildcard Characters

You can combine the two wildcard characters (% and _) with literal characters for pattern matching.

```
SELECT last_name
FROM   employees
WHERE  last_name LIKE '_o%' ;
```

	LAST_NAME
1	Kochhar
2	Lorentz
3	Mourgos

You can use the ESCAPE identifier to search for the actual % and _ symbols.

ORACLE

The % and _ symbols can be used in any combination with literal characters. The example in the slide displays the names of all employees whose last names have the letter “o” as the second character.

When you need to have an exact match for the actual % and _ characters, use the ESCAPE identifier.

Using NULL Conditions

Test for nulls with the IS NULL operator.

```
SELECT last_name, manager_id
FROM   employees
WHERE  manager_id IS NULL ;
```

	LAST_NAME	MANAGER_ID
1	King	(null)

The NULL conditions include the IS NULL condition and the IS NOT NULL condition.

The IS NULL condition tests for nulls. A null value means that the value is unavailable, unassigned, unknown, or inapplicable. Therefore, you cannot test with =, because a null cannot be equal or unequal to any value. The example in the slide retrieves the last_name and manager_id of all employees who do not have a manager.

Here is another example: To display the last name, job ID, and commission for all employees who are *not* entitled to receive a commission, use the following SQL statement:

```
SELECT last_name, job_id, commission_pct
FROM   employees
WHERE  commission_pct IS NULL;
```

...

Defining Conditions Using Logical Operators

Operator	Meaning
`AND`	Returns `TRUE` if *both* component conditions are true
`OR`	Returns `TRUE` if *either* component condition is true
`NOT`	Returns `TRUE` if the condition is false

A logical condition combines the results of two or more component conditions to produce a single result based on those conditions, or it inverts the result of a single condition. A row is returned only if the overall result of the condition is true.

Three logical operators are available in SQL:

- `AND`
- `OR`
- `NOT`

All the examples so far have specified only one condition in the `WHERE` clause. You can use several conditions in a single `WHERE` clause using the `AND` and `OR` operators.

Using the AND Operator

AND requires both the component conditions to be true.

```
SELECT employee_id, last_name, job_id, salary
FROM   employees
WHERE  salary >= 10000
AND    job_id LIKE '%MAN%' ;
```

	EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
1	149	Zlotkey	SA_MAN	10500
2	201	Hartstein	MK_MAN	13000

In the example, both the component conditions must be true for any record to be selected. Therefore, only those employees who have a job title that contains the string 'MAN' *and* earn $10,000 or more are selected.

All character searches are case-sensitive, that is, no rows are returned if 'MAN' is not uppercase. Further, character strings must be enclosed within quotation marks.

AND Truth Table

The following table shows the results of combining two expressions with AND:

AND	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

Using the OR Operator

OR requires at least one component condition to be true.

```
SELECT employee_id, last_name, job_id, salary
FROM   employees
WHERE  salary >= 10000
OR     job_id LIKE '%MAN%' ;
```

	EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
1	100	King	AD_PRES	24000
2	101	Kochhar	AD_VP	17000
3	102	De Haan	AD_VP	17000
4	124	Mourgos	ST_MAN	5800
5	149	Zlotkey	SA_MAN	10500
6	174	Abel	SA_REP	11000
7	201	Hartstein	MK_MAN	13000
8	205	Higgins	AC_MGR	12008

ORACLE

In the example, either component condition can be true for any record to be selected. Therefore, any employee who has a job ID that contains the string 'MAN' *or* earns $10,000 or both is selected.

OR Truth Table

The following table shows the results of combining two expressions with OR:

OR	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

Using the NOT Operator

```
SELECT last_name, job_id
FROM   employees
WHERE  job_id
NOT IN ('IT_PROG', 'ST_CLERK', 'SA_REP') ;
```

	LAST_NAME	JOB_ID
1	De Haan	AD_VP
2	Fay	MK_REP
3	Gietz	AC_ACCOUNT
4	Hartstein	MK_MAN
5	Higgins	AC_MGR
6	King	AD_PRES
7	Kochhar	AD_VP
8	Mourgos	ST_MAN
9	Whalen	AD_ASST
10	Zlotkey	SA_MAN

ORACLE

The example in the slide displays the last name and job ID of all employees whose job ID *is not* IT_PROG, ST_CLERK, or SA_REP.

NOT Truth Table

The following table shows the result of applying the NOT operator to a condition:

NOT	TRUE	FALSE	NULL
	FALSE	TRUE	NULL

Lesson Agenda

- Limiting rows with:
  - `WHERE` clause
  - Comparison conditions using =, <=, BETWEEN, IN, LIKE, and NULL operators
  - Logical conditions using AND, OR, and NOT operators
- Rules of precedence for operators in an expression
- Sorting rows using the ORDER BY clause
- SQL row limiting clause in a query
- Substitution variables
- DEFINE and VERIFY commands

ORACLE

Rules of Precedence

Operator	Meaning
1	Arithmetic operators
2	Concatenation operator
3	Comparison conditions
4	`IS [NOT] NULL`, `LIKE`, `[NOT] IN`
5	`[NOT] BETWEEN`
6	Not equal to
7	`NOT` logical operator
8	`AND` logical operator
9	`OR` logical operator

You can use parentheses to override rules of precedence.

ORACLE

The rules of precedence determine the order in which expressions are evaluated and calculated. The table in the slide lists the default order of precedence. However, you can override the default order by using parentheses around the expressions that you want to calculate first.

Rules of Precedence

```
SELECT last_name, department_id, salary
FROM   employees
WHERE  department_id = 60
OR     department_id = 80
AND    salary > 10000;
```

1

	LAST_NAME	DEPARTMENT_ID	SALARY
1	Hunold	60	9000
2	Ernst	60	6000
3	Lorentz	60	4200
4	Zlotkey	80	10500
5	Abel	80	11000

```
SELECT last_name, department_id, salary
FROM   employees
WHERE  (department_id = 60
OR     department_id = 80)
AND    salary > 10000;
```

2

	LAST_NAME	DEPARTMENT_ID	SALARY
1	Zlotkey	80	10500
2	Abel	80	11000

ORACLE

1. **Precedence of the `AND` Operator: Example**

   In this example, there are two conditions:

   - The first condition is that the department `ID` is `80` *and* the salary is greater than $10,000.
   - The second condition is that the department `ID` is `60`.

   Therefore, the `SELECT` statement reads as follows:

   "Select the row if an employee's department ID is 80 *and* earns more than $10,000, *or* if the employee's department ID is 60."

2. **Using Parentheses: Example**

   In this example, there are two conditions:

   - The first condition is that the department `ID` is 80 *or* 60.
   - The second condition is that the salary is greater than $10,000.

   Therefore, the `SELECT` statement reads as follows:

   "Select the row if an employee's department ID is 80 *or* 60, *and* if the employee earns more than $10,000."

Lesson Agenda

- Limiting rows with:
  – `WHERE` clause
  – Comparison conditions using `=`, `<=`, `BETWEEN`, `IN`, `LIKE`, and `NULL` operators
  – Logical conditions using `AND`, `OR`, and `NOT` operators
- Rules of precedence for operators in an expression
- Sorting rows using the `ORDER BY` clause
- SQL row limiting clause in a query
- Substitution variables
- `DEFINE` and `VERIFY` commands

ORACLE

Using the ORDER BY Clause

Sort the retrieved rows with the ORDER BY clause.

ASC: Ascending order, default

```
SELECT   last_name, job_id, department_id, hire_date
FROM     employees
ORDER BY hire_date ;
```

DESC: Descending order

	LAST_NAME	JOB_ID	DEPARTMENT_ID	HIRE_DATE
1	De Haan	AD_VP	90	13-JAN-09
2	Gietz	AC_ACCOUNT	110	07-JUN-10
3	Higgins	AC_MGR	110	07-JUN-10
4	King	AD_PRES	90	17-JUN-11
5	Whalen	AD_ASST	10	17-SEP-11
6	Rajs	ST_CLERK	50	17-OCT-11
7	Hartstein	MK_MAN	20	17-FEB-12

...

ORACLE

The order of rows that are returned in a query result is undefined. The ORDER BY clause can be used to sort the rows. You can specify an expression, an alias, or a column position as the sort condition. You can specify multiple expressions in the *order_by_clause*. Oracle Database first sorts rows based on their values for the first expression. Rows with the same value for the first expression are then sorted based on their values for the second expression, and so on.

Syntax

```
SELECT          expr
FROM            table
[WHERE          condition(s)]
[ORDER BY  {column, expr, numeric_position} [ASC|DESC]];
```

In the syntax:

ORDER BY	Specifies the order in which the retrieved rows are displayed
ASC	Orders the rows in ascending order (this is the default order)
DESC	Orders the rows in descending order

If the ORDER BY clause is not used, the sort order is undefined, and the Oracle server may not fetch rows in the same order for the same query twice. Use the ORDER BY clause to display the rows in a specific order.

Sorting

Sorting in descending order

```
SELECT   last_name, job_id, department_id, hire_date
FROM     employees
ORDER BY department_id DESC ;
```

1

Sorting by column alias

```
SELECT employee_id, last_name, salary*12 annsal
FROM   employees
ORDER BY annsal ;
```

2

ORACLE

The default sort order is ascending:

- Numeric values are displayed with the lowest values first (for example, 1 to 999).
- Date values are displayed with the earliest value first (for example, 01-JAN-12 before 01-JAN-16).
- Character values are displayed in the alphabetical order (for example, “A” first and “Z” last).
- Null values are displayed last for ascending sequences and first for descending sequences.
- You can also sort by a column that is not in the `SELECT` list.

Examples

1. To reverse the order in which the rows are displayed, specify the `DESC` keyword after the column name in the `ORDER BY` clause. The example in the slide sorts the result by the department_id.
2. You can also use a column alias in the `ORDER BY` clause. The slide example sorts the data by annual salary.

Note: Use the keywords `NULLS FIRST` or `NULLS LAST` to specify whether returned rows containing null values should appear first or last in the ordering sequence.

Sorting

Sorting by using the column's numeric position

```
SELECT   last_name, job_id, department_id, hire_date
FROM     employees
ORDER BY 3;
```

3

Sorting by multiple columns

```
SELECT last_name, department_id, salary
FROM   employees
ORDER BY department_id, salary DESC;
```

4

ORACLE

Examples

3. You can sort query results by specifying the numeric position of the column in the `SELECT` clause. The example in the slide sorts the result by the `department_id` as this column is at the third position in the `SELECT` clause.
4. You can sort query results by more than one column. You list the columns (or `SELECT` list column sequence numbers) in the `ORDER BY` clause, delimited by commas. The results are ordered by the first column, then the second, and so on for as many columns as the `ORDER BY` clause includes. If you want any results sorted in descending order, your `ORDER BY` clause must use the `DESC` keyword directly after the name or the number of the relevant column. The result of the query example shown in the slide is sorted by `department_id` in ascending order and also by salary in descending order.

Lesson Agenda

- Limiting rows with:
  - `WHERE` clause
  - Comparison conditions using `=`, `<=`, `BETWEEN`, `IN`, `LIKE`, and `NULL` operators
  - Logical conditions using `AND`, `OR`, and `NOT` operators
- Rules of precedence for operators in an expression
- Sorting rows using the `ORDER BY` clause
- SQL row limiting clause in a query
- Substitution variables
- `DEFINE` and `VERIFY` commands

Using SQL Row Limiting Clause in a Query

The row_limiting_clause limits the number of rows that are returned in the result set.

```
SELECT …
   FROM …
 [ WHERE …  ]
 [ ORDER BY …  ]
 [OFFSET offset { ROW | ROWS }]
[FETCH { FIRST | NEXT } [{ row_count | percent PERCENT }] { ROW | ROWS }
  { ONLY | WITH TIES }]
```


You specify the `row_limiting_clause` in the SQL `SELECT` statement by placing it after the `ORDER BY` clause. Note that an `ORDER BY` clause is required if you want to sort the rows for consistency.

- `OFFSET`: Use this clause to specify the number of rows to skip before row limiting begins. The value for offset must be a number. If you specify a negative number, offset is treated as 0. If you specify `NULL` or a number greater than or equal to the number of rows that are returned by the query, 0 rows are returned.
- `ROW` | `ROWS`: Use these keywords interchangeably. They are provided for semantic clarity.
- `FETCH`: Use this clause to specify the number of rows or percentage of rows to return.
- `FIRST` | `NEXT`: Use these keywords interchangeably. They are provided for semantic clarity.
- `row_count` | *percent* `PERCENT`: Use `row_count` to specify the number of rows to return. Use *percent* `PERCENT` to specify the percentage of the total number of selected rows to return. The value for percent must be a number.
- `ONLY` | `WITH TIES`: Specify `ONLY` to return exactly the specified number of rows or percentage of rows. Specify `WITH TIES` to return additional rows with the same sort key as the last row fetched. If you specify `WITH TIES`, then you must specify the `order_by_clause`. If you do not specify the `order_by_clause`, then no additional rows will be returned.

SQL Row Limiting Clause: Example

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

Script Output × | Query Result ×

SQL | All Rows Fetched: 5

	EMPLOYEE_ID	FIRST_NAME
1	100	Steven
2	101	Neena
3	102	Lex
4	103	Alexander
5	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
1	107	Diana
2	124	Kevin
3	141	Trenna
4	142	Curtis
5	143	Randall

ORACLE

- The first code example returns the five employees with the lowest `employee_id`.
- The second code example returns the five employees with the next set of lowest `employee_id`.

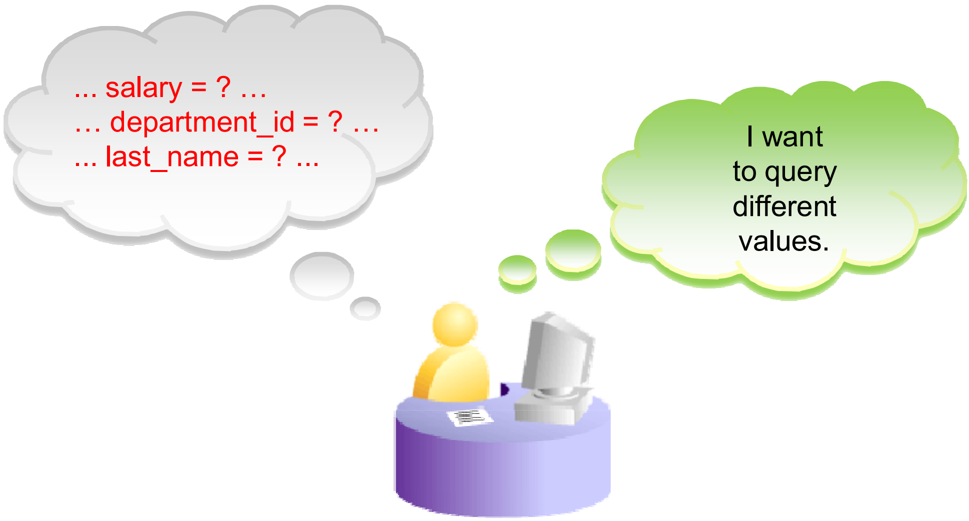
Note: If `employee_id` is assigned sequentially by the date when the employee joined the organization, these examples give us the top 5 employees and then employees 6-10, all in terms of seniority.

Lesson Agenda

- Limiting rows with:
  - `WHERE` clause
  - Comparison conditions using =, <=, `BETWEEN`, `IN`, `LIKE`, and `NULL` operators
  - Logical conditions using `AND`, `OR`, and `NOT` operators
- Rules of precedence for operators in an expression
- Sorting rows using the `ORDER BY` clause
- SQL row limiting clause in a query
- Substitution variables
- `DEFINE` and `VERIFY` commands

ORACLE

Substitution Variables



So far, all the SQL statements were executed with predetermined columns, conditions, and their values. Suppose that you want a query that lists the employees with various jobs and not just those whose `job_ID` is `SA_REP`. You can edit the `WHERE` clause to provide a different value each time you run the command, but there is also an easier way.

By using a substitution variable in place of the exact values in the `WHERE` clause, you can run the same query for different values.

You can create reports that prompt users to supply their own values to restrict the range of data returned, by using substitution variables. You can embed *substitution variables* in a command file or in a single SQL statement. A variable can be thought of as a container in which values are temporarily stored. When the statement is run, the stored value is substituted.

Substitution Variables

- Use substitution variables to temporarily store values with:
  - Single-ampersand (`&`) substitution
  - Double-ampersand (`&&`) substitution
- Use substitution variables to supplement the following:
  - `WHERE` conditions
  - `ORDER BY` clauses
  - Column expressions
  - Table names
  - Entire `SELECT` statements

ORACLE

You can use single-ampersand (`&`) substitution variables to temporarily store values.

You can also predefine variables by using the `DEFINE` command. `DEFINE` creates and assigns a value to a variable.

Restricted Ranges of Data: Examples

- Reporting figures only for the current quarter or the specified date range
- Reporting on data relevant only to the user requesting the report
- Displaying personnel only within a given department

Other Interactive Effects

Interactive effects are not restricted to direct user interaction with the `WHERE` clause. The same principles can also be used to achieve other goals, such as:

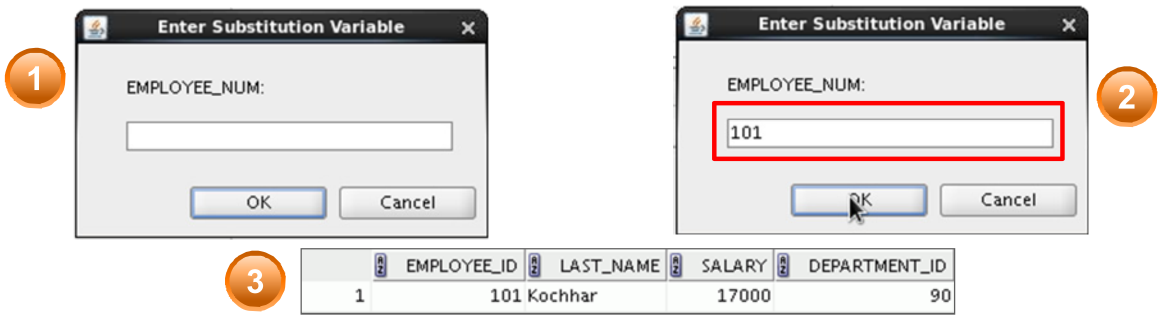
- Obtaining input values from a file rather than from a person
- Passing values from one SQL statement to another

Note: SQL Developer supports substitution variables and the `DEFINE/UNDEFINE` commands.

Using the Single-Ampersand Substitution Variable

Use a variable prefixed with an ampersand (&) to prompt the user for a value:

```
SELECT employee_id, last_name, salary, department_id
FROM   employees
WHERE  employee_id = &employee_num ;
```



ORACLE

When running a report, users often want to restrict the data that is returned dynamically. SQL Developer provides this flexibility with user variables. Use an ampersand (&) to identify each variable in your SQL statement. However, you do not need to define the value of each variable.

Notation	Description
&user_variable	Indicates a variable in a SQL statement; if the variable does not exist, SQL Developer prompts the user for a value. (The new variable is discarded after it is used.)

1. The example in the slide creates a SQL Developer substitution variable for an employee number. When the statement is executed, SQL Developer prompts the user for an employee number.

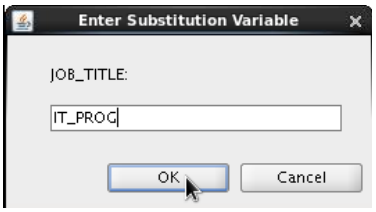
Note: With the single ampersand, the user is prompted every time the command is executed if the variable does not exist.

2. You enter a value and click the OK button.
3. The employee number, last name, salary, and department number for that employee is displayed in the result.

Character and Date Values with Substitution Variables

Use single quotation marks for date and character values:

```
SELECT last_name, department_id, salary*12
FROM   employees
WHERE  job_id = '&job_title' ;
```



	LAST_NAME	DEPARTMENT_ID	SALARY*12
1	Hunold	60	108000
2	Ernst	60	72000
3	Lorentz	60	50400

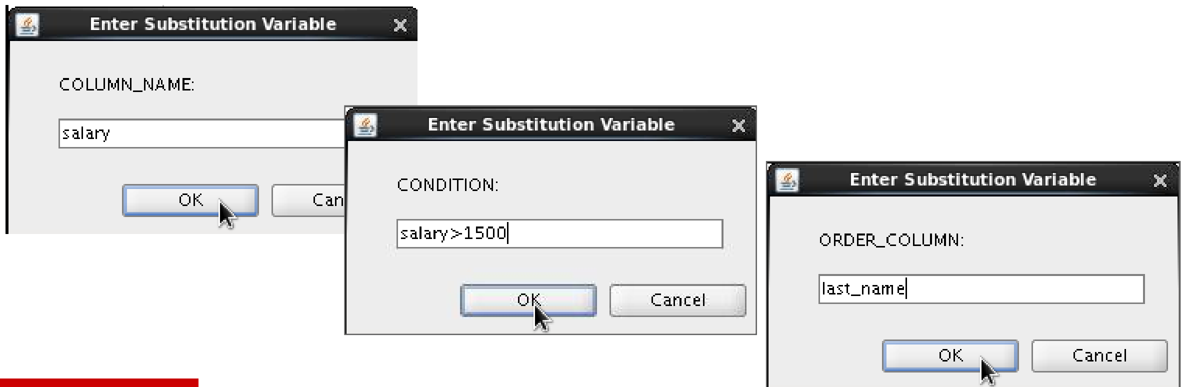
In a `WHERE` clause, date and character values must be enclosed within single quotation marks. The same rule applies to the substitution variables.

Enclose the variables with single quotation marks within the SQL statement itself.

The slide shows a query to retrieve the employee names, department numbers, and annual salaries of all employees based on the job title value of the SQL Developer substitution variable.

Specifying Column Names, Expressions, and Text

```
SELECT employee_id, last_name, job_id,&column_name
FROM   employees
WHERE  &condition
ORDER BY &order_column ;
```



You can use the substitution variables not only in the `WHERE` clause of a SQL statement, but also as substitution for column names, expressions, or text.

Example

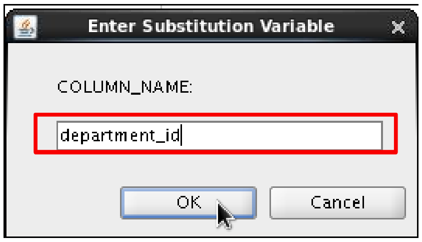
The example in the slide displays the employee number, last name, job title, and any other column that is specified by the user at run time, from the `EMPLOYEES` table. For each substitution variable in the `SELECT` statement, you are prompted to enter a value, and then click OK to proceed.

If you do not enter a value for the substitution variable, you get an error when you execute the statement in the slide.

Note: A substitution variable can be used anywhere in the `SELECT` statement, except as the first word entered at the command prompt.

Using the Double-Ampersand Substitution Variable

```
SELECT    employee_id, last_name, job_id, &&column_name
FROM      employees
ORDER BY &column_name ;
```



Use double ampersand (&&) if you want to reuse the variable value without prompting the user each time.

	EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
1	200	Whalen	AD_ASST	10
2	201	Hartstein	MK_MAN	20
3	202	Fay	MK_REP	20

You can use the double-ampersand (`&&`) substitution variable if you want to reuse the variable value without prompting the user each time. The user sees the prompt for the value only once. In the example in the slide, the user is asked to give the value for the `column_name` variable only once. The value that is supplied by the user (`department_id`) is used for both display and ordering of data. If you run the query again, you will not be prompted for the value of the variable.

SQL Developer stores the value that is supplied by using the `DEFINE` command; it uses it again whenever you reference the variable name. After a user variable is in place, you need to use the `UNDEFINE` command to delete it:

```
UNDEFINE column_name;
```

Lesson Agenda

- Limiting rows with:
  - `WHERE` clause
  - Comparison conditions using `=`, `<=`, `BETWEEN`, `IN`, `LIKE`, and `NULL` operators
  - Logical conditions using `AND`, `OR`, and `NOT` operators
- SQL row limiting clause in a query
- Rules of precedence for operators in an expression
- Sorting rows using the `ORDER BY` clause
- Substitution variables
- `DEFINE` and `VERIFY` commands

ORACLE

Using the DEFINE Command

Use the DEFINE command to create and assign a value to a variable.

Use the UNDEFINE command to remove a variable.

```
DEFINE employee_num = 200

SELECT employee_id, last_name, salary, department_id
FROM   employees
WHERE  employee_id = &employee_num ;

UNDEFINE employee_num
```

	EMPLOYEE_ID	LAST_NAME	SALARY	DEPARTMENT_ID
1	200	Whalen	4400	10

The example shown creates a substitution variable for an employee number by using the DEFINE command. At run time, the above query displays the employee number, name, salary, and department number for the employee whose ID is 200.

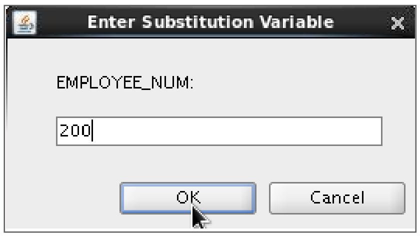
Because the variable is created using the SQL Developer DEFINE command, the user is not prompted to enter a value for the employee number. Instead, the defined variable value is automatically substituted in the SELECT statement.

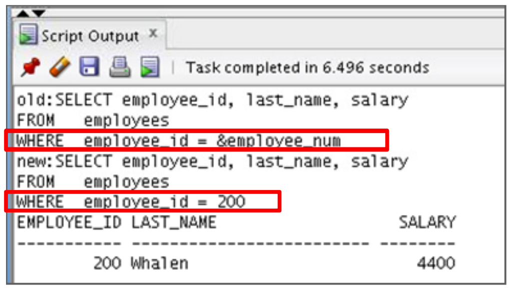
The EMPLOYEE_NUM substitution variable is present in the session until the user undefines it or exits the SQL Developer session.

Using the VERIFY Command

Use the VERIFY command to toggle the display of the substitution variable, both before and after SQL Developer replaces substitution variables with values.

```
SET VERIFY ON
SELECT employee_id, last_name, salary
FROM   employees
WHERE  employee_id = &employee_num;
```





To confirm the changes in the SQL statement, use the `VERIFY` command. Setting `SET VERIFY ON` forces SQL Developer to display the text of a command after it replaces substitution variables with values. To see the `VERIFY` output, you should use the Run Script (F5) icon in the SQL Worksheet. SQL Developer displays the text of a command after it replaces substitution variables with values, on the Script Output tab as shown in the slide.

The example in the slide displays the new value of the `EMPLOYEE_ID` column in the SQL statement followed by the output.

Quiz

Which four of the following are valid operators for the `WHERE` clause?

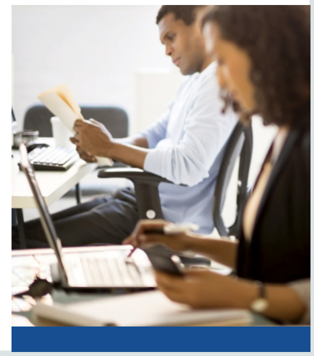
a. `>=`
b. `IS NULL`
c. `!=`
d. `IS LIKE`
e. `IN BETWEEN`
f. `<>`

Answer: a, b, c, f

Summary

In this lesson, you should have learned how to:

- Limit the rows that are retrieved by a query
- Sort the rows that are retrieved by a query
- Use ampersand substitution to restrict and sort output at run time



ORACLE

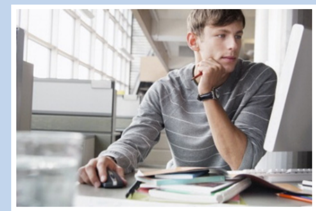
In this lesson, you should have learned about restricting and sorting rows that are returned by the `SELECT` statement. You should also have learned how to implement various operators and conditions.

By using the substitution variables, you can add flexibility to your SQL statements. This enables the queries to prompt for the filter condition for the rows during run time.

Practice 6: Overview

This practice covers the following topics:

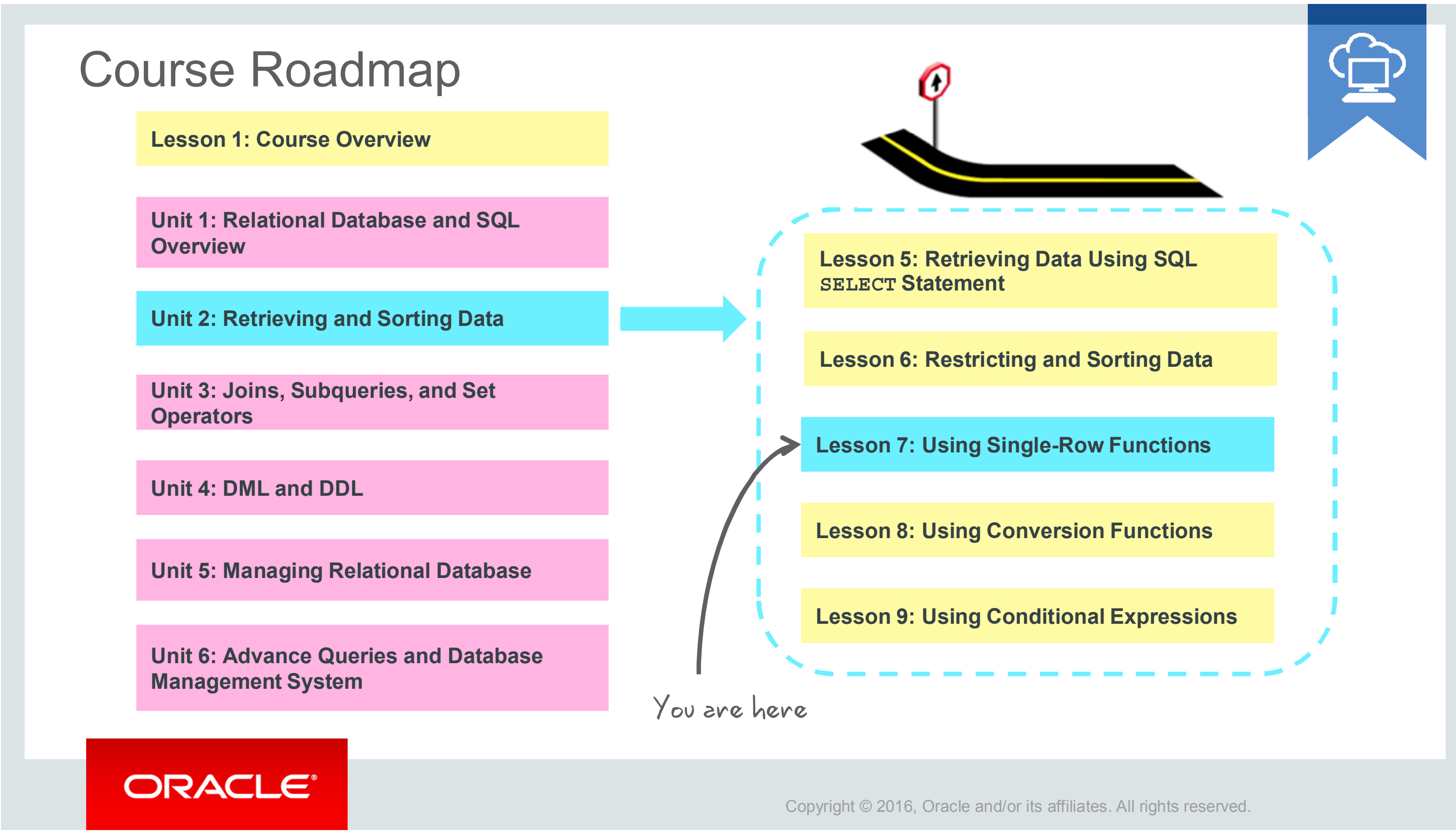
- Selecting data and changing the order of the rows that are displayed
- Restricting rows by using the `WHERE` clause
- Sorting rows by using the `ORDER BY` clause
- Using substitution variables to add flexibility to your SQL `SELECT` statements



ORACLE

In this practice, you build more reports, including statements that use the `WHERE` clause and the `ORDER BY` clause. You make the SQL statements more reusable and generic by including the ampersand substitution.


Lesson 7: Using Single-Row Functions to Customize Output
ORACLE
Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

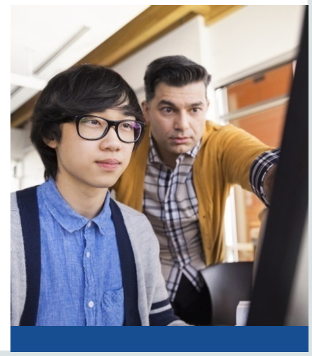


In Unit 2, you will dive into the concepts of SQL. You will learn to use the SQL SELECT statement to retrieve data from database tables and restrict and sort the retrieved data. You will also learn about single-row functions, conversion functions, and conditional expressions in SQL.

Objectives

After completing this lesson, you should be able to:

- Describe the various types of functions available in SQL
- Use the character, number, and date functions in `SELECT` statements



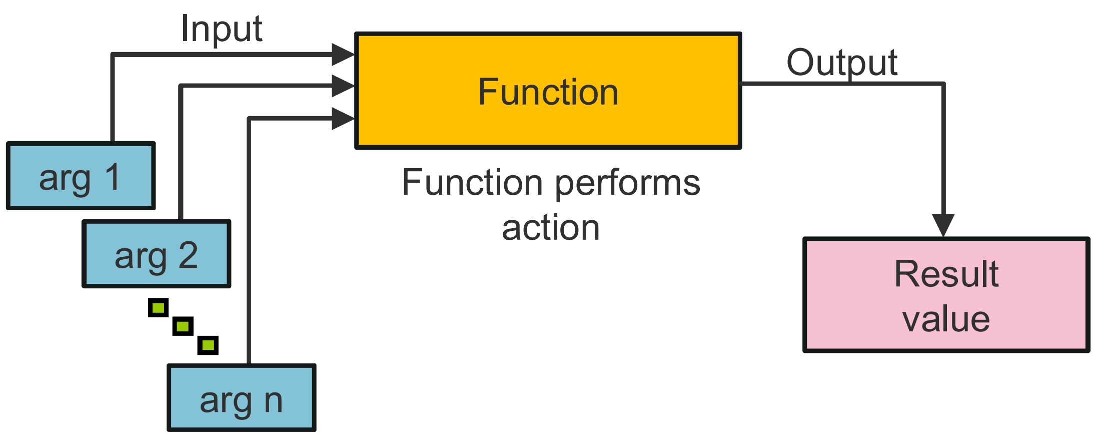
Functions make the basic query block more powerful, and they are used to manipulate data values. This is the first of two lessons that explore functions. It focuses on single-row character, number, and date functions.

Lesson Agenda

- Single-row SQL functions
- Character functions
- Nesting functions
- Number functions
- Working with dates
- Date functions

ORACLE

SQL Functions

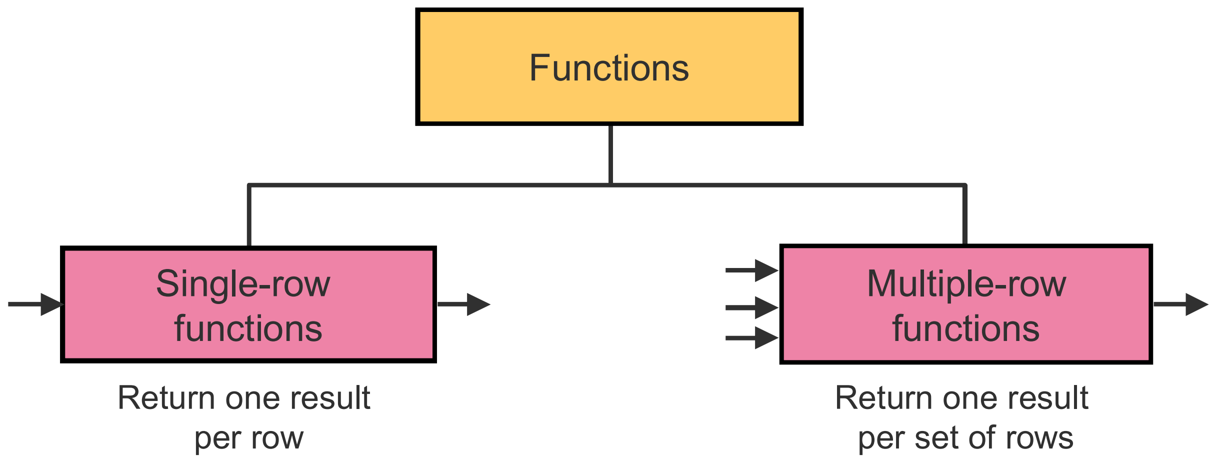


Functions are a very powerful feature of SQL. They can be used to do the following:

- Perform calculations on data.
- Modify individual data items.
- Manipulate output for groups of rows.
- Format dates and numbers for display.
- Convert column data types.

SQL functions sometimes take arguments and always return a value.

Two Types of SQL Functions



There are two types of functions:

- Single-row functions
- Multiple-row functions

Single-Row Functions

These functions operate on single rows only and return one result per row. There are different types of single-row functions. This lesson covers the following functions:

- Character
- Number
- Date

Multiple-Row Functions

Functions can manipulate groups of rows to give one result per group of rows. These functions are also known as *group functions* (covered in the lesson titled “Reporting Aggregated Data Using the Group Functions”).

Note: For more information and a complete list of available functions and their syntax, see the “Functions” section in *Oracle Database SQL Language Reference* for 12*c* database.

Single-Row Functions

Single-row functions:
- Manipulate data items
- Accept arguments and return one value
- Act on each row that is returned
- Return one result per row
- May modify the data type
- Can be nested
- Accept arguments that can be a column or an expression

```
function_name [(arg1, arg2,...)]
```

ORACLE

Single-row functions are used to manipulate data items. They accept one or more arguments and return one value for each row that is returned by the query. An argument can be one of the following:

- User-supplied constant
- Variable value
- Column name
- Expression

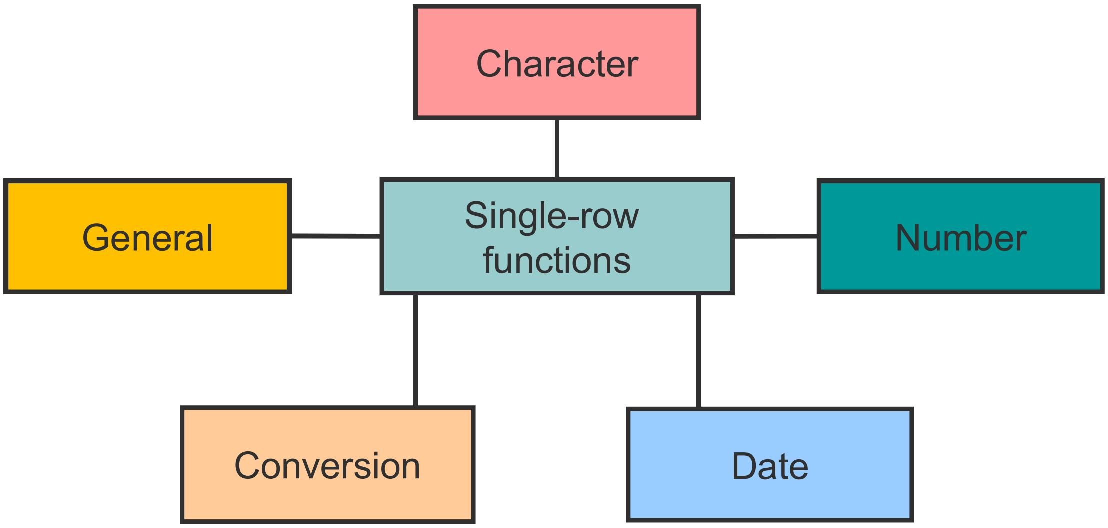
Features of single-row functions include:

- Act on each row that is returned in the query
- Return one result per row
- Possibly return a data value of a different type than the one that is referenced
- Possibly expect one or more arguments
- Can be used in `SELECT`, `WHERE`, and `ORDER BY` clauses; can be nested

In the syntax:

function_name	Is the name of the function
arg1, arg2	Is any argument to be used by the function. This can be represented by a column name or expression.

Single-Row Functions



This lesson covers the following single-row functions:

- **Character functions:** Accept character input and can return both character and number values
- **Number functions:** Accept numeric input and return numeric values
- **Date functions:** Operate on values of the `DATE` data type

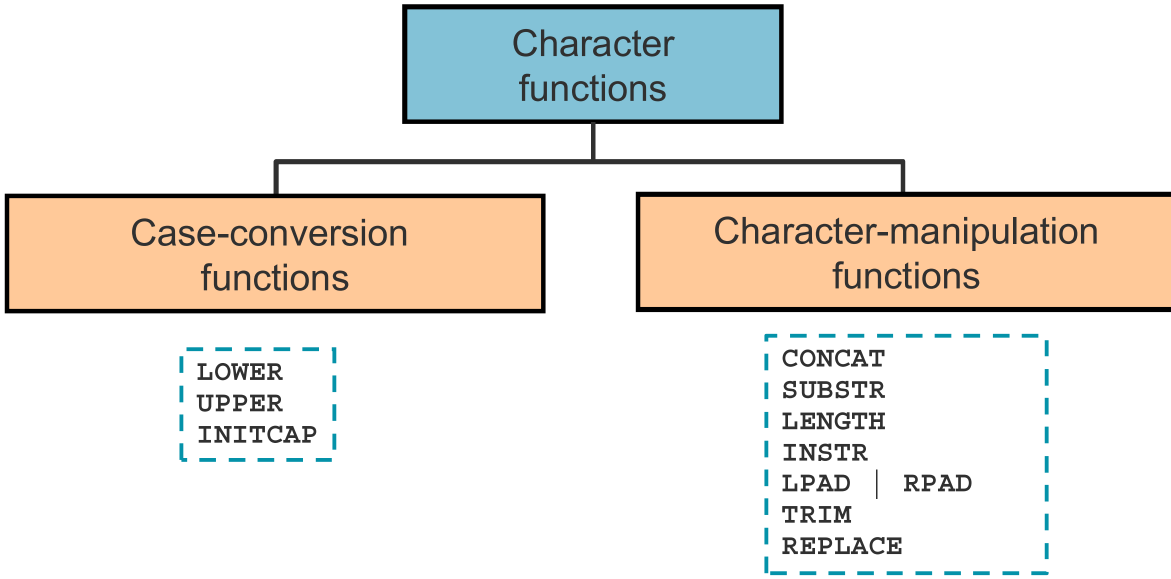
The following single-row functions are discussed in the lesson titled "Using Conversion Functions and Conditional Expressions":

- **Conversion functions:** Convert a value from one data type to another
- **General functions:** These functions take any data type and can also handle NULLs.

Lesson Agenda

- Single-row SQL functions
- Character functions
- Nesting functions
- Number functions
- Working with dates
- Date functions

Character Functions



Single-row character functions accept character data as input and can return both character and numeric values. Character functions can be divided into the following:

- Case-conversion functions
- Character-manipulation functions

Function	Purpose
`LOWER(`*`column`*`|`*`expression`*`)`	Converts alpha character values to lowercase
`UPPER(`*`column`*`|`*`expression`*`)`	Converts alpha character values to uppercase
`INITCAP(`*`column`*`|`*`expression`*`)`	Converts alpha character values to uppercase for the first letter of each word; all other letters in lowercase
`CONCAT(`*`column1`*`|`*`expression1`*`, `*`column2`*`|`*`expression2`*`)`	Concatenates the first character value to the second character value; equivalent to concatenation operator (||)
`SUBSTR(`*`column`*`|`*`expression`*`,`*`m`*`[,`*`n`*`])`	Returns specified characters from character value starting at character position *m*, *n* characters long (If *m* is negative, the count starts from the end of the character value. If *n* is omitted, all characters to the end of the string are returned.)

Note: The functions discussed in this lesson are only some of the available functions. You can learn about a few more functions in the next page.

Function	Purpose
LENGTH(*column*|*expression*)	Returns the number of characters in the expression
INSTR(*column*|*expression*, *'string'*, *[,m]*, *[n]*)	Returns the numeric position of a named string. Optionally, you can provide a position *m* to start searching, and the occurrence *n* of the string. *m* and *n* default to 1, meaning start the search at the beginning of the string and report the first occurrence.
LPAD(*column*|*expression*, *n*, '*string*') RPAD(*column*|*expression*, *n*, '*string*')	Returns an expression left-padded to length of *n* characters with the specified characters. Returns an expression right-padded to length of *n* characters with the specified characters.
TRIM(*leading*|*trailing*|*both*, *trim_character FROM trim_source*)	Enables you to trim leading or trailing characters (or both) from a character string. If *trim_character* or *trim_source* is a character literal, you must enclose it in single quotation marks.
REPLACE(*text*, *search_string*, *replacement_string*)	Searches a text expression for a character string and, if found, replaces it with a specified replacement string

Case-Conversion Functions

These functions convert the case for character strings:

Function	Result
`LOWER('SQL Course')`	`sql course`
`UPPER('SQL Course')`	`SQL COURSE`
`INITCAP('SQL Course')`	`Sql Course`

`LOWER`, `UPPER`, and `INITCAP` are the three case-conversion functions.

- `LOWER`: Converts mixed-case or uppercase character strings to lowercase
- `UPPER`: Converts mixed-case or lowercase character strings to uppercase
- `INITCAP`: Converts the first letter of each word to uppercase and the remaining letters to lowercase

Example

```
SELECT 'The job id for '||UPPER(last_name)||' is '
||LOWER(job_id) AS "EMPLOYEE DETAILS"
FROM   employees;
```

Using Case-Conversion Functions

Display the employee number, name, and department number for employee Higgins:

```
SELECT employee_id, last_name, department_id
FROM   employees
WHERE  last_name = 'higgins';
```

```
0 rows selected
```

```
SELECT employee_id, last_name, department_id
FROM   employees
WHERE  LOWER(last_name) = 'higgins';
```

	EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
1	205	Higgins	110

ORACLE

The slide example displays the employee number, name, and department number of employee Higgins.

The `WHERE` clause of the first SQL statement specifies the employee name as `higgins`. Because all the data in the `EMPLOYEES` table is stored in proper case, the name `higgins` does not find a match in the table, and no rows are selected.

The `WHERE` clause of the second SQL statement specifies that the employee name in the `EMPLOYEES` table is compared to `higgins`, after converting the `LAST_NAME` column to lowercase for comparison purposes. Because both names are now lowercase, a match is found and one row is selected. The `WHERE` clause can be rewritten in the following manner to produce the same result:

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

The name in the output appears as it was stored in the database. To display the name in uppercase, use the `UPPER` function in the `SELECT` statement.

```
SELECT employee_id, UPPER(last_name), department_id
FROM   employees
WHERE  INITCAP(last_name) = 'Higgins'
```

Character-Manipulation Functions

These functions manipulate character strings:

Function	Result
`CONCAT('Hello', 'World')`	`HelloWorld`
`SUBSTR('HelloWorld',1,5)`	`Hello`
`LENGTH('HelloWorld')`	`10`
`INSTR('HelloWorld', 'W')`	`6`
`LPAD(salary,10,'*')`	`*****24000`
`RPAD(salary,10,'*')`	`24000*****`

`CONCAT`, `SUBSTR`, `LENGTH`, `INSTR`, `LPAD`, and `RPAD` are the character-manipulation functions that are covered in this lesson.

- `CONCAT`: Joins values together (you are limited to using two parameters with `CONCAT`)
- `SUBSTR`: Extracts a string of determined length
- `LENGTH`: Shows the length of a string as a numeric value
- `INSTR`: Finds the numeric position of a named character
- `LPAD`: Returns an expression left-padded to the length of *n* characters with the specified characters
- `RPAD`: Returns an expression right-padded to the length of *n* characters with the specified characters

Note: You can use functions such as `UPPER` and `LOWER` with ampersand substitution. For example, use `UPPER('&job_title')` so that the user does not have to enter the job title in a specific case.

Using Character-Manipulation Functions

```
SELECT CONCAT(CONCAT(last_name, '''s job category is '), job_id)
"Job" FROM   employees
WHERE  SUBSTR(job_id, 4) = 'REP';
```

1

	Job
1	Abel's job category is SA_REP
2	Fay's job category is MK_REP
3	Grant's job category is SA_REP
4	Taylor's job category is SA_REP

```
SELECT employee_id, CONCAT(first_name, last_name) NAME,
LENGTH (last_name), INSTR(last_name, 'a') "Contains 'a'?"
FROM   employees
WHERE  SUBSTR(last_name, -1, 1) = 'n';
```

2

	EMPLOYEE_ID	NAME	LENGTH(LAST_NAME)	Contains 'a'?
1	102	LexDe Haan	7	5
2	200	JenniferWhalen	6	3
3	201	MichaelHartstein	9	2

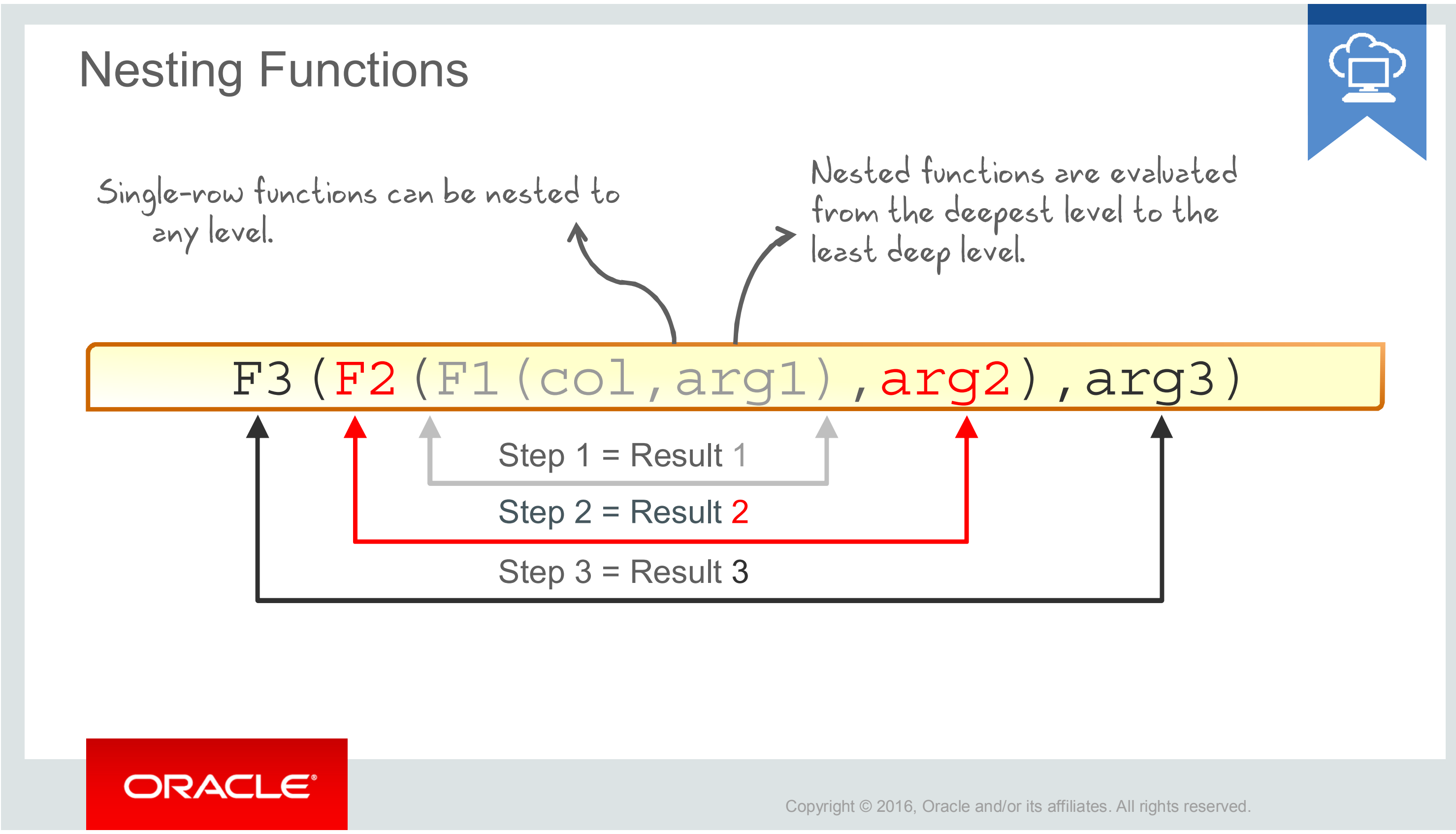
ORACLE

Examples

1. The first example in the slide displays employee last names and job IDs, joined together for all employees who have the string `REP` contained in the job `ID` starting at the fourth position.
2. The second SQL statement in the slide displays the concatenation of first name and last name, length of the last name, and the position for the first occurrence of the letter ‘a’ in the last name, if any, for those employees whose last names end with the letter “n.”

Lesson Agenda

- Single-row SQL functions
- Character functions
- Nesting functions
- Number functions
- Working with dates
- Date functions



Single-row functions can be nested to any depth. Nested functions are evaluated from the innermost level to the outermost level. Some examples follow to show you the flexibility of these functions.

Nesting Functions: Example

```
SELECT last_name,
  UPPER(CONCAT(SUBSTR (LAST_NAME, 1, 8), '_US'))
FROM   employees
WHERE  department_id = 60;
```

	LAST_NAME	UPPER(CONCAT(SUBSTR(LAST_NAME,1,8),'_US'))
1	Hunold	HUNOLD_US
2	Ernst	ERNST_US
3	Lorentz	LORENTZ_US

The example in the slide displays the last names of employees in department 60. The evaluation of the SQL statement involves three steps:

1. The inner function retrieves the first eight characters of the last name.
   `Result1 = SUBSTR (LAST_NAME, 1, 8)`
2. The outer function concatenates the result with `_US`.
   `Result2 = CONCAT(Result1, '_US')`
3. The outermost function converts the results to uppercase.

The entire expression becomes the column heading because no column alias was given.

Lesson Agenda

- Single-row SQL functions
- Character functions
- Nesting functions
- Number functions
- Working with dates
- Date Functions

Numeric Functions

- ROUND: Rounds value to a specified decimal
- TRUNC: Truncates value to a specified decimal
- CEIL: Returns the smallest whole number greater than or equal to a specified number
- FLOOR: Returns the largest whole number equal to or less than a specified number
- MOD: Returns remainder of division

Function	Result
`ROUND(45.926, 2)`	45.93
`TRUNC(45.926, 2)`	45.92
`CEIL (2.83)`	3
`FLOOR (2.83)`	2
`MOD (1600, 300)`	100

ORACLE

Numeric functions accept numeric input and return numeric values. This section describes some of the numeric functions.

Function	Purpose
ROUND(*column*|*expression*, *n*)	Rounds the column, expression, or value to *n* decimal places or, if *n* is omitted, no decimal places (If *n* is negative, numbers to the left of decimal point are rounded.)
TRUNC(*column*|*expression*, *n*)	Truncates the column, expression, or value to *n* decimal places or, if *n* is omitted, no decimal places
MOD(m,n)	Returns the remainder of *m* divided by *n*

Note: This list contains only some of the available numeric functions.

For more information, see the "Numeric Functions" section in *Oracle Database SQL Language Reference* for 12*c* database.

Using the ROUND Function

```
SELECT ROUND(45.923,2), ROUND(45.923,0),
       ROUND(45.923,-1)
FROM   DUAL;
```

	ROUND(45.923,2)	ROUND(45.923,0)	ROUND(45.923,-1)
1	45.92	46	50

The ROUND function rounds the column, expression, or value to *n* decimal places. If the second argument is 0 or is missing, the value is rounded to zero decimal places. If the second argument is 2, the value is rounded to two decimal places. Conversely, if the second argument is –2, the value is rounded to two decimal places to the left (rounded to the nearest unit of 100).

DUAL Table

The DUAL table is owned by the user SYS and can be accessed by all users. It contains one column, DUMMY, and one row with the value X. The DUAL table is useful when you want to return a value only once (for example, the value of a constant, pseudocolumn, or expression that is not derived from a table with user data). The DUAL table is generally used for completeness of the SELECT clause syntax, because both SELECT and FROM clauses are mandatory, and several calculations do not need to select from the actual tables.

Using the TRUNC Function

```
SELECT  TRUNC(45.923,2), TRUNC(45.923),
        TRUNC(45.923,-1)
FROM    DUAL;
```

	TRUNC(45.923,2)	TRUNC(45.923)	TRUNC(45.923,-1)
1	45.92	45	40

ORACLE

The TRUNC function truncates the column, expression, or value to *n* decimal places.

The TRUNC function works with arguments similar to those of the ROUND function. If the second argument is 0 or is missing, the value is truncated to zero decimal places. If the second argument is 2, the value is truncated to two decimal places. Conversely, if the second argument is –2, the value is truncated to two decimal places to the left. If the second argument is –1, the value is truncated to one decimal place to the left.

Using the MOD Function

Display the employee records where the employee_id is an even number.

```
SELECT employee_id as "Even Numbers", last_name
FROM employees
WHERE MOD(employee_id,2) = 0;
```

	Even Numbers	LAST_NAME
1	174	Abel
2	142	Davies
3	102	De Haan
4	104	Ernst
5	202	Fay
6	206	Gietz
7	178	Grant
8	100	King
9	124	Mourgos
10	176	Taylor
11	144	Vargas
12	200	Whalen

The MOD function finds the remainder of the first argument divided by the second argument. The slide example displays employee records where the employee_id is an even number.

Note: The MOD function is often used to determine whether a value is odd or even.

Lesson Agenda

- Single-row SQL functions
- Character functions
- Nesting functions
- Number functions
- Working with dates
- Date functions

Working with Dates

- The Oracle Database stores dates in an internal numeric format: century, year, month, day, hours, minutes, and seconds.
- The default date display format is `DD-MON-RR`.
  - Enables you to store 21st-century dates in the 20th century by specifying only the last two digits of the year
  - Enables you to store 20th-century dates in the 21st century in the same way

```
SELECT last_name, hire_date
FROM   employees
WHERE  hire_date < '01-FEB-2016';
```

	LAST_NAME	HIRE_DATE
1	King	17-JUN-11
2	Kochhar	21-SEP-13

...

ORACLE®

The Oracle Database stores dates in an internal numeric format, representing the century, year, month, day, hours, minutes, and seconds.

The default display and input format for any date is `DD-MON-RR`. Valid Oracle dates are between January 1, 4712 B.C., and December 31, 9999 A.D.

In the example in the slide, the `HIRE_DATE` column output is displayed in the default format `DD-MON-RR`. However, dates are not stored in the database in this format. All the components of the date and time are stored. So, although a `HIRE_DATE` such as 17-JUN-11 is displayed as day, month, and year, there is also *time* and *century* information associated with the date. The complete date might be June 17, 2011, 5:10:43 PM.

RR Date Format

Current Year	Specified Date	RR Format	YY Format
1995	27-OCT-95	1995	1995
1995	27-OCT-17	2017	1917
2001	27-OCT-17	2017	2017
2001	27-OCT-95	1995	2095

		If the specified two-digit year is:	
		0–49	50–99
If two digits of the current year are:	0–49	The return date is in the current century	The return date is in the century before the current one
	50–99	The return date is in the century after the current one	The return date is in the current century

The RR date format is similar to the YY element, but you can use it to specify different centuries. Use the RR date format element instead of YY so that the century of the return value varies according to the specified two-digit year and the last two digits of the current year. The table in the slide summarizes the behavior of the RR element.

Current Year	Given Date	Interpreted (RR)	Interpreted (YY)
1994	27-OCT-95	1995	1995
1994	27-OCT-17	2017	1917
2001	27-OCT-17	2017	2017
2048	27-OCT-52	1952	2052
2051	27-OCT-47	2147	2047

Note the values shown in the last two rows of the preceding table.

This data is stored internally as follows:

CENTURY	YEAR	MONTH	DAY	HOUR	MINUTE	SECOND
19	03	06	17	17	10	43

Centuries and the Year 2000

When a record with a date column is inserted into a table, the *century* information is picked up from the SYSDATE function. However, when the date column is displayed on the screen, the century component is not displayed (by default).

The DATE data type uses 2 bytes for the year information, one for century and one for year. The century value is always included, whether or not it is specified or displayed. In this case, RR determines the default value for century on INSERT.

Using the SYSDATE Function

SYSDATE is a function that returns:

- Date
- Time

```
SELECT sysdate
FROM   dual;
```

	SYSDATE
1	08-FEB-16

SYSDATE is a date function that returns the system date. You can use SYSDATE just as you would use any other column name. For example, you can display the system date by selecting SYSDATE from a table. It is customary to select SYSDATE from a public table called DUAL.

Note: SYSDATE returns the current date and time set for the operating system on which the database resides. Therefore, if you are in a place in Australia and connected to a remote database in a location in the United States (U.S.), the sysdate function will return the U.S. date and time. In that case, you can use the CURRENT_DATE function that returns the current date in the session time zone.

Using the CURRENT_DATE and CURRENT_TIMESTAMP Functions

CURRENT_DATE returns the current date from the user session.

```
SELECT SESSIONTIMEZONE, CURRENT_DATE FROM DUAL;
```

	SESSIONTIMEZONE	SYSDATE
1	Etc/Universal	08-FEB-16

CURRENT_TIMESTAMP returns the current date and time from the user session.

```
SELECT SESSIONTIMEZONE, CURRENT_TIMESTAMP FROM DUAL;
```

	SESSIONTIMEZONE	CURRENT_TIMESTAMP
1	Etc/Universal	08-FEB-16 11.05.31.239152000 AM ETC/UNIVERSAL

The CURRENT_DATE and CURRENT_TIMESTAMP functions return the current date and current time stamp, respectively.

Note: The SESSIONTIMEZONE function returns the value of the current session's time zone. The return type is a time zone offset (a character type in the format '[+|-]TZH:TZM') or a time zone region name, depending on how the user specified the session time zone value in the most recent ALTER SESSION statement. The example in the slide shows that the session time zone is offset to UTC by –5 hours. Observe that the database time zone is different from the current session's time zone.

Arithmetic with Dates

- Add to or subtract a number from a date for a resultant date value.
- Subtract two dates to find the number of days between those dates.
- Add hours to a date by dividing the number of hours by 24.

You can perform the following operations:

Operation	Result	Description
date + number	Date	Adds a number of days to a date
date - number	Date	Subtracts a number of days from a date
date - date	Number of days	Subtracts one date from another
date + number/24	Date	Adds a number of hours to a date

ORACLE

Because the database stores dates as numbers, you can perform calculations using arithmetic operators such as addition and subtraction. You can add and subtract number constants as well as dates.

Using Arithmetic Operators with Dates

```
SELECT last_name, (SYSDATE-hire_date)/7 AS WEEKS
FROM   employees
WHERE  department_id = 90;
```

	LAST_NAME	WEEKS
1	King	242.7803207671957671957671957671957671196
2	Kochhar	124.63746362433862433862433862433862433 9
3	De Haan	369.208892195767195767195767195767195767

The example in the slide displays the last name and the number of weeks employed for all employees in department 90. It subtracts the date on which the employee was hired from the current date (SYSDATE) and divides the result by 7 to calculate the number of weeks that a worker has been employed.

Lesson Agenda

- Single-row SQL functions
- Character functions
- Nesting functions
- Number functions
- Working with dates
- Date functions

ORACLE

Date-Manipulation Functions

Function	Result
MONTH_BETWEEN	Number of months between two dates
ADD_MONTHS	Add calendar months to date
NEXT_DAY	Next occurrence date of the weekday specified
LAST_DAY	Last day of the month
ROUND	Round date
TRUNC	Truncate date

ORACLE

Date functions operate on Oracle dates. All date functions return a value of the `DATE` data type except `MONTHS_BETWEEN`, which returns a numeric value.

- **`MONTHS_BETWEEN`**`(`*`date1, date2`*`)`: Finds the number of months between *`date1`* and *`date2`*. The result can be positive or negative. If *`date1`* is later than *`date2`*, the result is positive; if *`date1`* is earlier than *`date2`*, the result is negative. The noninteger part of the result represents a portion of the month.
- **`ADD_MONTHS`**`(`*`date, n`*`)`: Adds *`n`* number of calendar months to *`date`*. The value of *`n`* must be an integer and can be negative.
- **`NEXT_DAY`**`(`*`date, 'char'`*`)`: Finds the next occurrence date of the weekday specified `('`*`char`*`')` following *`date`*. The value of *`char`* may be a number representing a day or a character string.
- **`LAST_DAY`**`(`*`date`*`)`: Finds the date of the last day of the month that contains *`date`*

The preceding list is a subset of the available date functions. `ROUND` and `TRUNC` number functions can also be used to manipulate the date values as shown below:

- **`ROUND`**`(`*`date`*`[,'`*`fmt`*`'])`: Returns *`date`* rounded to the unit that is specified by the format model *`fmt`*. If the format model *fmt* is omitted, *date* is rounded to the nearest day.
- **`TRUNC`**`(`*`date`*`[, '`*`fmt`*`'])`: Returns *`date`* with the time portion of the day truncated to the unit that is specified by the format model *`fmt`*. If the format model *`fmt`* is omitted, *`date`* is truncated to the nearest day.

The format models are covered in detail in the lessons titled “Using Conversion Functions” and “Using Conditional Expressions.”

Using Date Functions

Function	Result
MONTHS_BETWEEN('01-SEP-14','11-FEB-16')	19.6774194
ADD_MONTHS ('31-JAN-16',1)	'29-FEB-16'
NEXT_DAY ('01-FEB-16', 'FRIDAY')	'05-FEB-16'
LAST_DAY ('01-FEB-16')	'29-FEB-16'

ORACLE

In the example in the slide, the `ADD_MONTHS` function adds one month to the supplied date value "31-JAN-16" and returns "29-FEB-16." The function recognizes the year 2016 as the leap year and, therefore, returns the last day of the February month. If you change the input date value to "31-JAN-14," the function returns "28-FEB-14."

For example, display the employee number, hire date, number of months employed, six-month review date, first Friday after hire date, and the last day of the hire month for all employees who have been employed for fewer than 150 months.

```
SELECT employee_id, hire_date, MONTHS_BETWEEN (SYSDATE, hire_date)
TENURE, ADD_MONTHS (hire_date, 6) REVIEW,  NEXT_DAY (hire_date,
'FRIDAY'), LAST_DAY(hire_date)
FROM employees WHERE  MONTHS_BETWEEN (SYSDATE, hire_date) < 150;
```

Using ROUND and TRUNC Functions with Dates

If SYSDATE is 18-JUL-15:

Function	Result
ROUND(SYSDATE,'MONTH')	'01-AUG-15'
ROUND(SYSDATE,'YEAR')	'01-JAN-16'
TRUNC(SYSDATE,'MONTH')	'01-JUL-15'
TRUNC(SYSDATE,'YEAR')	'01-JAN-15'

ORACLE

The ROUND and TRUNC functions can be used for number and date values. When used with dates, these functions round or truncate to the specified format model. Therefore, you can round dates to the nearest year or month. If the format model is month, dates 1-15 result in the first day of the current month. Dates 16-31 result in the first day of the next month. If the format model is year, months 1-6 result in January 1 of the current year. Months 7-12 result in January 1 of the next year.

Example

Compare the hire dates for all employees who started in 2014. Display the employee number, hire date, and starting month using the ROUND and TRUNC functions.

```
SELECT employee_id, hire_date,
ROUND(hire_date, 'MONTH'), TRUNC(hire_date, 'MONTH')
FROM   employees
WHERE  hire_date LIKE '%14';
```

Quiz

Which four of the following statements are true about single-row functions?

a. Manipulate data items
b. Accept arguments and return one value per argument
c. Act on each row that is returned
d. Return one result per set of rows
e. Never modify the data type
f. Can be nested
g. Accept arguments that can be a column or an expression

Answer: a, c, f, g

Quiz

Arithmetic operation on dates always returns a date.

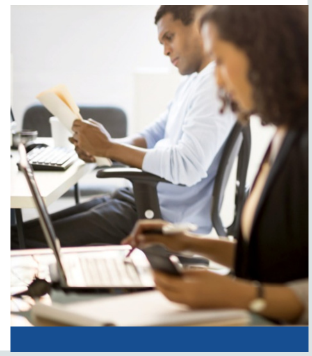
a. True
b. False

Answer: b

Summary

In this lesson, you should have learned how to:

- Use the various types of functions available in SQL
- Use the character, number, and date functions in `SELECT` statements

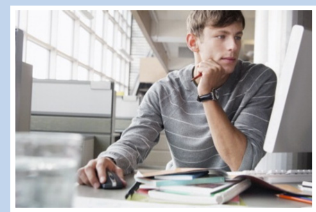


ORACLE

Practice 7: Overview

This practice covers the following topics:

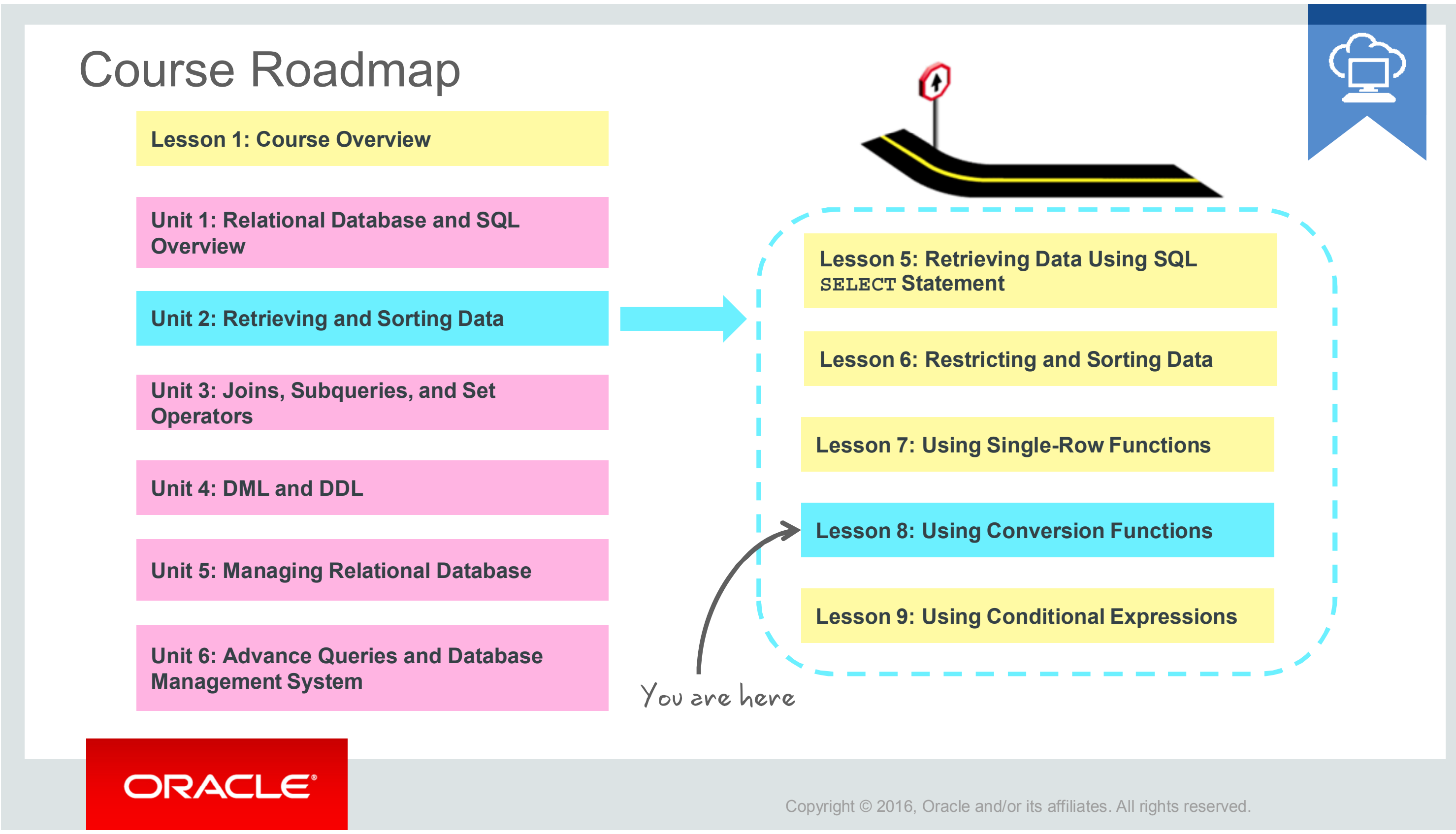
- Writing a query that displays the `SYSDATE`
- Creating queries that require the use of numeric, character, and date functions
- Performing calculations of years and months of a course for a student



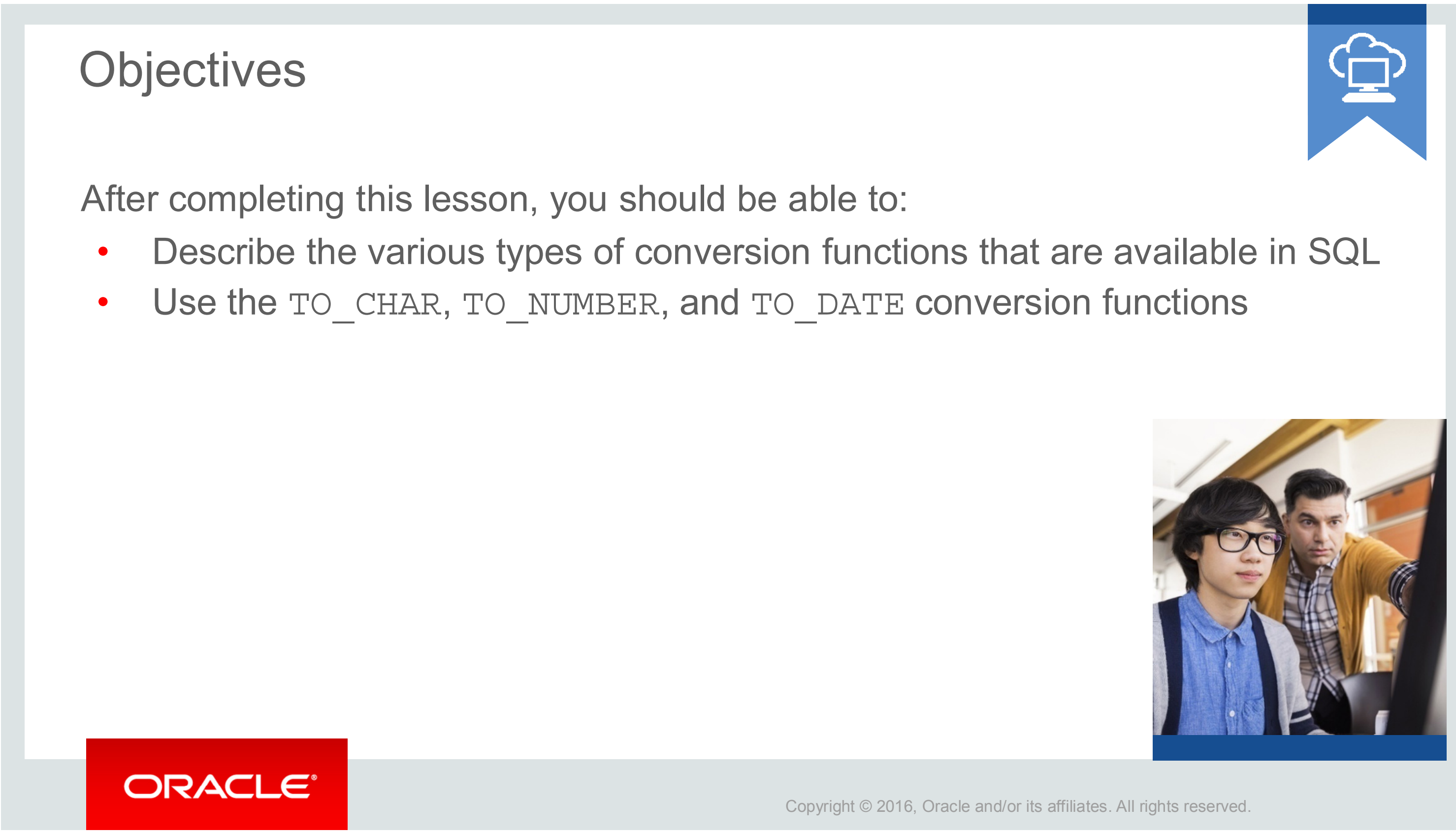
ORACLE

This practice provides a variety of exercises using different functions that are available for character, number, and date data types.

Lesson 8: Using Conversion Functions



In Unit 2, you will dive into the concepts of SQL. You will learn to use the SQL SELECT statement to retrieve data from database tables and restrict and sort the retrieved data. You will also learn about single-row functions, conversion functions, and conditional expressions in SQL.



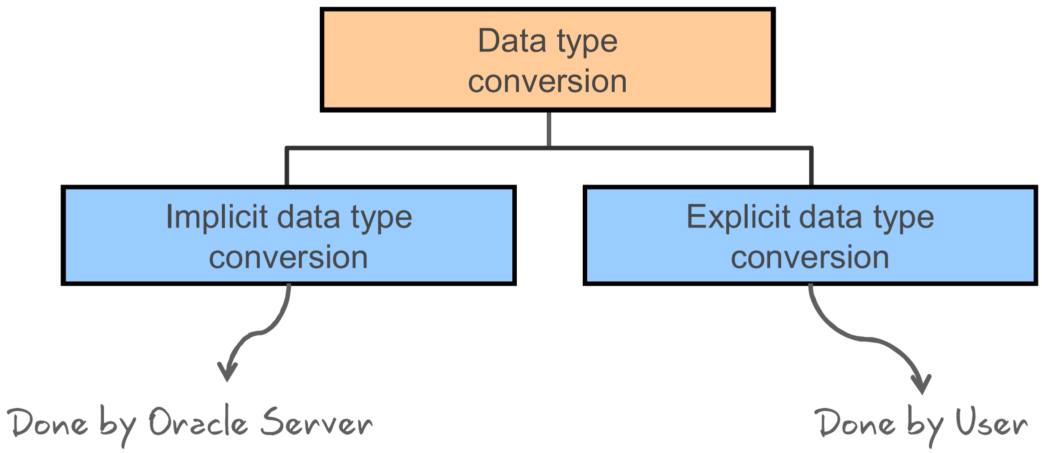
This lesson focuses on functions that convert data from one type to another (for example, conversion from character data to numeric data) and discusses the conditional expressions in SQL `SELECT` statements.

Lesson Agenda

- Implicit and explicit data type conversion
- TO_CHAR, TO_DATE, TO_NUMBER functions
- General functions:
  - NVL
  - NVL2
  - NULLIF
  - COALESCE

ORACLE

Conversion Functions



In addition to Oracle data types, columns of tables in an Oracle Database can be defined by using the American National Standards Institute (ANSI), DB2, and SQL/DS data types. However, the Oracle server internally converts such data types to Oracle data types.

In some cases, the Oracle server receives data of one data type where it expects data of a different data type. When this happens, the Oracle server can automatically convert the data to the expected data type. This data type conversion can be done *implicitly* by the Oracle server or *explicitly* by the user.

Implicit data type conversions work according to the rules explained in the following slides.

Explicit data type conversions are performed by using the conversion functions. Conversion functions convert a value from one data type to another. Generally, the form of the function names follows the convention `data type TO data type`. The first data type is the input data type and the second data type is the output.

Note: Although implicit data type conversion is available, it is recommended that you do the explicit data type conversion to ensure the reliability of your SQL statements.

Implicit Data Type Conversion

In expressions, the Oracle server can automatically convert the following:

String or character should be a valid number

From	To
VARCHAR2 or CHAR	NUMBER
VARCHAR2 or CHAR	DATE

String should be a valid date

ORACLE

The Oracle server can automatically perform data type conversion in an expression. For example, the expression `hire_date > '01-JAN-90'` results in the implicit conversion from the string `'01-JAN-90'` to a date. Therefore, a `VARCHAR2` or `CHAR` value can be implicitly converted to a number or date data type in an expression.

Note: `CHAR` to `NUMBER` conversions succeed only if the character string represents a valid number.

Implicit Data Type Conversion

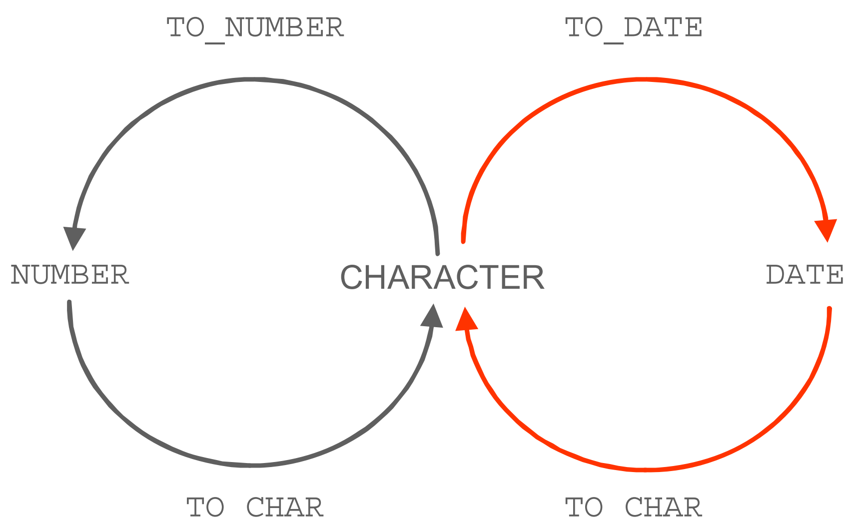
For expression evaluation, the Oracle server can automatically convert the following:

From	To
NUMBER	VARCHAR2 or CHAR
DATE	VARCHAR2 or CHAR

ORACLE

In general, the Oracle server uses the rule for expressions when a data type conversion is needed. For example, the expression `job_id = 2` results in the implicit conversion of the number `2` to the string "2" because `job_id` is a `VARCHAR(2)` column.

Explicit Data Type Conversion



SQL provides three functions to convert a value from one data type to another:

Function	Purpose
`TO_CHAR(`*`number|date [, fmt [, nlsparams] ]`*`)`	Converts a number or date value to a `VARCHAR2` character string with the format model *`fmt`* **Number conversion:** The `nlsparams` parameter specifies the following characters, which are returned by number format elements: • Decimal character • Group separator • Local currency symbol • International currency symbol If `nlsparams` or any other parameter is omitted, this function uses the default parameter values for the session.

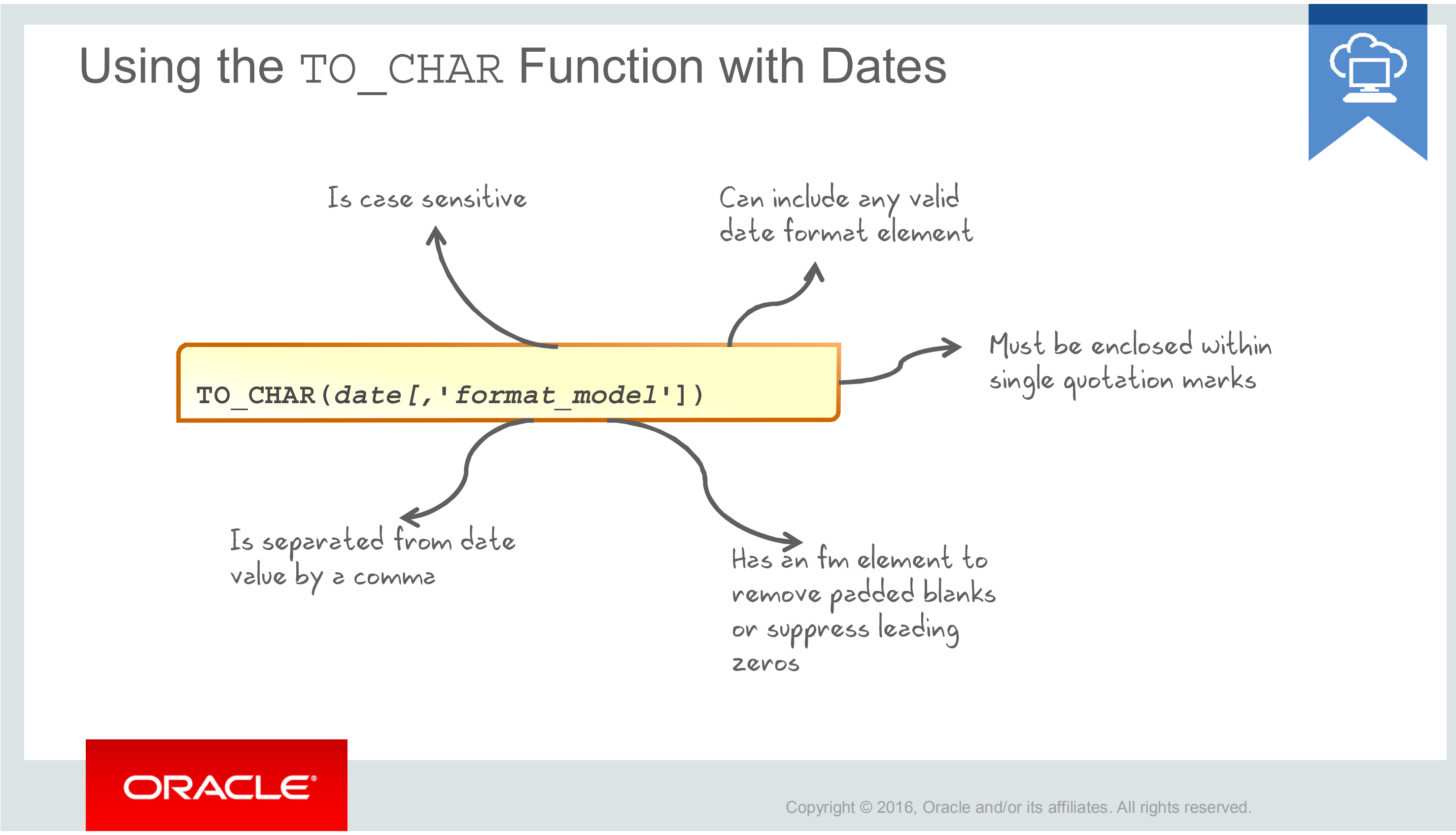
Function	Purpose
TO_NUMBER(*char[,fmt[, nlsparams]]*)	Converts a character string containing digits to a number in the format specified by the optional format model *fmt* The nlsparams parameter has the same purpose in this function as in the TO_CHAR function for number conversion.
TO_DATE(c*har[,fmt*[,*nlsparams*]])	Converts a character string representing a date to a date value according to *fmt* that is specified. If *fmt* is omitted, the format is DD-MON-YY. The nlsparams parameter has the same purpose in this function as in the TO_CHAR function for date conversion.

Note: The list of functions mentioned in this lesson includes only some of the available conversion functions.

For more information, see the "Conversion Functions" section in *Oracle Database SQL Language Reference* for 12*c* database.

Lesson Agenda

- Implicit and explicit data type conversion
- TO_CHAR, TO_DATE, TO_NUMBER functions
- General functions:
  - NVL
  - NVL2
  - NULLIF
  - COALESCE



TO_CHAR converts a datetime data type to a value of VARCHAR2 data type in the format specified by the *format_model*. A format model is a character literal that describes the format of datetime stored in a character string. For example, the datetime format model for the string '11-Nov-2015' is 'DD-Mon-YYYY'. You can use the TO_CHAR function to convert a date from its default format to the one that you specify.

Guidelines

- The format model must be enclosed within single quotation marks and is case-sensitive.
- The format model can include any valid date format element. But be sure to separate the date value from the format model with a comma.
- The names of days and months in the output are automatically padded with blanks.
- To remove padded blanks or to suppress leading zeros, use the fill mode *fm* element.

Example

```
SELECT employee_id, TO_CHAR(hire_date, 'MM/YY') Month_Hired
FROM   employees
WHERE  last_name = 'Higgins';
```

Elements of the Date Format Model

Element	Result
YYYY	Full year in numbers
YEAR	Year spelled out (in English)
MM	Two-digit value for the month
MONTH	Full name of the month
MON	Three-letter abbreviation of the month
DY	Three-letter abbreviation of the day of the week
DAY	Full name of the day of the week
DD	Numeric day of the month

ORACLE

Element	Description
SCC or CC	Century; server prefixes B.C. date with -
Years in dates YYYY or SYYYY	Year; server prefixes B.C. date with -
YYY or YY or Y	Last three, two, or one digit of the year
Y,YYY	Year with comma in this position
IYYY, IYY, IY, I	Four-, three-, two-, or one-digit year based on the ISO standard
SYEAR or YEAR	Year spelled out; server prefixes B.C. date with -
BC or AD	Indicates B.C. or A.D. year
B.C. or A.D.	Indicates B.C. or A.D. year using periods
Q	Quarter of year
MM	Month: two-digit value
MONTH	Name of the month padded with blanks to a length of nine characters
MON	Name of the month, three-letter abbreviation
RM	Roman numeral month
WW or W	Week of the year or month
DDD or DD or D	Day of the year, month, or week
DAY	Name of the day padded with blanks to a length of nine characters
DY	Name of the day; three-letter abbreviation
J	Julian day; the number of days since December 31, 4713 B.C.
IW	Weeks in the year from ISO standard (1 to 53)

Elements of the Date Format Model

- Use time elements to format the time portion of the date:

HH24:MI:SS AM	15:45:32 PM

- Add character strings by enclosing them within double quotation marks:

DD "of" MONTH	12 of OCTOBER

- Use number suffixes to spell out numbers:

ddspth	fourteenth

ORACLE

Use the formats that are listed in the following tables to display time information and literals, and to change numerals to spelled numbers:

Element	**Description**
AM or PM	Meridian indicator
A.M. or P.M.	Meridian indicator with periods
HH or HH12	12 hour format
HH24	24 hour format
MI	Minute (0–59)
SS	Second (0–59)
SSSSS	Seconds past midnight (0–86399)

Other Formats

Element	Description
/ . ,	Punctuation is reproduced in the result.
"of the"	Quoted string is reproduced in the result.

Specifying Suffixes to Influence Number Display

Element	Description
TH	Ordinal number (for example, DDTH for 4TH)
SP	Spelled-out number (for example, DDSP for FOUR)
SPTH or THSP	Spelled-out ordinal numbers (for example, DDSPTH for FOURTH)

Using the TO_CHAR Function with Dates

```
SELECT last_name,
       TO_CHAR(hire_date, 'fmDD Month YYYY')
       AS HIREDATE
FROM   employees;
```

	LAST_NAME	HIREDATE
1	King	17 June 2011
2	Kochhar	21 September 2013
3	De Haan	13 January 2009
4	Hunold	3 January 2014
5	Ernst	21 May 2015
6	Lorentz	7 February 2015
7	Mourgos	16 November 2015
8	Rajs	17 October 2011

...

The SQL statement in the slide displays the last names and hire dates for all the employees. The hire date appears as 17 June 2011.

Example

Modify the example in the slide to display the dates in a format that appears as "Seventeenth of June 2011 12:00:00 AM."

```
SELECT   last_name,
TO_CHAR(hire_date,'fmDdspth "of" Month YYYY fmHH:MI:SS AM')
HIREDATE
FROM     employees;
```

Notice that the month follows the format model specified; in other words, the first letter is capitalized and the rest are in lowercase.

Using the TO_CHAR Function with Numbers

```
TO_CHAR(number[, 'format_model'])
```

These are some of the format elements that you can use with the TO_CHAR function to display a number value as a character:

Element	Result
9	Represents a number
0	Forces a zero to be displayed
$	Places a floating dollar sign
L	Uses the floating local currency symbol
.	Prints a decimal point
,	Prints a comma as a thousands indicator

ORACLE

When working with number values, such as character strings, you should convert those numbers to the character data type using the TO_CHAR function, which translates a value of NUMBER data type to VARCHAR2 data type. This technique is especially useful with concatenation.

Number Format Elements

If you are converting a number to the character data type, you can use the following format elements:

Element	Description	Example	Result
9	Numeric position (number of 9s determine display width)	999999	1234
0	Display leading zeros	099999	001234
$	Floating dollar sign	$999999	$1234
L	Floating local currency symbol	L999999	FF1234
D	Returns the decimal character in the specified position. The default is a period (.).	9999D99	1234.00
.	Decimal point in position specified	999999.99	1234.00
G	Returns the group separator in the specified position. You can specify multiple group separators in a number format model.	9G999	1,234
,	Comma in position specified	999,999	1,234
MI	Minus signs to right (negative values)	999999MI	1234-
PR	Parenthesize negative numbers	999999PR	<1234>
EEEE	Scientific notation (format must specify four Es)	99.999EEEE	1.234E+03
U	Returns in the specified position the “Euro” (or other) dual currency	U9999	€1234
V	Multiply by 10 *n* times (*n* = number of 9s after V)	9999V99	123400
S	Returns the negative or positive value	S9999	-1234 or +1234
B	Display zero values as blank, not 0	B9999.99	1234.00

Using the TO_CHAR Function with Numbers

```
SELECT TO_CHAR(salary, '$99,999.00') SALARY
FROM   employees
WHERE  last_name = 'Ernst';
```

	SALARY
1	$6,000.00

Displays the salary of an employee as a string of characters according to the given format model

- The Oracle server displays a string of number signs (#) in place of a whole number whose digits exceed the number of digits provided in the format model.
- The Oracle server rounds the stored decimal value to the number of decimal places provided in the format model.

Using the TO_NUMBER and TO_DATE Functions

Convert a character string to a number format using the TO_NUMBER function.

```
TO_NUMBER(char[, 'format_model'])
```

Convert a character string to a date format using the TO_DATE function.

```
TO_DATE(char[, 'format_model'])
```

These functions have an fx modifier. This modifier specifies the exact match for the character argument and date format model of a TO_DATE function.

ORACLE

You may want to convert a character string to either a number or a date. To accomplish this task, use the `TO_NUMBER` or `TO_DATE` function. The format model that you select is based on the previously demonstrated format elements.

The `fx` modifier specifies the exact match for the character argument and date format model of a `TO_DATE` function:

- Punctuation and quoted text in the character argument must exactly match (except for case) the corresponding parts of the format model.
- The character argument cannot have extra blanks. Without `fx`, the Oracle server ignores extra blanks.
- Numeric data in the character argument must have the same number of digits as the corresponding element in the format model. Without `fx`, the numbers in the character argument can omit leading zeros.

Example

Display the name and hire date for all employees who started on May 24, 2015. There are two spaces after the month *May* and before the number *24* in the following example. Because the `fx` modifier is used, an exact match is required and the spaces after the word *May* are not recognized:

```
SELECT last_name, hire_date
FROM   employees
WHERE  hire_date = TO_DATE('May  24, 2015', 'fxMonth DD, YYYY');
```

The resulting error output looks like this:

```
ORA-01858: a non-numeric character was found where a numeric was
expected
01858. 00000 - "a non-numeric character was found where a numeric was
  expected"
*Cause: The input data to be converted using a date format model was
  incorrect. The input data did not contain a number where a number
  was required by the format model.
*Action: Fix the input data or the date format model to make sure the
  elements match in number and type. Then retry the operation.
```

To see the output, correct the query by deleting the extra space between 'May' and '24'.

```
SELECT last_name, hire_date
FROM   employees
WHERE  hire_date = TO_DATE('May 24, 2015', 'fxMonth DD, YYYY');
```

Using the TO_CHAR and TO_DATE Functions with the RR Date Format

To find employees hired before 2010, use the RR date format, which produces the same results whether the command is run in 1999 or now:

```
SELECT last_name, TO_CHAR(hire_date, 'DD-Mon-YYYY')
FROM   employees
WHERE hire_date < TO_DATE('01-Jan-10','DD-Mon-RR');
```

	LAST_NAME	TO_CHAR(HIRE_DATE,'DD-MON-YYYY')
1	De Haan	13-Jan-2009

To find employees who were hired before 2010, the RR format can be used. Because the current year is greater than 1999, the RR format interprets the year portion of the date from 2000 to 2050.

Alternatively, the following command results in no rows being selected because the YY format interprets the year portion of the date in the century previous to the current one (1990).

```
SELECT last_name, TO_CHAR(hire_date, 'DD-Mon-yyyy')
FROM   employees
WHERE  TO_DATE(hire_date, 'DD-Mon-yy') < '01-Jan-90';
```

Notice that no rows are retrieved from the preceding query.

Lesson Agenda

- Implicit and explicit data type conversion
- `TO_CHAR`, `TO_DATE`, `TO_NUMBER` functions
- General functions:
  - `NVL`
  - `NVL2`
  - `NULLIF`
  - `COALESCE`

General Functions

The following functions work with any data type and pertain to using nulls:

- `NVL (expr1, expr2)`
- `NVL2 (expr1, expr2, expr3)`
- `NULLIF (expr1, expr2)`
- `COALESCE (expr1, expr2, ..., exprn)`

These functions work with any data type and pertain to the use of null values in the expression list.

Function	Description
`NVL`	Converts a null value to an actual value
`NVL2`	If `expr1` is not null, `NVL2` returns `expr2`. If `expr1` is null, `NVL2` returns `expr3`. The argument `expr1` can have any data type.
`NULLIF`	Compares two expressions and returns null if they are equal; returns the first expression if they are not equal
`COALESCE`	Returns the first non-null expression in the expression list

Note: For more information about the hundreds of functions available, see the "Functions" section in *Oracle Database SQL Language Reference* for 12*c* database.

NVL Function

NVL (expr1, expr2)

Converts a null value to an actual value:

- Data types that can be used are date, character, and number.
- Data types must match:
  - NVL(commission_pct,0)
  - NVL(hire_date,'01-JAN-16')
  - NVL(job_id,'No Job Yet')

ORACLE

To convert a null value to an actual value, use the NVL function.

Syntax

NVL (*expr1*, *expr2*)

In the syntax:

- *expr1* is the source value or expression that may contain a null
- *expr2* is the target value for converting the null

You can use the NVL function with any data type, but the return value is always the same as the data type of *expr1*.

NVL Conversions for Various Data Types

Data Type	Conversion Example
NUMBER	NVL(*number_column*,9)
DATE	NVL(*date_column*, '01-JAN-16')
CHAR or VARCHAR2	NVL(*character_column*, 'Unavailable')

Using the NVL Function

```
SELECT last_name, salary, NVL(commission_pct, 0),
   (salary*12) + (salary*12*NVL(commission_pct, 0)) AN_SAL
FROM employees;
```

	LAST_NAME	SALARY	NVL(COMMISSION_PCT,0)	AN_SAL
1	King	24000	0	288000
2	Kochhar	17000	0	204000
3	De Haan	17000	0	204000
4	Hunold	9000	0	108000
5	Ernst	6000	0	72000
6	Lorentz	4200	0	50400
7	Mourgos	5800	0	69600
8	Rajs	3500	0	42000
9	Davies	3100	0	37200
10	Matos	2600	0	31200

...

To calculate the annual compensation of all employees, you need to multiply the monthly salary by 12 and then add the commission percentage to the result:

```
SELECT last_name, salary, commission_pct,
 (salary*12) + (salary*12*commission_pct) AN_SAL
FROM   employees;
```

Notice that the annual compensation is calculated for only those employees who earn a commission. If any column value in an expression is null, the result is null. To calculate values for all employees, you must convert the null value to a number before applying the arithmetic operator. In the example in the slide, the NVL function is used to convert null values to zero.

Using the NVL2 Function

```
SELECT last_name,  salary, commission_pct,
       NVL2(commission_pct,
            'SAL+COMM', 'SAL') income
FROM   employees WHERE department_id IN (50, 80);
```

	LAST_NAME	SALARY	COMMISSION_PCT	INCOME
1	Mourgos	5800	(null)	SAL
2	Rajs	3500	(null)	SAL
3	Davies	3100	(null)	SAL
4	Matos	2600	(null)	SAL
5	Vargas	2500	(null)	SAL
6	Zlotkey	10500	0.2	SAL+COMM
7	Abel	11000	0.3	SAL+COMM
8	Taylor	8600	0.2	SAL+COMM

ORACLE

The NVL2 function examines the first expression. If the first expression is not null, the NVL2 function returns the second expression. If the first expression is null, the third expression is returned.

Syntax

`NVL2(`*`expr1, expr2, expr3`*`)`

In the syntax:

- *expr1* is the source value or expression that may contain a null
- *expr2* is the value that is returned if *expr1* is not null
- *expr3* is the value that is returned if *expr1* is null

In the example shown in the slide, the COMMISSION_PCT column is examined. If a value is detected, the text literal value of SAL+COMM is returned. If the COMMISSION_PCT column contains a null value, the text literal value of SAL is returned.

Note: The argument *expr1* can be any data type, but *expr2* and *expr3* should be the same data type.

Using the NULLIF Function

```
SELECT first_name, LENGTH(first_name) "expr1",
       last_name,  LENGTH(last_name)  "expr2",
       NULLIF(LENGTH(first_name), LENGTH(last_name)) result
FROM   employees;
```

	FIRST_NAME	expr1	LAST_NAME	expr2	RESULT
1	Ellen	5	Abel	4	5
2	Curtis	6	Davies	6	(null)
3	Lex	3	De Haan	7	3
4	Bruce	5	Ernst	5	(null)
5	Pat	3	Fay	3	(null)
6	William	7	Gietz	5	7
7	Kimberely	9	Grant	5	9
8	Michael	7	Hartstein	9	7
9	Shelley	7	Higgins	7	(null)

...

ORACLE

The NULLIF function compares two expressions.

Syntax

```
NULLIF (expr1, expr2)
```

In the syntax:

- NULLIF compares *expr1* and *expr2*. If they are equal, the function returns null. If they are not, the function returns *expr1*. However, you cannot specify the literal NULL for *expr1*.

In the example shown in the slide, the length of the first name in the EMPLOYEES table is compared with the length of the last name in the EMPLOYEES table. When the lengths of the names are equal, a null value is displayed. When the lengths of the names are not equal, the length of the first name is displayed.

Using the COALESCE Function

```
COALESCE (expr1, expr2, ..., exprn)
```

The advantage of the COALESCE function over the NVL function is that the COALESCE function can take multiple alternative values.

If the first expression is not null, the COALESCE function returns that expression; otherwise, it does a COALESCE of the remaining expressions.

The `COALESCE` function returns the first non-null expression in the list.

Syntax

```
COALESCE (expr1, expr2, ... exprn)
```

In the syntax:

- *expr1* returns this expression if it is not null
- *expr2* returns this expression if the first expression is null and this expression is not null
- *exprn* returns this expression if the preceding expressions are null

Note that all expressions must be of the same data type.

Using the COALESCE Function

```
SELECT last_name, salary, commission_pct,
COALESCE((salary+(commission_pct*salary)), salary+2000)"New Salary"
FROM   employees;
```

	LAST_NAME	SALARY	COMMISSION_PCT	New Salary
1	King	24000	(null)	26000
2	Kochhar	17000	(null)	19000
3	De Haan	17000	(null)	19000
4	Hunold	9000	(null)	11000
5	Ernst	6000	(null)	8000
6	Lorentz	4200	(null)	6200
7	Mourgos	5800	(null)	7800
8	Rajs	3500	(null)	5500
9	Davies	3100	(null)	5100
10	Matos	2600	(null)	4600
11	Vargas	2500	(null)	4500
12	Zlotkey	10500	0.2	12600
13	Abel	11000	0.3	14300
14	Taylor	8600	0.2	10320
15	Grant	7000	0.15	8050
16	Whalen	4400	(null)	6400
17	Hartstein	13000	(null)	15000
18	Fay	6000	(null)	8000
19	Higgins	12008	(null)	14008
20	Gietz	8300	(null)	10300

ORACLE

In the example shown in the slide, for the employees who do not get any commission, your organization wants to give a salary increment of $2,000 and for employees who get commission, the query should compute the new salary that is equal to the existing salary added to the commission amount.

Note: Examine the output. For employees who do not get any commission, the New Salary column shows the salary incremented by $2,000 and for employees who get commission, the New Salary column shows the computed commission amount added to the salary.

Quiz

The `TO_NUMBER` function converts either character strings or date values to a number in the format specified by the optional format model.

a. True

b. False

Answer: b

Quiz

The `NVL` function can be used with any data type.

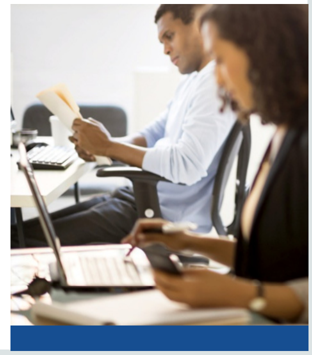
a. True
b. False

Answer: a

Summary

In this lesson, you should have learned how to:

- Alter date formats for display by using functions
- Convert column data types using functions
- Use NVL functions



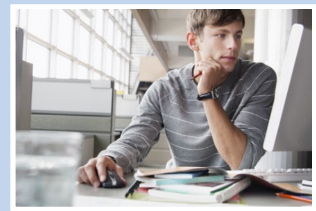
ORACLE

Remember the following:

- Conversion functions can convert character, date, and numeric values, and include TO_CHAR, TO_DATE, and TO_NUMBER.
- There are several functions that pertain to nulls, including NVL, NVL2, NULLIF, and COALESCE.

Practice 8: Overview

This practice covers creating queries that use TO_CHAR, TO_DATE, and other DATE functions.



This practice provides a variety of exercises for using the TO_CHAR and TO_DATE functions.

Remember that for nested functions, the results are evaluated from the innermost function to the outermost function.

Lesson 9: Using Conditional Expressions

Course Roadmap

- Lesson 1: Course Overview
- Unit 1: Relational Database and SQL Overview
- Unit 2: Retrieving and Sorting Data
  - Lesson 5: Retrieving Data Using SQL `SELECT` Statement
  - Lesson 6: Restricting and Sorting Data
  - Lesson 7: Using Single-Row Functions
  - Lesson 8: Using Conversion Functions
  - Lesson 9: Using Conditional Expressions
- Unit 3: Joins, Subqueries, and Set Operators
- Unit 4: DML and DDL
- Unit 5: Managing Relational Database
- Unit 6: Advance Queries and Database Management System

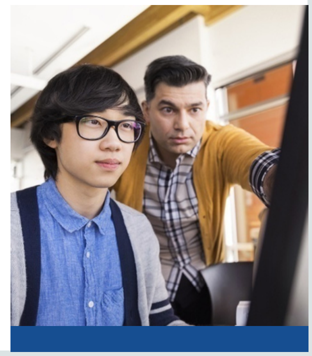
You are here

ORACLE

In Unit 2, you will dive into the concepts of SQL. You will learn to use the SQL SELECT statement to retrieve data from database tables, and restrict and sort the retrieved data. You will also learn about single-row functions, conversion functions, and conditional expressions in SQL.

Objectives

After completing this lesson, you should be able to apply conditional expressions in a `SELECT` statement.



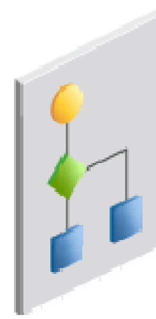
This lesson discusses the conditional expressions in SQL `SELECT` statements. You can use CASE or DECODE to apply conditional expressions in a `SELECT` statement.

Lesson Agenda

- Conditional expressions:
  - CASE
  - Searched CASE
  - DECODE

Conditional Expressions

- Provide the use of the `IF-THEN-ELSE` logic within a SQL statement
- Use the following methods:
  - `CASE` expression
  - Searched `CASE` expression
  - `DECODE` function



The two methods that are used to implement conditional processing (`IF-THEN-ELSE` logic) in a SQL statement are the `CASE` expression and the `DECODE` function.

Note: The `CASE` expression complies with the ANSI SQL. The `DECODE` function is specific to Oracle syntax.

CASE Expression

Facilitates conditional inquiries by doing the work of an IF-THEN-ELSE statement:

```
CASE expr WHEN comparison_expr1 THEN return_expr1
         [WHEN comparison_expr2 THEN return_expr2
          WHEN comparison_exprn THEN return_exprn
          ELSE else_expr]
END
```

expr and comparison_expr must be of the same data type.

All the return values must be of the same data type.

ORACLE

CASE expressions allow you to use the IF-THEN-ELSE logic in SQL statements without having to invoke procedures.

In a simple CASE expression, the Oracle server searches for the first WHEN ... THEN pair for which expr is equal to comparison_expr and returns return_expr. If none of the WHEN ... THEN pairs meet this condition, and if an ELSE clause exists, the Oracle server returns else_expr. Otherwise, the Oracle server returns a null. You cannot specify the literal NULL for all the return_exprs and the else_expr.

The expressions expr and comparison_expr must be of the same data type, which can be CHAR, VARCHAR2, NCHAR, NVARCHAR2, NUMBER, BINARY_FLOAT, or BINARY_DOUBLE or must all have a numeric data type. All of the return values (return_expr) must be of the same data type.

Using the CASE Expression

```
SELECT last_name, job_id, salary,
       CASE job_id WHEN 'IT_PROG'  THEN  1.10*salary
                   WHEN 'ST_CLERK' THEN  1.15*salary
                   WHEN 'SA_REP'   THEN  1.20*salary
       ELSE      salary END   REVISED_SALARY
FROM   employees;
```

	LAST_NAME	JOB_ID	SALARY	REVISED_SALARY
1	King	AD_PRES	24000	24000
4	Hunold	IT_PROG	9000	9900
5	Ernst	IT_PROG	6000	6600
6	Lorentz	IT_PROG	4200	4620
7	Mourgos	ST_MAN	5800	5800
8	Rajs	ST_CLERK	3500	4025
9	Davies	ST_CLERK	3100	3565
10	Matos	ST_CLERK	2600	2990
11	Vargas	ST_CLERK	2500	2875
...				
13	Abel	SA_REP	11000	13200
14	Taylor	SA_REP	8600	10320
15	Grant	SA_REP	7000	8400

CASE evaluates whether job_id is the same as the comparison_expr('IT_PROG', 'ST_CLERK', or 'SA_REP').

ORACLE

In the SQL statement in the slide, the value of JOB_ID is decoded. If JOB_ID is IT_PROG, the salary increase is 10%; if JOB_ID is ST_CLERK, the salary increase is 15%; if JOB_ID is SA_REP, the salary increase is 20%. For all other job roles, there is no increase in salary.

The same statement can be written with the DECODE function.

Note: The column label should be in double quotation marks if it is two or more words separated by spaces. For example, if the column label is REVISED SALARY, enclose it in double quotation marks. Using single quotation marks returns error.

Searched CASE Expression

```
CASE
    WHEN condition1 THEN use_expression1
    WHEN condition2 THEN use_expression2
    WHEN condition3 THEN use_expression3
    ELSE default_use_expression
END
```

CASE evaluates the conditions independently under each WHEN option.

If no condition is true, default_use_expression is returned.

ORACLE

In a searched CASE expression, the search occurs from left to right until an occurrence of the listed condition is found, and then it returns the return expression. If no condition is found to be true, and if an ELSE clause exists, the return expression in the ELSE clause is returned; otherwise, a NULL is returned. The searched CASE evaluates the conditions independently under each of the WHEN options.

The difference between the CASE expression and the searched CASE expression is that in a searched CASE expression, you specify a condition or predicate instead of a comparison_expression after the WHEN keyword.

For both simple and searched CASE expressions, all of the return_exprs must either have the same data type CHAR, VARCHAR2, NCHAR, NVARCHAR2, NUMBER, BINARY_FLOAT, or BINARY_DOUBLE or must all have a numeric data type.

Searched CASE Expression

```
SELECT last_name,salary,
(CASE WHEN salary<5000 THEN 'Low'
      WHEN salary<10000 THEN 'Medium'
      WHEN salary<20000 THEN 'Good'
      ELSE 'Excellent'
END) qualified_salary
FROM employees;
```

	LAST_NAME	SALARY	QUALIFIED_SALARY
1	King	24000	Excellent
2	Kochhar	17000	Good
3	De Haan	17000	Good
4	Hunold	9000	Medium
5	Ernst	6000	Medium
6	Lorentz	4200	Low

...

The code in the slide is an example of the searched CASE expression. For each row, the condition is checked. If salary < 5000, 'Low' is displayed as the QUALIFIED_SALARY.

Lesson Agenda

- Conditional expressions:
  - CASE
  - Searched CASE
  - DECODE

DECODE Function

Facilitates conditional inquiries by doing the work of a CASE expression or an IF-THEN-ELSE statement:

```
DECODE(col|expression, search1, result1
                      [, search2, result2,...,]
                      [, default])
```

Compares column or expression with the search, and if they are same, returns result

The DECODE function decodes an expression in a way similar to the IF-THEN-ELSE logic that is used in various languages. The DECODE function decodes *expression* after comparing it to each *search* value. If the expression is the same as *search*, *result* is returned.

If the default value is omitted, a null value is returned in case a search value does not match any of the result values.

Using the DECODE Function

```
SELECT last_name, job_id, salary,
       DECODE(job_id, 'IT_PROG',  1.10*salary,
                      'ST_CLERK', 1.15*salary,
                      'SA_REP',   1.20*salary,
              salary)
       REVISED_SALARY
FROM   employees;
```

	LAST_NAME	JOB_ID	SALARY	REVISED_SALARY
...				
4	Hunold	IT_PROG	9000	9900
5	Ernst	IT_PROG	6000	6600
6	Lorentz	IT_PROG	4200	4620
7	Mourgos	ST_MAN	5800	5800
8	Rajs	ST_CLERK	3500	4025
9	Davies	ST_CLERK	3100	3565
10	Matos	ST_CLERK	2600	2990
11	Vargas	ST_CLERK	2500	2875
12	Zlotkey	SA_MAN	10500	10500
...				
13	Abel	SA_REP	11000	13200
14	Taylor	SA_REP	8600	10320
15	Grant	SA_REP	7000	8400

ORACLE

In the SQL statement in the slide, the value of `JOB_ID` is tested. If `JOB_ID` is `IT_PROG`, the salary increase is 10%; if `JOB_ID` is `ST_CLERK`, the salary increase is 15%; if `JOB_ID` is `SA_REP`, the salary increase is 20%. For all other job roles, there is no increase in salary.

The same statement can be expressed in pseudocode as an `IF-THEN-ELSE` statement:

```
IF job_id = 'IT_PROG'     THEN   salary = salary*1.10
IF job_id = 'ST_CLERK'    THEN   salary = salary*1.15
IF job_id = 'SA_REP'      THEN   salary = salary*1.20
ELSE salary = salary
```

Using the DECODE Function

Display the applicable tax rate for each employee in department 80:

```
SELECT last_name, salary,
       DECODE (TRUNC(salary/2000, 0),
                         0, 0.00,
                         1, 0.09,
                         2, 0.20,
                         3, 0.30,
                         4, 0.40,
                         5, 0.42,
                         6, 0.44,
                            0.45) TAX_RATE
FROM   employees
WHERE  department_id = 80;
```


This slide shows another example that uses the DECODE function. In this example, you determine the tax rate for each employee in department 80 based on the monthly salary. The tax rates are as follows:

Monthly Salary Range	***Tax Rate***
$0.00–1,999.99	00%
$2,000.00–3,999.99	09%
$4,000.00–5,999.99	20%
$6,000.00–7,999.99	30%
$8,000.00–9,999.99	40%
$10,000.00–11,999.99	42%
$12,200.00–13,999.99	44%
$14,000.00 or greater	45%

Note: The TRUNC function truncates the column, expression, or value to *n* decimal places.

Quiz

To apply `IF-THEN-ELSE` logic within a SQL statement, you must use `CASE` or `DECODE`.

a. True

b. False

Answer: a

Quiz

Q

Which one of the following statements best describes the difference between `CASE` and searched `CASE`?

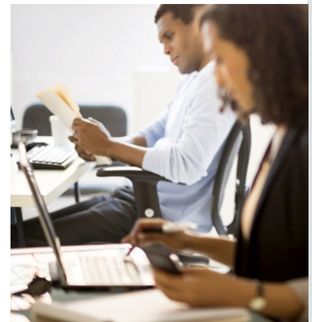
a. `CASE` is used for character searches while searched `CASE` is used for other data types.

b. `CASE` compares the *`expr`* with *`comparison_expr`* while searched `CASE` evaluates a `condition` for each `WHEN` option.

Answer: b

Summary

In this lesson, you should have learned how to use `IF-THEN-ELSE` logic and other conditional expressions in a `SELECT` statement.

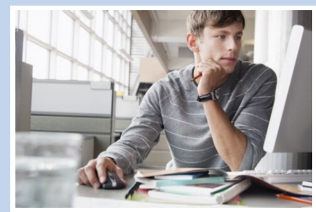


Remember the following:

- The `IF-THEN-ELSE` logic can be applied within a SQL statement by using the `CASE` expression, searched `CASE`, or the `DECODE` function.

Practice 9: Overview

This practice covers creating queries that use conditional expressions such as `CASE`, searched `CASE`, and `DECODE`.



This practice provides exercises on conditional expressions such as `CASE`, searched `CASE`, and `DECODE`.


Lesson 10: Reporting Aggregated Data Using the Group Functions
ORACLE
Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Course Roadmap

You are here

- Lesson 1: Course Overview
- Unit 1: Relational Database and SQL Overview
- Unit 2: Retrieving and Sorting Data
- Unit 3: Joins, Subqueries, and Set Operators
  - Lesson 10: Reporting Aggregated Data Using the Group Functions
  - Lesson 11: Retrieving Data from Multiple Tables Using Joins
  - Lesson 12: Using the Set Operators
  - Lesson 13: Using Subqueries to Solve Queries
- Unit 4: DML and DDL
- Unit 5: Managing Relational Database
- Unit 6: Advance Queries and Database Management System

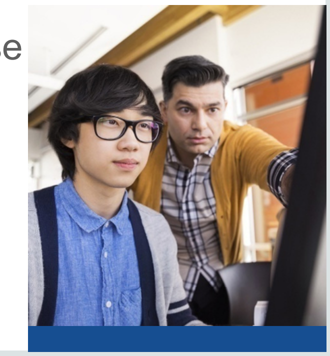
ORACLE

In Unit 3, you will learn about using joins, subqueries, and set operators. You will learn to write compound queries in SQL to generate customized reports using group functions, joins, and subqueries.

Objectives

After completing this lesson, you should be able to:

- Define Aggregation
- Identify the available group functions
- Describe the use of group functions
- Group data by using the `GROUP BY` clause
- Include or exclude grouped rows by using the `HAVING` clause



ORACLE

This lesson further addresses functions. It focuses on obtaining summary information (such as averages) for groups of rows. It discusses how to group rows in a table into smaller sets and how to specify search criteria for groups of rows.

Lesson Agenda

- What is Data Aggregation?
- Group functions:
  – Types and syntax
  – Use AVG, SUM, MIN, MAX, COUNT
  – Use the DISTINCT keyword within group functions
  – NULL values in a group function
- Grouping rows:
  – GROUP BY clause
  – HAVING clause
- Nesting group functions

ORACLE

What Is Data Aggregation?

- Process of computing data to present it in summary form for statistical analysis
- Used to get more information about particular groups based on specific variables, for example, age, job, or salary

ORACLE

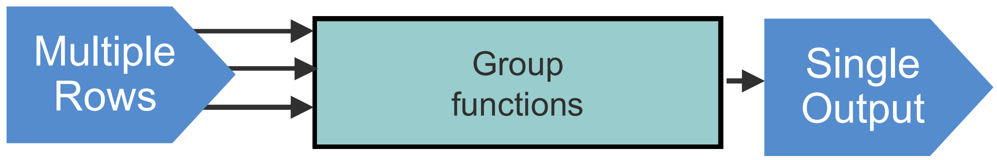
Data Aggregation is the process of computing data to present it in summary form for statistical analysis. It is the process of compiling data for specific groups based on variables such as age, job, or salary. The information about such groups can then be used for further processing or decision-making. For example, you may want to know what is the average salary of employees for each department. A college may want to collect information regarding the top scoring students, subject-wise.

Lesson Agenda

- What is Data Aggregation?
- Group functions:
  - Types and syntax
  - Use AVG, SUM, MIN, MAX, COUNT
  - Use the DISTINCT keyword within group functions
  - NULL values in a group function
- Grouping rows:
  - GROUP BY clause
  - HAVING clause
- Nesting group functions

ORACLE®

Types of Group Functions



Unlike single-row functions, group functions operate on sets of rows to give one result per group. These sets may comprise the entire table or the table that is split into groups.

Note: For a complete list of the group functions, see *Oracle Database SQL Language Reference* for 12*c* database.

Group Functions

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

EMPLOYEES

	DEPARTMENT_ID	SALARY
1	10	4400
2	20	13000
3	20	6000
4	110	12000
5	110	8300
6	90	24000
7	90	17000
8	90	17000
9	60	9000
10	60	6000
. . .		
18	80	11000
19	80	8600
20	(null)	7000

Maximum salary in EMPLOYEES table

MAX(SALARY)
24000

ORACLE

For example, if you want to know the employee who gets the maximum salary, you can use the MAX(salary) group function in your SELECT query. You can also compute the MAX(salary) for employees belonging to each department or for each job role.

Group Functions: Syntax

The group functions are placed after the SELECT keyword.

```
SELECT group_function([DISTINCT|ALL]expr)
FROM       table
[WHERE     condition];
```

DISTINCT makes the function compute only nonduplicate values.

ALL includes the duplicates.

- All group functions ignore null values.
- Multiple group functions can be used separated by commas.

ORACLE

The group function is placed after the `SELECT` keyword. You may have multiple group functions separated by commas. The data types for the functions with an `expr` argument may be `CHAR`, `VARCHAR2`, `NUMBER`, or `DATE`. All group functions ignore null values. To substitute a value for null values, use the `NVL`, `NVL2`, `COALESCE`, `CASE`, or `DECODE` functions, discussed previously in the course.

Common Group Functions

Function	Description
AVG([DISTINCT|ALL]n)	Average value of n, ignoring null values
COUNT([DISTINCT|ALL]n)	Number of rows (count all selected rows using *, including duplicates and rows with nulls)
MAX([DISTINCT|ALL]*expr*)	Maximum value of *expr*, ignoring null values
MIN([DISTINCT|ALL]*expr*)	Minimum value of *expr*, ignoring null values
SUM([DISTINCT|ALL]n)	Sum values of *n*, ignoring null values

ORACLE

Every function accepts an argument. The table in the slide identifies the options that you can use in the syntax of some of the common group functions.

Guideline for using the group functions: `DISTINCT` makes the function consider only nonduplicate values; `ALL` makes it consider every value, including duplicates. The default is `ALL`.

Using the AVG and SUM Functions

You can use AVG and SUM for numeric data.

```
SELECT  AVG(salary), MAX(salary),
        MIN(salary), SUM(salary)
FROM    employees
WHERE   job_id LIKE '%REP%';
```

	AVG(SALARY)	MAX(SALARY)	MIN(SALARY)	SUM(SALARY)
1	8150	11000	6000	32600

ORACLE

You can use the AVG, SUM, MIN, and MAX functions against the columns that store numeric data. The example in the slide displays the average salary, highest salary, lowest salary, and sum of monthly salaries for all sales representatives.

Using the MIN and MAX Functions

You can use MIN and MAX for numeric, character, and date data types.

```
SELECT MIN(hire_date), MAX(hire_date)
FROM        employees;
```

	MIN(HIRE_DATE)	MAX(HIRE_DATE)
1	13-JAN-09	29-JAN-16

Displays the most junior and the most senior employees

You can use the MAX and MIN functions for numeric, character, and date data types. The example in the slide displays the most junior and most senior employees.

The following example displays the employee last name that is first and the employee last name that is last in an alphabetic list of all employees:

```
SELECT MIN(last_name), MAX(last_name)
FROM   employees;
```

Note: The AVG and SUM functions can be used only with numeric data types.

Using the `COUNT` Function

`COUNT(*)` returns the number of rows in a table:

1

```
SELECT COUNT(*)
FROM   employees
WHERE  department_id = 50;
```

	COUNT(*)
1	5

`COUNT(expr)` returns the number of rows with non-null values for *`expr`*:

2

```
SELECT COUNT(commission_pct)
FROM   employees
WHERE  department_id = 50;
```

	COUNT(COMMISSION_PCT)
1	0

The `COUNT` function has three formats:

- `COUNT(*)`
- `COUNT(`*`expr`*`)`
- `COUNT(DISTINCT` *`expr`*`)`

`COUNT(*)` returns the number of rows in a table that satisfy the criteria of the `SELECT` statement, including duplicate rows and rows containing null values in any of the columns. If a `WHERE` clause is included in the `SELECT` statement, `COUNT(*)` returns the number of rows that satisfy the condition in the `WHERE` clause.

In contrast, `COUNT(`*`expr`*`)` returns the number of non-null values that are in the column identified by *`expr`*.

Examples

1. The first example in the slide displays the number of employees in department 50.
2. The second example in the slide displays the number of employees in department 50 who can earn a commission.

Using DISTINCT in COUNT function

- `COUNT(DISTINCT expr)` returns the number of distinct non-null values of *expr*.
- To display the number of distinct department values in the `EMPLOYEES` table:

```
SELECT COUNT(DISTINCT department_id)
FROM   employees;
```

	COUNT(DISTINCTDEPARTMENT_ID)
1	7

ORACLE

`COUNT(DISTINCT expr)` returns the number of unique, non-null values that are in the column identified by *expr*.

Use the `DISTINCT` keyword to suppress the counting of any duplicate values in a column. The example in the slide displays the number of distinct department values that are in the `EMPLOYEES` table.

Group Functions and Null Values

Group functions ignore null values in the column:

1

```
SELECT AVG(commission_pct)
FROM   employees;
```

	AVG(COMMISSION_PCT)
1	0.2125

The NVL function forces group functions to include null values:

2

```
SELECT AVG(NVL(commission_pct, 0))
FROM   employees;
```

	AVG(NVL(COMMISSION_PCT,0))
1	0.0425

ORACLE

All group functions ignore null values in the column.

However, the NVL function forces group functions to include null values. The null values are substituted by the value zero.

Examples

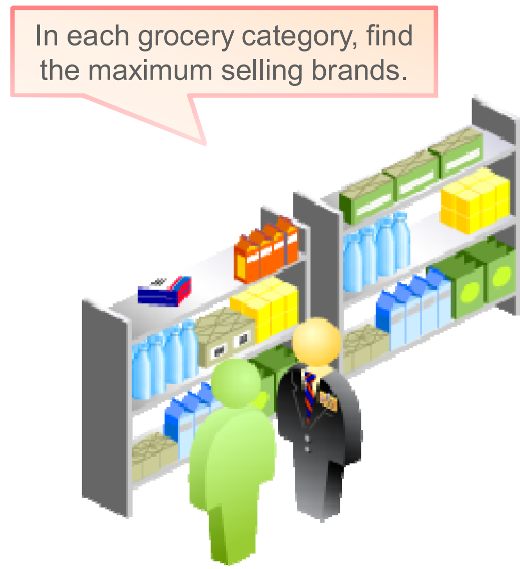
1. The average is calculated based on *only* those rows in the table in which a valid value is stored in the COMMISSION_PCT column. The average is calculated as the total commission that is paid to all employees divided by the number of employees receiving a commission (four).
2. The average is calculated based on *all* rows in the table, regardless of whether null values are stored in the COMMISSION_PCT column. The average is calculated as the total commission that is paid to all employees divided by the total number of employees in the company (20).

Lesson Agenda

- What is Data Aggregation?
- Group functions:
  - Types and syntax
  - Use `AVG`, `SUM`, `MIN`, `MAX`, `COUNT`
  - Use `DISTINCT` keyword within group functions
  - `NULL` values in a group function
- Grouping rows:
  - `GROUP BY` clause
  - `HAVING` clause
- Nesting group functions

Creating Groups of Data

- Consider the following queries:
  - Find the average salary for each department (**Note:** not the average salary of all the employees).
  - Compute the sum of salaries of all the employees with the same job in each department.
  - Find the oldest employee in each job role.



ORACLE

Many times you want to aggregate data based on groups and not on the whole data set. The queries on the slide are some of the examples. So, you want to know the average salary for each department, and not the average salary of all the employees. Similarly, in a grocery store, the store manager may want to know which is the best-selling brand of cookies, detergent, and milk products.

Creating Groups of Data

EMPLOYEES

	DEPARTMENT_ID	SALARY	
1	10	4400	4400
2	20	13000	
3	20	6000	9500
4	50	2500	
5	50	2600	
6	50	3100	3500
7	50	3500	
8	50	5800	
9	60	9000	
10	60	6000	6400
11	60	4200	
12	80	11000	
13	80	8600	10033
18	110	8300	
19	110	12000	
20	(null)	7000	

Average salary in the EMPLOYEES table for each department

	DEPARTMENT_ID	AVG(SALARY)
1	(null)	7000
2	20	9500
3	90	19333.333333333333...
4	110	10150
5	50	3500
6	80	10033.333333333333...
7	10	4400
8	60	6400

Until this point in the discussion, all group functions have treated the table as one large group of information. At times, however, you need to divide the table of information into smaller groups. This can be done by using the `GROUP BY` clause.

Creating Groups of Data: GROUP BY Clause Syntax

You can divide rows in a table into smaller groups by using the GROUP BY clause.

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

Specifies the columns whose values determine the grouping

ORACLE

You can use the GROUP BY clause to divide the rows in a table into groups. You can then use the group functions to return summary information for each group.

In the syntax:

group_by_expression Specifies the columns whose values determine the basis for grouping rows

Guidelines

- If you include a group function in a SELECT clause, you cannot select the individual column as well, *unless* the individual column appears in the GROUP BY clause. You receive an error message if you fail to include the column list in the GROUP BY clause.
- Using a WHERE clause, you can exclude rows before dividing them into groups.
- You can substitute *column* with an expression in the SELECT statement.
- You must include the *columns* in the GROUP BY clause.
- You cannot use a column alias in the GROUP BY clause.

Using the GROUP BY Clause

All the columns in the SELECT list that are not in group functions must be in the GROUP BY clause.

```
SELECT   department_id, AVG(salary)
FROM     employees
GROUP BY department_id ;
```

	DEPARTMENT_ID	AVG(SALARY)
1	(null)	7000
2	90	19333.3333333333333333333333333333333333
3	20	9500
4	110	10154
5	50	3500
6	80	10033.3333333333333333333333333333333333
7	60	6400
8	10	4400

When using the GROUP BY clause, make sure that all columns in the SELECT list that are not group functions are included in the GROUP BY clause. The example in the slide displays the department number and the average salary for each department. Here is how this SELECT statement, containing a GROUP BY clause, is evaluated:

- The SELECT clause specifies the columns to be retrieved, as follows:
  - Department number column in the EMPLOYEES table
  - The average of all salaries in the group that you specified in the GROUP BY clause
- The FROM clause specifies the tables that the database must access: the EMPLOYEES table.
- The WHERE clause specifies the rows to be retrieved. Because there is no WHERE clause, all rows are retrieved by default.
- The GROUP BY clause specifies how the rows should be grouped. The rows are grouped by department number, so the AVG function that is applied to the salary column calculates the average salary for each department.

Note: To order the query results in ascending or descending order, include the ORDER BY clause in the query.

Using the GROUP BY Clause

The GROUP BY column does not have to be in the SELECT list.

```
SELECT   AVG(salary)
FROM     employees
GROUP BY department_id ;
```

	AVG(SALARY)
1	7000
2	19333.33333333333333333333333333333333333
3	9500
4	10154
5	3500
6	10033.33333333333333333333333333333333333
7	6400
8	4400

Without the department numbers, the results are not meaningful.

ORACLE

The GROUP BY column does not have to be in the SELECT clause. For example, the SELECT statement in the slide displays the average salaries for each department without displaying the respective department numbers. Without the department numbers, however, the results do not look meaningful.

You can also use the group function in the ORDER BY clause:

```
SELECT   department_id, AVG(salary)
FROM     employees
GROUP BY department_id
ORDER BY AVG(salary);
```

Grouping by More Than One Column

EMPLOYEES

	DEPARTMENT_ID	JOB_ID	SALARY
1	10	AD_ASST	4400
2	20	MK_MAN	13000
3	20	MK_REP	6000
4	50	ST_CLERK	2500
5	50	ST_CLERK	2600
6	50	ST_CLERK	3100
7	50	ST_CLERK	3500
8	50	ST_MAN	5800
9	60	IT_PROG	9000
10	60	IT_PROG	6000
11	60	IT_PROG	4200
12	80	SA_REP	11000
13	80	SA_REP	8600
14	80	SA_MAN	10500
. . .			
19	110	AC_MGR	12000
20	(null)	SA_REP	7000

Add the salaries in the EMPLOYEES table for each job, grouped by department.

	DEPARTMENT_ID	JOB_ID	SUM(SALARY)
1	110	AC_ACCOUNT	8300
2	110	AC_MGR	12008
3	10	AD_ASST	4400
4	90	AD_PRES	24000
5	90	AD_VP	34000
6	60	IT_PROG	19200
7	20	MK_MAN	13000
8	20	MK_REP	6000
9	80	SA_MAN	10500
10	80	SA_REP	19600
11	(null)	SA_REP	7000
12	50	ST_CLERK	11700
13	50	ST_MAN	5800

Sometimes, you need to see results for groups within groups. The slide shows a report that displays the total salary that is paid to each job title in each department.

The EMPLOYEES table is grouped first by the department number, and then by the job title within that grouping. For example, the four stock clerks in department 50 are grouped together, and a single result (total salary) is produced for all stock clerks in the group.

The following SELECT statement returns the result shown in the slide:

```
SELECT   department_id, job_id, sum(salary)
FROM     employees
GROUP BY department_id, job_id
ORDER BY job_id;
```

Using the GROUP BY Clause on Multiple Columns

```
SELECT   department_id, job_id, SUM(salary)
FROM     employees
WHERE     department_id > 40
GROUP BY department_id, job_id
ORDER BY department_id;
```

The SUM function is applied to the salary column for all job IDs in the result set in each DEPARTMENT_ID group.

	DEPARTMENT_ID	JOB_ID	SUM(SALARY)
1	50	ST_CLERK	11700
2	50	ST_MAN	5800
3	60	IT_PROG	19200
4	80	SA_MAN	10500
5	80	SA_REP	19600
6	90	AD_PRES	24000
7	90	AD_VP	34000
8	110	AC_ACCOUNT	8300
9	110	AC_MGR	12008

ORACLE

You can return summary results for groups and subgroups by listing multiple GROUP BY columns. The GROUP BY clause groups rows but does not guarantee the order of the result set. To order the groupings, use the ORDER BY clause.

In the example in the slide, the SELECT statement that contains a GROUP BY clause is evaluated as follows:

- The SELECT clause specifies the column to be retrieved:
  - DEPARTMENT_ID in the EMPLOYEES table
  - JOB_ID in the EMPLOYEES table
  - The sum of all salaries in the group that you specified in the GROUP BY clause
- The FROM clause specifies the tables that the database must access: the EMPLOYEES table.
- The WHERE clause reduces the result set to those rows where DEPARTMENT_ID is greater than 40.
- The GROUP BY clause specifies how you must group the resulting rows:
  - First, the rows are grouped by the DEPARTMENT_ID.
  - Second, the rows are grouped by JOB ID in the DEPARTMENT_ID groups.
- The ORDER BY clause sorts the results by DEPARTMENT_ID.

Note: The SUM function is applied to the salary column for all job IDs in the result set in each DEPARTMENT_ID group.

Common Errors: Using Group Functions

Any column or expression in the SELECT list that is not an aggregate function must be in the GROUP BY clause:

```
SELECT department_id, COUNT(last_name)
FROM   employees;
```

```
ORA-00937: not a single-group group function
00937. 00000 - "not a single-group group function"
```

A GROUP BY clause must be added to count the last names for each department_id.

Correct SQL statement

```
SELECT    department_id, count(last_name)
FROM      employees
GROUP BY  department_id;
```


Whenever you use a mixture of individual items (DEPARTMENT_ID) and group functions (COUNT) in the same SELECT statement, you must include a GROUP BY clause that specifies the individual items (in this case, DEPARTMENT_ID). If the GROUP BY clause is missing, the error message "not a single-group group function" appears and an asterisk (*) points to the offending column. You can correct the error in the first example in the slide by adding the GROUP BY clause:

```
SELECT    department_id, count(last_name)
FROM      employees
GROUP BY department_id;
```

Common Errors: Using Group Functions

Any column or expression in the SELECT list that is not an aggregate function must be in the GROUP BY clause:

```
SELECT department_id, job_id, COUNT(last_name)
FROM   employees
GROUP BY department_id;
```

```
ORA-00979: not a GROUP BY expression
00979. 00000 - "not a GROUP BY expression"
```

Correct SQL statement

Either add job_id in the GROUP BY clause or remove the job_id column from the SELECT list.

```
SELECT department_id, job_id, COUNT(last_name)
FROM   employees
GROUP BY department_id, job_id;
```

ORACLE

Any column or expression in the SELECT list that is not an aggregate function must be in the GROUP BY clause. In the second example in the slide, JOB_ID is neither in the GROUP BY clause nor is it being used by a group function, so there is a "not a GROUP BY expression" error. You can correct the error in the second slide example by adding JOB_ID in the GROUP BY clause.

```
SELECT department_id, job_id, COUNT(last_name)
FROM   employees
GROUP BY department_id, job_id;
```

Common Errors: Using Group Functions

- You cannot use the `WHERE` clause to restrict groups.
- You cannot use group functions in the `WHERE` clause.

```
SELECT    department_id, AVG(salary)
FROM      employees
WHERE     AVG(salary) > 8000
GROUP BY department_id;
```

```
ORA-00934: group function is not allowed here
00934. 00000 - "group function is not allowed here"
*Cause:
*Action:
Error at Line: 3 Column: 9
```

With `GROUP BY`, use the `HAVING` clause to restrict groups.

(discussed in the next slide)

The `WHERE` clause cannot be used to restrict groups. The `SELECT` statement in the example in the slide results in an error because it uses the `WHERE` clause to restrict the display of the average salaries of those departments that have an average salary greater than $8,000.

Restricting Group Results: Using the `HAVING` Clause

EMPLOYEES

	DEPARTMENT_ID	SALARY
1	10	4400
2	20	13000
3	20	6000
4	50	2500
5	50	2600
6	50	3100
7	50	3500
8	50	5800
9	60	9000
10	60	6000
11	60	4200
12	80	11000
13	80	8600
...		
18	110	8300
19	110	12000
20	(null)	7000

You use the `HAVING` clause to restrict groups in the same way that you use the `WHERE` clause to restrict the rows that you select.

	DEPARTMENT_ID	MAX(SALARY)
1	20	13000
2	90	24000
3	110	12000
4	80	11000

The maximum salary per department when it is greater than $10,000

ORACLE

You use the `HAVING` clause to restrict groups in the same way that you use the `WHERE` clause to restrict the rows that you select. To find the maximum salary in each of the departments that have a maximum salary greater than $10,000, you need to do the following:

1. Find the average salary for each department by grouping by department number.
2. Restrict the groups to those departments with a maximum salary greater than $10,000.

Restricting Group Results with the HAVING Clause

When you use the HAVING clause, the Oracle server restricts groups as follows:

1. Rows are grouped.
2. The group function is applied.
3. Groups matching the HAVING clause are displayed.

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

ORACLE

You use the HAVING clause to specify the groups that are to be displayed, thus further restricting the groups on the basis of aggregate information.

In the syntax, *group_condition* restricts the groups of rows returned to those groups for which the specified condition is true.

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

1. Rows are grouped.
2. The group function is applied to the group.
3. The groups that match the criteria in the HAVING clause are displayed.

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. Groups are formed and group functions are calculated before the HAVING clause is applied to the groups in the SELECT list.

Note: The WHERE clause restricts rows, whereas the HAVING clause restricts groups.

Using the HAVING Clause

```
SELECT    department_id, MAX(salary)
FROM      employees
GROUP BY  department_id
HAVING    MAX(salary)>10000 ;
```

	DEPARTMENT_ID	MAX(SALARY)
1	90	24000
2	20	13000
3	110	12008
4	80	11000

Query Processing Steps:
1. Group the rows on department_ids.
2. Calculate the MAX(Salary) for each department.
3. Display only those rows that match the HAVING condition, that is, departments with a maximum salary greater than $10,000.

ORACLE

The example in the slide displays the department numbers and maximum salaries for those departments with a maximum salary greater than $10,000.

You can use the GROUP BY clause without using a group function in the SELECT list. If you restrict rows based on the result of a group function, you must have a GROUP BY clause as well as the HAVING clause.

The following example displays the department numbers and average salaries for those departments with a maximum salary greater than $10,000:

```
SELECT    department_id, AVG(salary)
FROM      employees
GROUP BY  department_id
HAVING    max(salary)>10000;
```

Using the HAVING Clause

```
SELECT   job_id, SUM(salary) PAYROLL
FROM     employees
WHERE    job_id NOT LIKE '%REP%'
GROUP BY job_id
HAVING   SUM(salary) > 13000
ORDER BY SUM(salary);
```

	JOB_ID	PAYROLL
1	IT_PROG	19200
2	AD_PRES	24000
3	AD_VP	34000

Query Processing Steps:

1. Eliminate rows with job_id as SALES_REP.
2. Group the selected rows on job_ids.
3. Calculate the SUM(Salary) for each job_id.
4. Select rows with sum of salary greater than $13,000.
5. Display the rows in ascending order of the total monthly salary.

ORACLE

The example in the slide displays the JOB_ID and total monthly salary for each job that has a total payroll exceeding $13,000. The example excludes sales representatives and sorts the list by the total monthly salary.

Lesson Agenda

- What is Data Aggregation?
- Group functions:
  – Types and syntax
  – Use AVG, SUM, MIN, MAX, COUNT
  – Use DISTINCT keyword within group functions
  – NULL values in a group function
- Grouping rows:
  – GROUP BY clause
  – HAVING clause
- Nesting group functions

Nesting Group Functions

- Group functions can be nested to a depth of two functions.
- The GROUP BY clause is mandatory when nesting group functions.

Display the maximum average salary.

```
SELECT    MAX(AVG(salary))
FROM      employees
GROUP BY  department_id;
```

	MAX(AVG(SALARY))
1	19333.3333333333333333333333333333333333

Group functions can be nested to a depth of two functions. The example in the slide calculates the average salary for each DEPARTMENT_ID and then displays the maximum average salary.

Note that the GROUP BY clause is mandatory when nesting group functions. When you specify GROUP BY on DEPARTMENT_ID, the AVG(salary) is computed for each department and then the MAX functions returns the maximum average salary.

Quiz

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

a. True
b. False

Answer: a

Quiz

Group functions process null values in the column.

a. True
b. False

Answer: b

Quiz

Which one of the following clauses can you use to divide rows in a table into smaller data sets?

a. `WHERE` clause

b. `ORDER BY` clause

c. `GROUP BY` clause

d. `HAVING` clause

Answer: c

Quiz

Identify the three guidelines for the GROUP BY clause.

a. You cannot use group functions with SELECT queries having the GROUP BY clause.
b. The GROUP BY column should be in the SELECT clause.
c. All the columns in the SELECT list that are not in group functions must be in the GROUP BY clause.
d. You cannot use the WHERE clause to restrict groups.

ORACLE

Answer: b,c,d

Quiz

Q

Which one of the following clauses do you use to restrict groups formed by the `GROUP BY` clause?

a. `WHERE` clause

b. `HAVING` clause

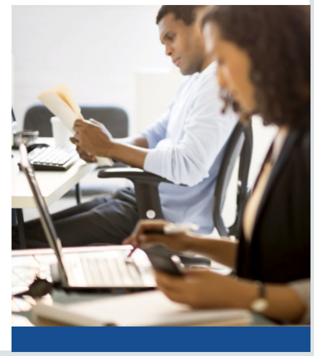
Answer: b

Summary

In this lesson, you should have learned how to:

- Use the COUNT, MAX, MIN, SUM, and AVG group functions
- Write queries that use the GROUP BY clause
- Write queries that use the HAVING clause

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



ORACLE

There are several group functions available in SQL, such as AVG, COUNT, MAX, MIN, and SUM.

You can create subgroups by using the GROUP BY clause. Further, groups can be restricted by using the HAVING clause.

Place the HAVING and GROUP BY clauses after the WHERE clause in a statement. The order of the GROUP BY and HAVING clauses following the WHERE clause is not important. You can have either the GROUP BY clause or the HAVING clause first, as long as they follow the WHERE clause. Place the ORDER BY clause at the end.

The Oracle server evaluates the clauses in the following order:

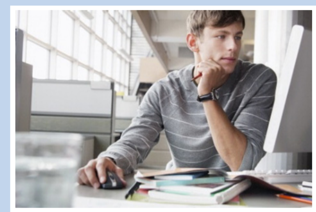
1. If the statement contains a WHERE clause, the server establishes the candidate rows.
2. The server identifies the groups that are specified in the GROUP BY clause.
3. The HAVING clause further restricts result groups that do not meet the group criteria in the HAVING clause.

Note: For a complete list of the group functions, see *Oracle Database SQL Language Reference* for 12*c* database.

Practice 10: Overview

This practice covers the following topics:

- Writing queries that use group functions
- Grouping by rows to achieve more than one result
- Restricting groups by using the `HAVING` clause

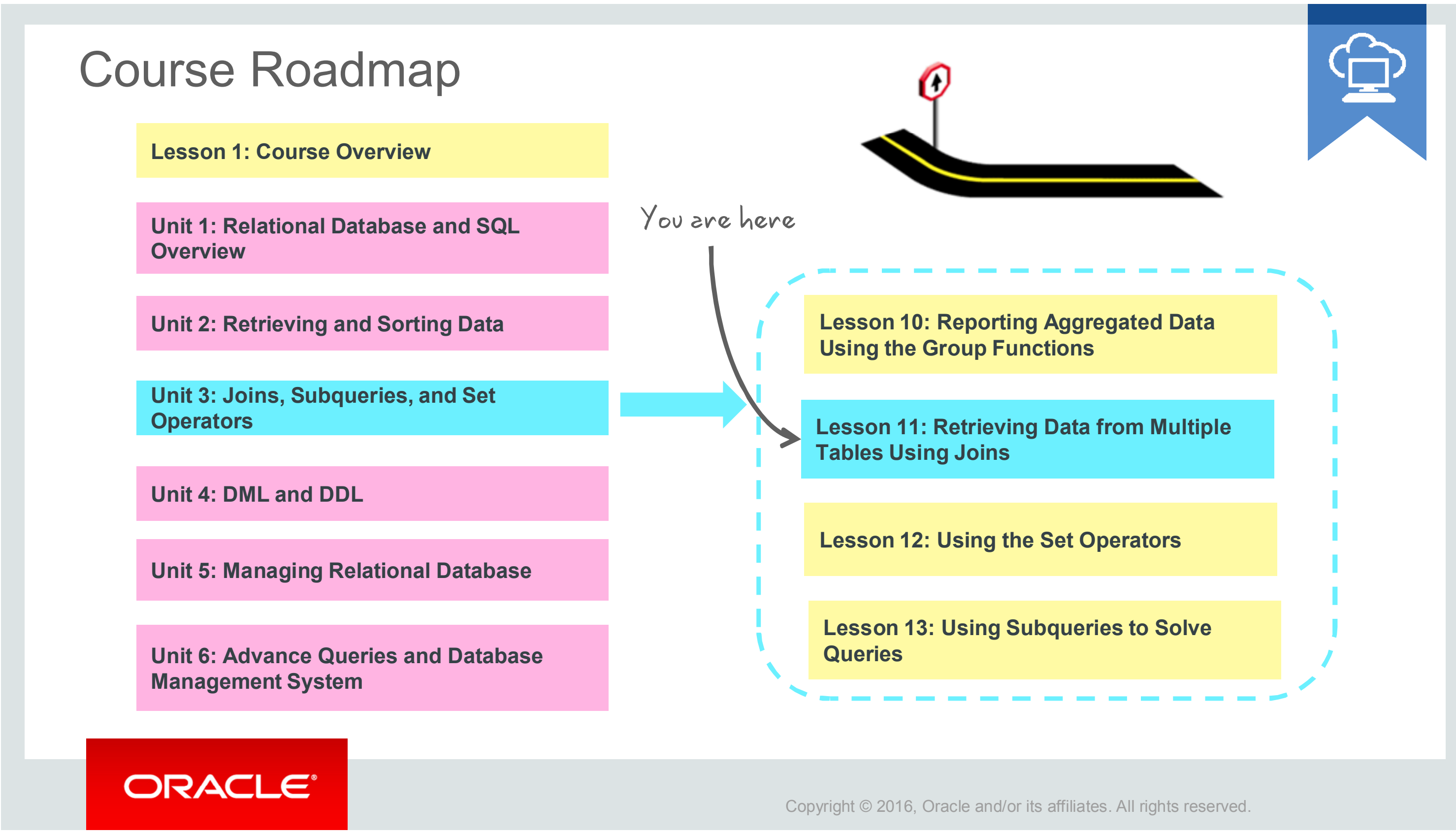


ORACLE

In this practice, you learn to use group functions and select groups of data.

Lesson 11: Retrieving Data from Multiple Tables Using Joins

ORACLE

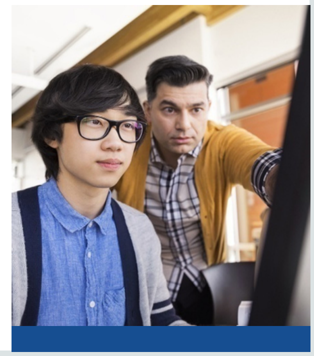


In Unit 3, you will learn about using joins, subqueries, and set operators. You will learn to write compound queries in SQL to generate customized reports using group functions, joins, and subqueries.`

Objectives

After completing this lesson, you should be able to:

- Write SELECT statements to access data from more than one table using equijoins and nonequijoins
- Join a table to itself by using a self-join
- View data that generally does not meet a join condition by using OUTER joins
- Generate a Cartesian product of all rows from two or more tables



ORACLE

This lesson explains how to obtain data from more than one table. A *join* is used to view information from multiple tables. Therefore, you can *join* tables together to view information from more than one table.

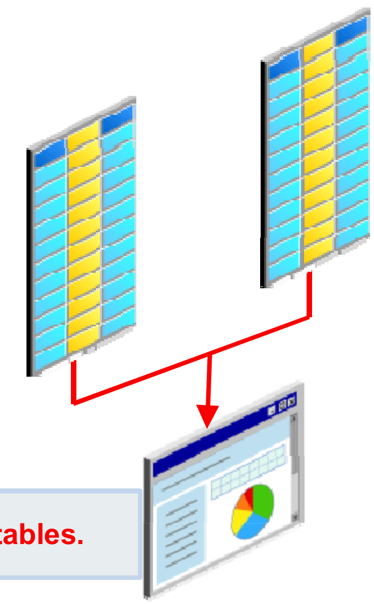
Note: Information about joins is found in the "SQL Queries and Subqueries: Joins" section in *Oracle Database SQL Language Reference* for 12*c* database.

Lesson Agenda

- Types of `JOINS` and their syntax
  - Natural join
  - Join with the `USING` clause
  - Join with the `ON` clause
- Self-join
- Nonequijoins
- `OUTER` joins
- Cartesian product
  - Cross join

Why Join?

- Different information is stored in different tables.
- Tables are linked to each other through common attributes.
- Using the common attributes, data from multiple tables can be retrieved.



Use Join with `SELECT` queries to retrieve data from multiple tables.

ORACLE

In a relational database, the data in normalized form is stored in different tables. You may need to get information from multiple tables in one report. The tables are normally linked to each other through common attributes. Using these common attributes or columns, you can join the tables and display the required information. View the next slide for an example.

Obtaining Data from Multiple Tables

EMPLOYEES

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	JOB_ID
1	100	Steven	King	AD_PRES
2	101	Neena	Kochhar	AD_VP
3	102	Lex	De Haan	AD_VP
4	103	Alexander	Hunold	IT_PROG
5	104	Bruce	Ernst	IT_PROG
6	105	David	Austin	IT_PROG
7	106	Valli	Pataballa	IT_PROG
8	107	Diana	Lorentz	IT_PROG
9	108	Nancy	Greenberg	FI_MGR
10	109	Daniel	Faviet	FI_ACCOUNT

JOBS

	JOB_ID	JOB_TITLE
1	AD_PRES	President
2	AD_VP	Administration Vice President
3	AD_ASST	Administration Assistant
4	FI_MGR	Finance Manager
5	FI_ACCOUNT	Accountant
6	AC_MGR	Accounting Manager
7	AC_ACCOUNT	Public Accountant
8	SA_MAN	Sales Manager
9	SA_REP	Sales Representative

	EMPLOYEE_ID	JOB_ID	JOB_TITLE
1	206	AC_ACCOUNT	Public Accountant
2	205	AC_MGR	Accounting Manager
3	200	AD_ASST	Administration Assistant
4	100	AD_PRES	President
5	101	AD_VP	Administration Vice President
6	102	AD_VP	Administration Vice President
7	109	FI_ACCOUNT	Accountant

. . .

ORACLE

Sometimes you need to use data from more than one table. In the example in the slide, the report displays data from two separate tables:

- Employees IDs exist in the EMPLOYEES table.
- Job IDs exist in both the EMPLOYEES and JOBS tables.
- Job titles exist in the JOBS table.

To produce the report, you need to link the EMPLOYEES table and the JOBS table, and access data from both of them.

Types of Joins

Joins that are compliant with the SQL:1999 standard include the following:

- Natural join with the `NATURAL JOIN` clause
- Join with the `USING` clause
- Join with the `ON` clause
- `OUTER` joins:
  - `LEFT OUTER JOIN`
  - `RIGHT OUTER JOIN`
  - `FULL OUTER JOIN`
- Cross joins

ORACLE

To join tables, you can use a join syntax that is compliant with the SQL:1999 standard.

Note

- Before the Oracle9*i* release, the Oracle join syntax was different from the American National Standards Institute (ANSI) standards. The SQL:1999–compliant join syntax does not offer any performance benefits over the Oracle-proprietary join syntax that existed in the prior releases.
- The following slide discusses the SQL:1999 join syntax.

Joining Tables Using the SQL:1999 Syntax

Use a join to query data from more than one table:

table1.column denotes the table and the column from which data is retrieved.

```
SELECT   table1.column, table2.column
FROM     table1
[NATURAL JOIN table2] |
[JOIN table2 USING (column_name)] |
[JOIN table2 ON (table1.column_name = table2.column_name)]|
[LEFT|RIGHT|FULL OUTER JOIN table2
 ON (table1.column_name = table2.column_name)]|
[CROSS JOIN table2];
```

Inner equijoins

Cross join or Cartesian product

Outer joins: LEFT, RIGHT, FULL

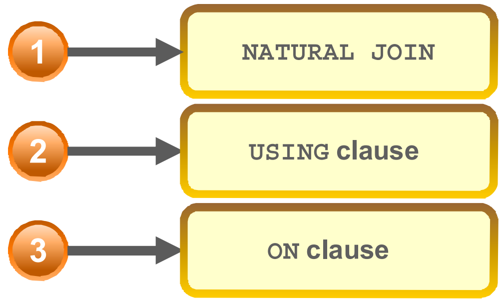
In the syntax:

- `table1.column` denotes the table and the column from which data is retrieved
- `NATURAL JOIN` joins two tables based on the same column name
- `JOIN table2 USING column_name` performs an equijoin based on the column name
- `JOIN table2 ON table1.column_name = table2.column_name performs` an equijoin based on the condition in the `ON` clause
- `LEFT/RIGHT/FULL OUTER` is used to perform `OUTER` joins
- `CROSS JOIN` returns a Cartesian product from the two tables

For more information, see the section titled "`SELECT`" in *Oracle Database SQL Language Reference* for 12*c* database.

Inner Joins

- In SQL:1999, the join of two tables returning only matched rows is called an `INNER` join.



Joining tables with the `NATURAL JOIN`, `USING`, or `ON` clauses results in an `INNER` join. Only matched rows are returned in an inner join.

Creating Natural Joins

- The NATURAL JOIN clause is based on all the columns that have the same name in two tables.
- It selects rows from the two tables that have equal values in all matched columns.
- If the columns having the same names have different data types, an error is returned.

```
SELECT * FROM table1 NATURAL JOIN table2;
```

1

ORACLE

You can join tables automatically based on the columns in the two tables that have matching data types and names. You do this by using the NATURAL JOIN keywords.

Note: The join can happen on only those columns that have the same names and data types in both tables. If the columns have the same name but different data types, the NATURAL JOIN syntax causes an error.

Retrieving Records with Natural Joins

```
SELECT employee_id, first_name, job_id, job_title
from employees NATURAL JOIN jobs;
```

	EMPLOYEE_ID	FIRST_NAME	JOB_ID	JOB_TITLE
1	100	Steven	AD_PRES	President
2	101	Neena	AD_VP	Administration Vice President
3	102	Lex	AD_VP	Administration Vice President
4	103	Alexander	IT_PROG	Programmer
5	104	Bruce	IT_PROG	Programmer
6	105	David	IT_PROG	Programmer
7	106	Valli	IT_PROG	Programmer
8	107	Diana	IT_PROG	Programmer
9	108	Nancy	FI_MGR	Finance Manager
10	109	Daniel	FI_ACCOUNT	Accountant
11	110	John	FI_ACCOUNT	Accountant

...

The JOBS table is joined to the EMPLOYEES table by the JOB_ID column.

JOB_TITLE from the JOBS table is displayed with the employee details.

ORACLE

In the example in the slide, the JOBS table is joined to the EMPLOYEES table by the JOB_ID column, which is the only column of the same name in both tables. If other common columns were present, the join would have used them all.

Natural Joins with a WHERE Clause

Additional restrictions on a natural join are implemented by using a WHERE clause. The following example limits the rows of output to those with a DEPARTMENT_ID equal to 20 or 50:

```
SELECT   department_id, department_name,
         location_id, city
FROM     departments
NATURAL JOIN locations
WHERE    department_id IN (20, 50);
```

Creating Joins with the USING Clause

- If several columns have the same names but the data types do not match, use the USING clause to *specify* the columns for the equijoin.
- Use the USING clause to match only one column when more than one column matches.

```
SELECT   table1.column, table2.column
FROM     table1
JOIN table2 USING (column_name);
```

2

ORACLE

Natural joins use all columns with matching names and data types to join the tables. The USING clause can be used to specify only those columns that should be used for an equijoin.

Joining Column Names

EMPLOYEES

	EMPLOYEE_ID	DEPARTMENT_ID
1	200	10
2	201	20
3	202	20
4	205	110
5	206	110
6	100	90
7	101	90
8	102	90
9	103	60
10	104	60

. . .

Foreign key

DEPARTMENTS

	DEPARTMENT_ID	DEPARTMENT_NAME
1	10	Administration
2	20	Marketing
3	50	Shipping
4	60	IT
5	80	Sales
6	90	Executive
7	110	Accounting
8	190	Contracting

Primary key

ORACLE

To determine an employee's department name, you compare the value in the `DEPARTMENT_ID` column in the `EMPLOYEES` table with the `DEPARTMENT_ID` values in the `DEPARTMENTS` table. The relationship between the `EMPLOYEES` and `DEPARTMENTS` tables is an *equijoin*; that is, values in the `DEPARTMENT_ID` column in both the tables must be equal. Frequently, this type of join involves primary and foreign key complements.

Retrieving Records with the USING Clause

```
SELECT employee_id, last_name,
       location_id, department_id
FROM   employees JOIN departments
USING (department_id) ;
```

	EMPLOYEE_ID	LAST_NAME	LOCATION_ID	DEPARTMENT_ID
1	200	Whalen	1700	10
2	201	Hartstein	1800	20
3	202	Fay	1800	20
4	144	Vargas	1500	50
5	143	Matos	1500	50
6	142	Davies	1500	50
7	141	Rajs	1500	50
8	124	Mourgos	1500	50
...				
18	206	Gietz	1700	110
19	205	Higgins	1700	110

ORACLE

In the example in the slide, the DEPARTMENT_ID columns in the EMPLOYEES and DEPARTMENTS tables are joined and thus the LOCATION_ID of the department where an employee works is shown.

Qualifying Ambiguous Column Names

- Use table prefixes to qualify column names that are in multiple tables.
- Use table prefixes to increase the speed of parsing of the statement.
- Instead of full table name prefixes, use table aliases.
- A table alias gives a table a shorter name:
  - Keeps SQL code smaller
  - Uses less memory
- Use column aliases to distinguish columns that have identical names, but reside in different tables.

ORACLE

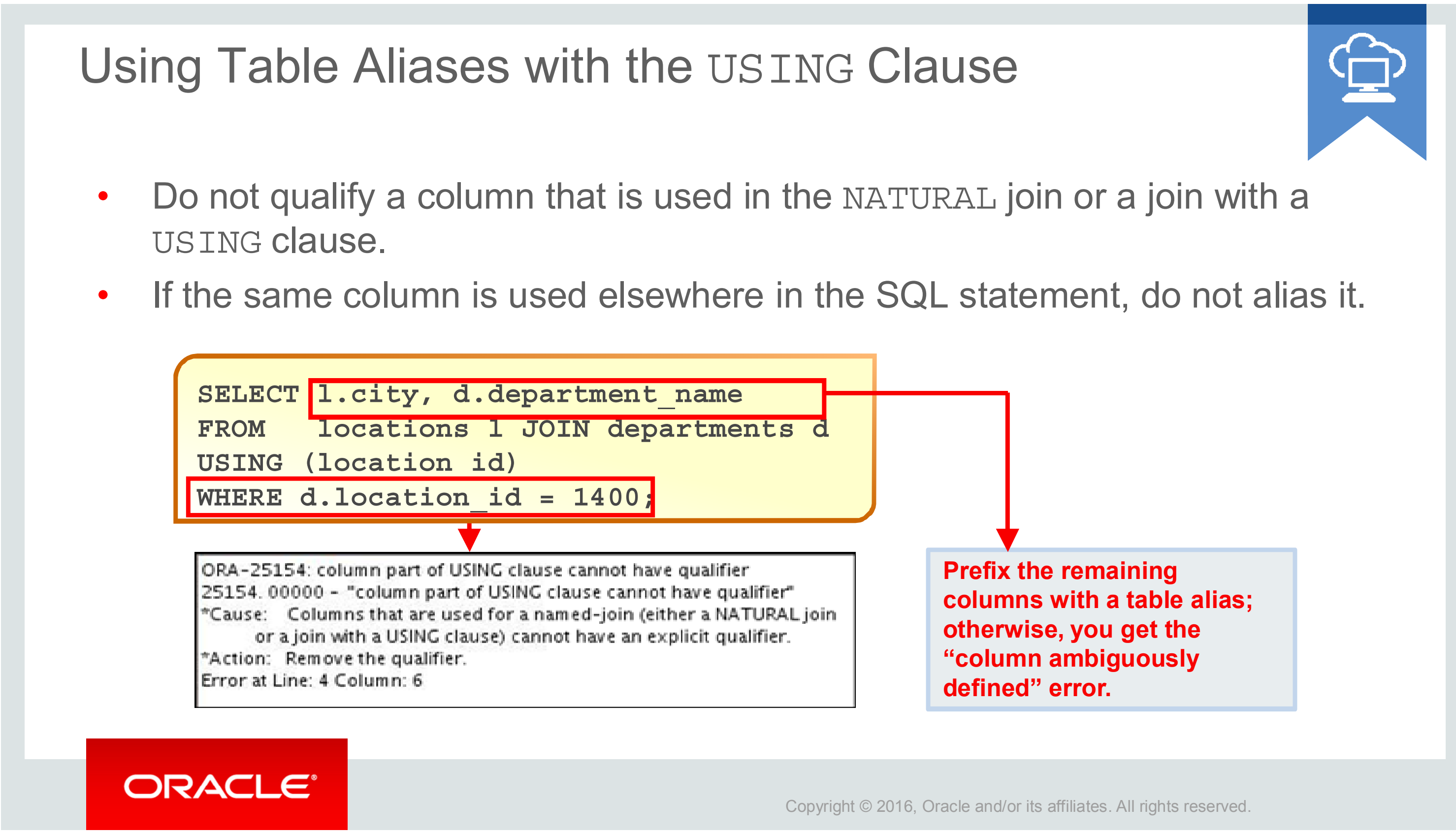
When joining two or more tables, you need to qualify the names of the columns with the table name to avoid ambiguity. Without the table prefixes, the `DEPARTMENT_ID` column in the `SELECT` list could be from either the `DEPARTMENTS` table or the `EMPLOYEES` table. It is necessary to add the table prefix to execute your query. If there are no common column names between the two tables, there is no need to qualify the columns. However, using the table prefix increases the speed of parsing of the statement, because you tell the Oracle server exactly where to find the columns.

However, qualifying column names with table names can be time consuming, particularly if the table names are lengthy. Instead, you can use *table aliases*. Just as a column alias gives a column another name, a table alias gives a table another name. Table aliases help to keep SQL code smaller, therefore, using less memory.

The table name is specified in full, followed by a space, and then the table alias. For example, the `EMPLOYEES` table can be given an alias of `e`, and the `DEPARTMENTS` table an alias of `d`.

Guidelines

- Table aliases can be up to 30 characters in length, but shorter aliases are better than longer ones.
- If a table alias is used for a particular table name in the `FROM` clause, that table alias must be substituted for the table name throughout the `SELECT` statement.
- Table aliases should be meaningful.
- The table alias is valid for only the current `SELECT` statement.



When joining with the USING clause, you cannot qualify a column that is used in the USING clause itself. Furthermore, if that column is used anywhere in the SQL statement, you cannot alias it. For example, in the query mentioned in the slide, you should not alias the location_id column in the WHERE clause because the column is used in the USING clause.

The columns that are referenced in the USING clause should not have a qualifier (table name or alias) anywhere in the SQL statement. For example, the following statement is valid:

```
SELECT l.city, d.department_name
FROM   locations l JOIN departments d USING (location_id)
WHERE  location_id = 1400;
```

The columns that are common in both the tables, but not used in the USING clause, must be prefixed with a table alias; otherwise, you get the "column ambiguously defined" error.

In the following statement, manager_id is present in both the employees and departments table; if manager_id is not prefixed with a table alias, it gives a "column ambiguously defined" error.

The following statement is valid:

```
SELECT first_name, d.department_name, d.manager_id
FROM   employees e JOIN departments d USING (department_id)
WHERE  department_id = 50;
```

Creating Joins with the ON Clause

- The join condition for the natural join is basically an equijoin of all columns with the same name.
- Use the ON clause to specify arbitrary conditions or specify columns to join.
- The join condition is separated from other search conditions.
- The ON clause makes code easy to understand.

```
SELECT   table1.column, table2.column
FROM     table1
JOIN table2 ON (table1.column_name = table2.column_name);
```

3

ORACLE

Use the ON clause to specify a join condition. With this, you can specify join conditions separate from any search or filter conditions in the WHERE clause.

Retrieving Records with the ON Clause

```
SELECT e.employee_id, e.last_name, e.department_id,
       d.department_id, d.location_id
FROM   employees e JOIN departments d
ON     (e.department_id = d.department_id);
```

	EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_ID_1	LOCATION_ID
1	200	Whalen	10	10	1700
2	201	Hartstein	20	20	1800
3	202	Fay	20	20	1800
4	124	Mourgos	50	50	1500
5	144	Vargas	50	50	1500
6	143	Matos	50	50	1500
7	142	Davies	50	50	1500
8	141	Rajs	50	50	1500
9	107	Lorentz	60	60	1400
10	104	Ernst	60	60	1400
11	103	Hunold	60	60	1400

...

ORACLE

In this example, the DEPARTMENT_ID columns in the EMPLOYEES and DEPARTMENTS table are joined using the ON clause. Wherever a department ID in the EMPLOYEES table equals a department ID in the DEPARTMENTS table, the row is returned. The table alias is necessary to qualify the matching column_names.

You can also use the ON clause to join columns that have different names. The parentheses around the joined columns, as in the example in the slide (e.department_id = d.department_id), is optional. So, even ON e.department_id = d.department_id will work.

Note: When you use the Execute Statement icon to run the query, SQL Developer suffixes a '_1' to differentiate between the two department_ids.

Creating Three-Way Joins

```
SELECT employee_id, city, department_name
FROM   employees e
JOIN   departments d
ON     d.department_id = e.department_id
JOIN   locations l
ON     d.location_id = l.location_id;
```

	EMPLOYEE_ID	CITY	DEPARTMENT_NAME
1	100	Seattle	Executive
2	101	Seattle	Executive
3	102	Seattle	Executive
4	103	Southlake	IT
5	104	Southlake	IT
6	107	Southlake	IT
7	124	South San Francisco	Shipping
8	141	South San Francisco	Shipping
9	142	South San Francisco	Shipping

...

ORACLE

A three-way join is a join of three tables. Here, the first join to be performed is `EMPLOYEES JOIN DEPARTMENTS`. The first join condition can reference columns in `EMPLOYEES` and `DEPARTMENTS` but cannot reference columns in `LOCATIONS`. The second join condition can reference columns from all three tables.

Note: The code example in the slide can also be accomplished with the `USING` clause:

```
SELECT e.employee_id, l.city, d.department_name
FROM employees e
JOIN departments d
USING (department_id)
JOIN locations l
USING (location_id);
```

Applying Additional Conditions to a Join

Use the AND clause or the WHERE clause to apply additional conditions:

```
SELECT e.employee_id, e.last_name, e.department_id,
       d.department_id, d.location_id
FROM   employees e JOIN departments d
ON     (e.department_id = d.department_id)
AND    e.manager_id = 149 ;
```

Or

```
SELECT e.employee_id, e.last_name, e.department_id,
       d.department_id, d.location_id
FROM   employees e JOIN departments d
ON     (e.department_id = d.department_id)
WHERE   e.manager_id = 149 ;
```

ORACLE

You can apply additional conditions to the join.

The example shown in the slide performs a join on the EMPLOYEES and DEPARTMENTS tables and, in addition, displays only employees who have a manager ID of 149. To add additional conditions to the ON clause, you can add AND clauses. Alternatively, you can use a WHERE clause to apply additional conditions.

Both the queries produce the same output.

Lesson Agenda

- Types of `JOINS` and their syntax
  - Natural join
  - Join with the `USING` clause
  - Join with the `ON` clause
- Self-join
- Nonequijoins
- `OUTER` joins
- Cartesian product
  - Cross join

Joining a Table to Itself

EMPLOYEES (WORKER)

EMPLOYEE_ID	LAST_NAME	MANAGER_ID
200	Whalen	101
201	Hartstein	100
202	Fay	201
205	Higgins	101
206	Gietz	205
100	King	(null)
101	Kochhar	100
102	De Haan	100
103	Hunold	102
104	Ernst	103

. . .

EMPLOYEES (MANAGER)

EMPLOYEE_ID	LAST_NAME
200	Whalen
201	Hartstein
202	Fay
205	Higgins
206	Gietz
100	King
101	Kochhar
102	De Haan
103	Hunold
104	Ernst

. . .

MANAGER_ID in the WORKER table is equal to EMPLOYEE_ID in the MANAGER table.

ORACLE®

Sometimes, you need to join a table to itself. To find the name of each employee's manager, you need to join the EMPLOYEES table to itself, or perform a self-join. For example, to find the name of Ernst's manager, you need to perform the following steps:

- Find Ernst in the EMPLOYEES table by looking at the LAST_NAME column.
- Find the manager number for Ernst by looking at the MANAGER_ID column. Ernst's manager number is 103.
- Find the name of the manager with EMPLOYEE_ID 103 by looking at the LAST_NAME column. Hunold's employee number is 103, so Hunold is Ernst's manager.

In this process, you look in the table twice. The first time you look in the table to find Ernst in the LAST_NAME column and the MANAGER_ID value of 103. The second time you look in the EMPLOYEE_ID column to find 103 and the LAST_NAME column to find Hunold.

Self-Joins Using the ON Clause

```
SELECT worker.last_name emp, manager.last_name mgr
FROM   employees worker JOIN employees manager
ON     (worker.manager_id = manager.employee_id);
```

	EMP	MGR
1	Hunold	De Haan
2	Fay	Hartstein
3	Gietz	Higgins
4	Lorentz	Hunold
5	Ernst	Hunold
6	Zlotkey	King
7	Mourgos	King
8	Kochhar	King

...

The same EMPLOYEES table is referred to as 'worker' and 'manager' using table alias.

Using the column alias, you differentiate between the employee and the corresponding manager in the output.

The ON clause can also be used to join columns that have different names, within the same table or in a different table.

The example shown is a self-join of the EMPLOYEES table, based on the EMPLOYEE_ID and MANAGER_ID columns. In this process, you look in the table twice. The first time you look in the table to find Ernst in the LAST_NAME column and the MANAGER_ID value of 103. The second time you look in the EMPLOYEE_ID column to find 103 and the LAST_NAME column to find Hunold.

Lesson Agenda

- Types of `JOINS` and their syntax
  - Natural join
  - Join with the `USING` clause
  - Join with the `ON` clause
- Self-join
- Nonequijoins
- `OUTER` joins
- Cartesian product
  - Cross join

Nonequijoins

A nonequijoin is a join condition containing something other than an equality operator, for example, a BETWEEN operator.

EMPLOYEES

	LAST_NAME	SALARY
1	Whalen	4400
2	Hartstein	13000
3	Fay	6000
4	Higgins	12000
5	Gietz	8300
6	King	24000
7	Kochhar	17000
8	De Haan	17000
9	Hunold	9000
10	Ernst	6000
. . .		
19	Taylor	8600
20	Grant	7000

JOB_GRADES

	GRADE_LEVEL	LOWEST_SAL	HIGHEST_SAL
1	A	1000	2999
2	B	3000	5999
3	C	6000	9999
4	D	10000	14999
5	E	15000	24999
6	F	25000	40000

The JOB_GRADES table defines the LOWEST_SAL and HIGHEST_SAL range of values for each GRADE_LEVEL.

Therefore, the GRADE_LEVEL column can be used to assign grades to each employee.

A nonequijoin is a join condition containing something other than an equality operator.

The relationship between the EMPLOYEES table and the JOB_GRADES table is an example of a nonequijoin. The SALARY column in the EMPLOYEES table ranges between the values in the LOWEST_SAL and HIGHEST_SAL columns of the JOB_GRADES table. Therefore, each employee can be graded based on their salary. The relationship is obtained using an operator other than the equality (=) operator.

Retrieving Records with Nonequijoins

```
SELECT e.last_name, e.salary, j.grade_level
FROM   employees e JOIN job_grades j
ON     e.salary
       BETWEEN j.lowest_sal AND j.highest_sal;
```

	LAST_NAME	SALARY	GRADE_LEVEL
1	Vargas	2500	A
2	Matos	2600	A
3	Davies	3100	B
4	Rajs	3500	B
5	Lorentz	4200	B
6	Whalen	4400	B
7	Mourgos	5800	B
8	Ernst	6000	C
9	Fay	6000	C
10	Grant	7000	C

...

ORACLE

The example in the slide creates a nonequijoin to evaluate an employee's salary grade. The salary must be *between* any pair of the low and high salary ranges.

It is important to note that all employees appear exactly once when this query is executed. No employee is repeated in the list. There are two reasons for this:

- None of the rows in the JOB_GRADES table contain grades that overlap. That is, the salary value for an employee can lie only between the low-salary and high-salary values of one of the rows in the salary grade table.
- All of the employees' salaries lie within the limits provided by the job grade table. That is, no employee earns less than the lowest value contained in the LOWEST_SAL column or more than the highest value contained in the HIGHEST_SAL column.

Note: Other conditions (such as <= and >=) can be used, but BETWEEN is the simplest. Remember to specify the low value first and the high value last when using the BETWEEN condition. The Oracle server translates the BETWEEN condition to a pair of AND conditions. Therefore, using BETWEEN has no performance benefits, but should be used only for logical simplicity.

Table aliases have been specified in the slide example for performance reasons, not because of possible ambiguity.

Lesson Agenda

- Types of `JOINS` and their syntax
  - Natural join
  - Join with the `USING` clause
  - Join with the `ON` clause
- Self-join
- Nonequijoins
- `OUTER` joins
- Cartesian product
  - Cross join

ORACLE

Returning Records with No Direct Match Using OUTER Joins

DEPARTMENTS

	DEPARTMENT_NAME	DEPARTMENT_ID
1	Administration	10
2	Marketing	20
3	Shipping	50
4	IT	60
5	Sales	80
6	Executive	90
7	Accounting	110
8	Contracting	190

There are no employees in department 190.
Employee “Grant” does not have a department_ID; therefore is not seen in the equijoin result.

Equijoin with EMPLOYEES

	DEPARTMENT_ID	LAST_NAME
1	10	Whalen
2	20	Hartstein
3	20	Fay
4	110	Higgins
5	110	Gietz
6	90	King
7	90	Kochhar
8	90	De Haan
9	60	Hunold
10	60	Ernst
...		
18	80	Abel
19	80	Taylor

ORACLE

If a row does not satisfy a join condition, the row does not appear in the query result.

In the slide example, a simple equijoin condition is used on the EMPLOYEES and DEPARTMENTS tables to return the result on the right. The result set does not contain the following:

- Department ID 190, because there are no employees with that department ID recorded in the EMPLOYEES table
- The employee with the last name of Grant, because this employee has not been assigned a department ID

To return the department record that does not have any employees, or employees that do not have an assigned department, you can use an OUTER join.

INNER Versus OUTER Joins

- In SQL:1999, the join of two tables returning only matched rows is called an INNER join.
- A join between two tables that returns the results of the INNER join as well as the unmatched rows from the left (or right) table is called a left (or right) OUTER join.
- A join between two tables that returns the results of an INNER join as well as the results of a left and right join is a full OUTER join.

ORACLE

Joining tables with the NATURAL JOIN, USING, or ON clauses results in an INNER join. Any unmatched rows are not displayed in the output. To return the unmatched rows, you can use an OUTER join. An OUTER join returns all rows that satisfy the join condition and also returns some or all of those rows from one table for which no rows from the other table satisfy the join condition.

There are three types of OUTER joins:

- LEFT OUTER
- RIGHT OUTER
- FULL OUTER

LEFT OUTER JOIN

```
SELECT e.last_name, e.department_id, d.department_name
FROM   employees e LEFT OUTER JOIN departments d
ON    (e.department_id = d.department_id) ;
```

	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
1	Whalen	10	Administration
2	Fay	20	Marketing
3	Hartstein	20	Marketing
4	Vargas	50	Shipping
5	Matos	50	Shipping

. . .

16	Kochhar	90	Executive
17	King	90	Executive
18	Gietz	110	Accounting
19	Higgins	110	Accounting
20	Grant	(null)	(null)

`LEFT OUTER JOIN` shows the employee "Grant" who does not have a `department ID`.

ORACLE

This query retrieves all the rows in the `EMPLOYEES` table, which is the table on the left, even if there is no match in the `DEPARTMENTS` table.

RIGHT OUTER JOIN

```
SELECT e.last_name, d.department_id, d.department_name
FROM   employees e RIGHT OUTER JOIN departments d
ON     (e.department_id = d.department_id) ;
```

	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
1	Whalen	10	Administration
2	Hartstein	20	Marketing
3	Fay	20	Marketing
4	Davies	50	Shipping
5	Vargas	50	Shipping
6	Rajs	50	Shipping
7	Mourgos	50	Shipping
8	Matos	50	Shipping

. . .

18	Higgins	110	Accounting
19	Gietz	110	Accounting
20	(null)	190	Contracting

RIGHT OUTER JOIN shows the department that has no employees.

ORACLE

This query retrieves all the rows in the DEPARTMENTS table, which is the table on the right, even if there is no match in the EMPLOYEES table.

FULL OUTER JOIN

```
SELECT e.last_name, d.department_id, d.department_name
FROM   employees e FULL OUTER JOIN departments d
ON    (e.department_id = d.department_id) ;
```

	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
1	King	90	Executive
2	Kochhar	90	Executive
3	De Haan	90	Executive
4	Hunold	60	IT

. . .

15	Grant	(null)	(null)
16	Whalen	10	Administration
17	Hartstein	20	Marketing
18	Fay	20	Marketing
19	Higgins	110	Accounting
20	Gietz	110	Accounting
21	(null)	190	Contracting

FULL OUTER JOIN shows all the rows from the two tables, even if there is no match.

ORACLE

This query retrieves all the rows in the EMPLOYEES table, even if there is no match in the DEPARTMENTS table. It also retrieves all the rows in the DEPARTMENTS table, even if there is no match in the EMPLOYEES table.

Lesson Agenda

- Types of JOINS and their syntax
  - Natural join
  - Join with the USING clause
  - Join with the ON clause
- Self-join
- Nonequijoins
- OUTER joins
- Cartesian product
  - Cross join

Cartesian Products

- A Cartesian product is a join of every row of one table to every row of another table.
- It generates a large number of rows and the result is rarely useful.

Always include a valid join condition unless you have a specific need to combine all rows from all tables.

ORACLE

A Cartesian product tends to generate a large number of rows and the result is rarely useful. You should, therefore, always include a valid join condition unless you have a specific need to combine all rows from all tables.

Cartesian products are useful for some tests when you need to generate a large number of rows to simulate a reasonable amount of data.

Generating a Cartesian Product

EMPLOYEES (20 rows)

	EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
1	200	Whalen	10
2	201	Hartstein	20
3	202	Fay	20
4	205	Higgins	110
. . .			
19	176	Taylor	80
20	178	Grant	(null)

DEPARTMENTS (8 rows)

	DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
1	10	Administration	1700
2	20	Marketing	1800
3	50	Shipping	1500
4	60	IT	1400
5	80	Sales	2500
6	90	Executive	1700
7	110	Accounting	1700
8	190	Contracting	1700

Cartesian product:
20 x 8 = 160 rows

	EMPLOYEE_ID	DEPARTMENT_ID	LOCATION_ID
1	200	10	1700
2	201	20	1700
. . .			
21	200	10	1800
22	201	20	1800
. . .			
159	176	80	1700
160	178	(null)	1700

ORACLE®

A Cartesian product is generated if a join condition is omitted. The example in the slide displays the employee last name and the department name from the EMPLOYEES and DEPARTMENTS tables. Because no join condition was specified, all rows (20 rows) from the EMPLOYEES table are joined with all rows (8 rows) in the DEPARTMENTS table, thereby generating 160 rows in the output.

Creating Cross Joins

- A `CROSS JOIN` is a `JOIN` operation that produces the Cartesian product of two tables.
- To create a Cartesian product, specify the `CROSS JOIN` in your `SELECT` statement.

```
SELECT last_name, department_name
FROM   employees
CROSS JOIN departments ;
```

	LAST_NAME	DEPARTMENT_NAME
1	Abel	Administration
2	Davies	Administration
3	De Haan	Administration
4	Ernst	Administration
5	Fay	Administration
...		
158	Vargas	Contracting
159	Whalen	Contracting
160	Zlotkey	Contracting

ORACLE

The example in the slide produces a Cartesian product of the `EMPLOYEES` and `DEPARTMENTS` tables.

It is a good practice to explicitly state `CROSS JOIN` in your `SELECT` statement when you intend to create a Cartesian product. Therefore, it is very clear that you intend for this to happen and it is not the result of missing joins.

Quiz

Which of the following clauses of the JOIN syntax can you use to perform inner equijoins?

a. `CROSS JOIN`
b. `NATURAL JOIN`
c. `USING` clause
d. `ON` clause
e. `LEFT OUTER JOIN`

Answer: b, c, d

Quiz

Q

If you join a table to itself, what kind of join are you using?

a. Nonequijoin
b. Left `OUTER` join
c. Self-join
d. Natural join
e. Cartesian product

ORACLE

Answer: c

Quiz

Which type of join specifies a join condition containing operators other than the equality operator?

a. Nonequijoin

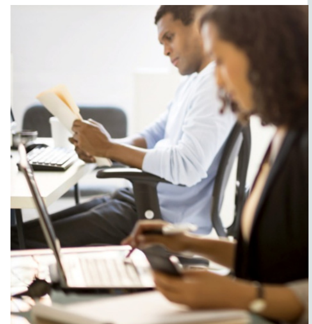
b. `CROSS JOIN`

Answer: a

Summary

In this lesson, you should have learned how to:

- Write SELECT statements to access data from more than one table using equijoins and nonequijoins
- Join a table to itself by using a self-join
- View data that generally does not meet a join condition by using OUTER joins
- Generate a Cartesian product of all rows from two or more tables



There are multiple ways to join tables.

Types of Joins

- Equijoins
- Nonequijoins
- OUTER joins
- Self-joins
- Cross joins
- Natural joins
- Full (or two-sided) OUTER joins

Cartesian Products

A Cartesian product results in the display of all combinations of rows. This is done by either omitting the WHERE clause or specifying the CROSS JOIN clause.

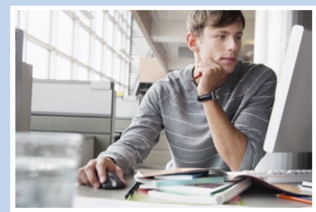
Table Aliases

- Table aliases speed up database access.
- They can help to keep SQL code smaller by conserving memory.
- They are sometimes mandatory to avoid column ambiguity.

Practice 11: Overview

This practice covers the following topics:

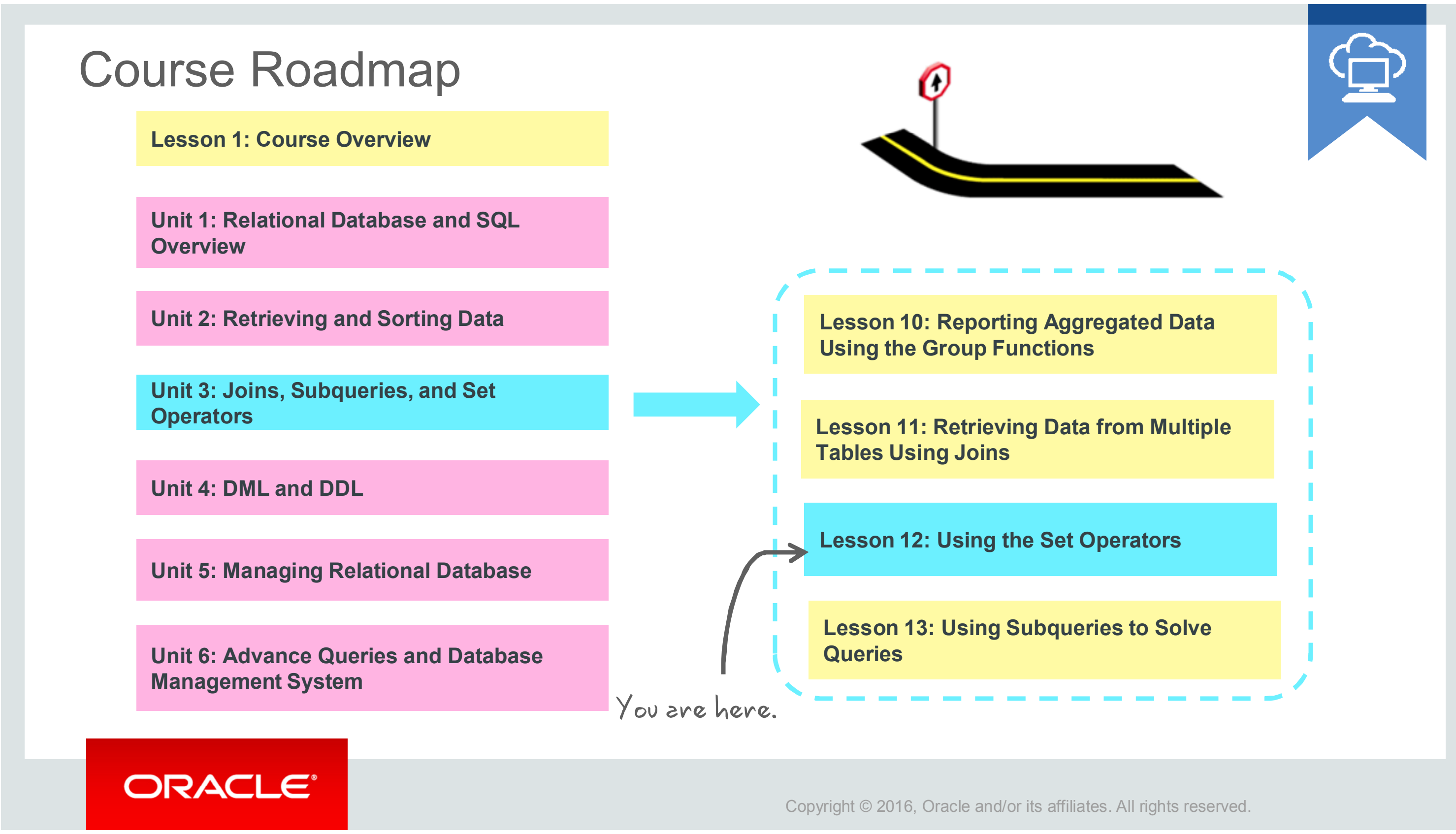
- Joining tables using an equijoin
- Performing outer and self-joins
- Adding conditions



ORACLE

This practice is intended to give you experience in extracting data from more than one table using the SQL:1999–compliant joins.


Lesson 12: Using the Set Operators
ORACLE
Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

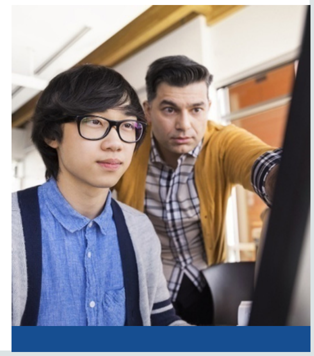


In Unit 3, you learn about using joins, subqueries, and set operators. You also learn to write compound queries in SQL to generate customized reports by using group functions, joins, and subqueries.

Objectives

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

- Describe set operators
- Use a set operator to combine multiple queries into a single query
- Control the order of rows returned



ORACLE

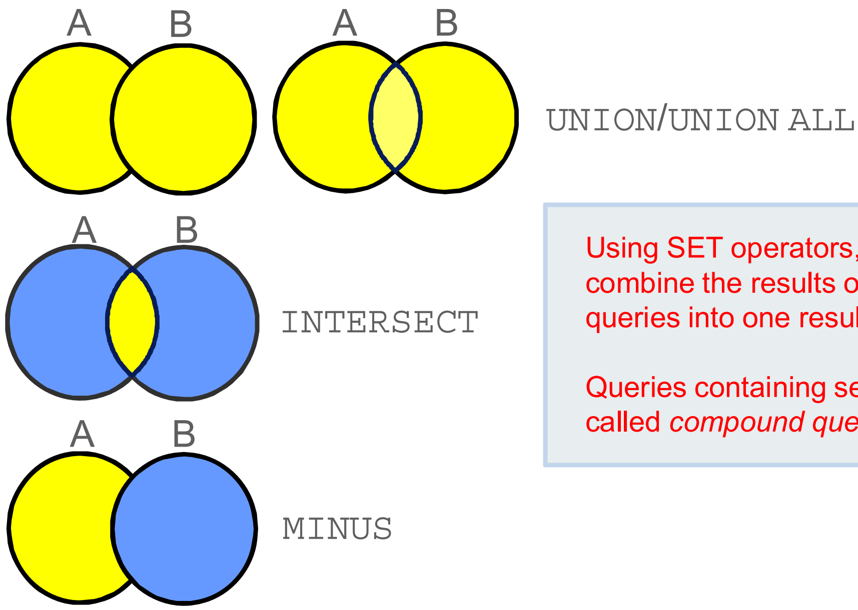
In this lesson, you learn how to write queries by using set operators.

Lesson Agenda

- Set operators: Types and guidelines
- Tables used in this lesson
- `UNION` and `UNION ALL` operators
- `INTERSECT` operator
- `MINUS` operator
- Matching the `SELECT` statements
- Using the `ORDER BY` clause in set operations

ORACLE

Set Operators



Using SET operators, you can combine the results of two or more queries into one result.

Queries containing set operators are called *compound queries*.

Set operators combine the results of two or more component queries into one result. Queries containing set operators are called *compound queries*.

All set operators have equal precedence. If a SQL statement contains multiple set operators, the Oracle server evaluates them from left (top) to right (bottom), if no parentheses explicitly specify another order. You should use parentheses to specify the order of evaluation explicitly in queries that use the `INTERSECT` operator with other set operators.

Set Operator Rules

- The expressions in the SELECT lists must match in number.
- The data type of each column in the subsequent query must match the data type of its corresponding column in the first query.
- Parentheses can be used to alter the sequence of execution.
- The ORDER BY clause can appear only at the very end of the statement.

- The expressions in the SELECT lists of the queries must match in number and data type. Queries that use the UNION, UNION ALL, INTERSECT, and MINUS operators must have the same number and data type of columns in their SELECT list. The data type of the columns in the SELECT list of the queries in the compound query may not be exactly the same. The column in the second query must be in the same data type group (such as numeric or character) as the corresponding column in the first query.
- Set operators can be used in subqueries.
- You should use parentheses to specify the order of evaluation in queries that use the INTERSECT operator with other set operators. This ensures compliance with emerging SQL standards that will give the INTERSECT operator greater precedence than the other set operators.

Oracle Server and Set Operators

- Duplicate rows are automatically eliminated except in UNION ALL.
- Column names from the first query appear in the result.
- The output is sorted in ascending order by default except in UNION ALL.

When a query uses set operators, the Oracle server eliminates duplicate rows automatically except in the case of the UNION ALL operator. The column names in the output are decided by the column list in the first SELECT statement. By default, the output is sorted in ascending order of the first column of the SELECT clause.

The corresponding expressions in the SELECT lists of the component queries of a compound query must match in number and data type. If component queries select character data, the data type of the return values is determined as follows:

- If both queries select values of CHAR data type, of equal length, the returned values have the CHAR data type of that length. If the queries select values of CHAR with different lengths, the returned value is VARCHAR2 with the length of the larger CHAR value.
- If either or both of the queries select values of VARCHAR2 data type, the returned values have the VARCHAR2 data type.

If component queries select numeric data, the data type of the return values is determined by numeric precedence. If all queries select values of the NUMBER type, the returned values have the NUMBER data type. In queries that use set operators, the Oracle server does not perform implicit conversion across data type groups. Therefore, if the corresponding expressions of component queries resolve to both character data and numeric data, the Oracle server returns an error.

Lesson Agenda

- Set operators: Types and guidelines
- Tables used in this lesson
- `UNION` and `UNION ALL` operators
- `INTERSECT` operator
- `MINUS` operator
- Matching the `SELECT` statements
- Using the `ORDER BY` clause in set operations

ORACLE

Tables Used in This Lesson

The tables used in this lesson are:

- EMPLOYEES: Provides details of all current employees
- RETIRED_EMPLOYEES: Provides details of all past employees

Two tables are used in this lesson: the EMPLOYEES table and the RETIRED_EMPLOYEES table.

You are already familiar with the EMPLOYEES table that stores employee details such as a unique identification number, email address, job identification (such as ST_CLERK, SA_REP, and so on), salary, manager, and so on.

RETIRED_EMPLOYEES stores the details of employees who have left the company.

The structure of and data from the EMPLOYEES and RETIRED_EMPLOYEES tables are shown on the following pages.

```
DESCRIBE employees
```

```
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)
```

```
SELECT employee_id, last_name, job_id, hire_date, department_id
FROM employees;
```

	EMPLOYEE_ID	LAST_NAME	JOB_ID	HIRE_DATE	DEPARTMENT_ID
1	100	King	AD_PRES	17-JUN-11	90
2	101	Kochhar	AD_VP	21-SEP-13	90
3	102	De Haan	AD_VP	13-JAN-09	90
4	103	Hunold	IT_PROG	03-JAN-14	60
5	104	Ernst	IT_PROG	21-MAY-15	60
6	107	Lorentz	IT_PROG	07-FEB-15	60
7	124	Mourgos	ST_MAN	16-NOV-15	50
8	141	Rajs	ST_CLERK	17-OCT-11	50
9	142	Davies	ST_CLERK	29-JAN-13	50
10	143	Matos	ST_CLERK	15-MAR-14	50
11	144	Vargas	ST_CLERK	09-JUL-14	50
12	149	Zlotkey	SA_MAN	29-JAN-16	80
13	174	Abel	SA_REP	11-MAY-12	80
14	176	Taylor	SA_REP	24-MAR-14	80
15	178	Grant	SA_REP	24-MAY-15	(null)
16	200	Whalen	AD_ASST	17-SEP-11	10
17	201	Hartstein	MK_MAN	17-FEB-12	20
18	202	Fay	MK_REP	17-AUG-13	20
19	205	Higgins	AC_MGR	07-JUN-10	110
20	206	Gietz	AC_ACCOUNT	07-JUN-10	110

```
DESCRIBE retired_employees
```

```
Name          Null Type
------------- ---- ------------
EMPLOYEE_ID        NUMBER(7)
FIRST_NAME         VARCHAR2(20)
LAST_NAME          VARCHAR2(20)
EMAIL              VARCHAR2(25)
RETIRED_DATE       DATE
JOB_ID             VARCHAR2(20)
SALARY             NUMBER(8,2)
MANAGER_ID         NUMBER(4)
DEPARTMENT_ID      NUMBER(6)
```

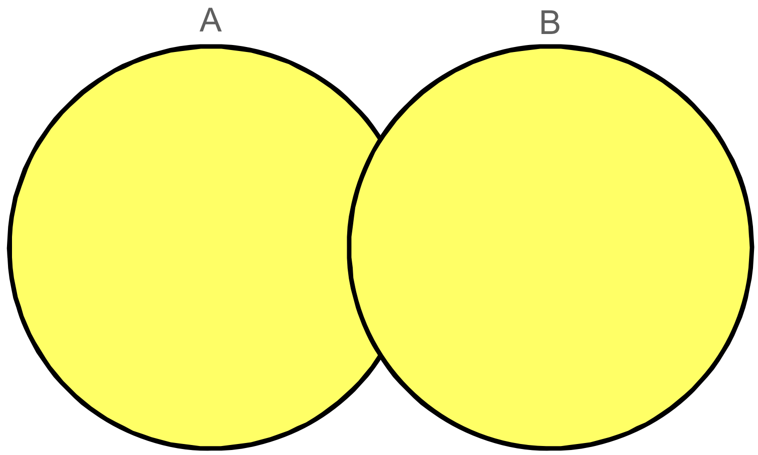
```
SELECT * FROM retired_employees;
```

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	RETIRED_DATE	JOB_ID	SALARY	MANAGER_ID	DEPARTMENT_ID
1	301	Rick	Dayle	RDAYLE	18-MAR-10	AD_PRES	8000	124	90
2	302	Meena	Rac	MRAC	21-SEP-11	AD_VP	11000	149	90
3	303	Mex	Haan	MHAAN	13-JAN-10	AD_VP	9500	149	80
4	304	Alexandera	Runold	ARUNOLD	03-JAN-11	IT_PROG	7500	124	60
5	305	Bruk	Ernst	BERNST	21-MAY-10	IT_PROG	6000	149	60
6	306	Dravid	Aust	DAUST	25-JUN-09	IT_PROG	4800	124	60
7	307	Raj	Patil	RPATIL	05-FEB-12	IT_PROG	4800	201	60
8	308	Rahul	Bose	RBOSE	17-AUG-12	FI_MGR	12008	124	100
9	309	Dany	Fav	DFAV	16-AUG-11	FI_ACCOUNT	9000	101	100
10	310	James	Ken	JKHEN	28-SEP-10	FI_ACCOUNT	8200	101	90
11	311	Shana	Garg	SGARG	30-SEP-10	FI_ACCOUNT	7700	201	100
12	312	Supriya	Ananth	SANANTH	07-JUN-14	FI_ACCOUNT	7800	124	100
13	313	Lui	Pops	LPOPS	07-DEC-10	FI_ACCOUNT	6900	201	100
14	314	Del	Raph	DRAPH	07-DEC-12	PU_MAN	11000	101	30
15	315	Alex	Khurl	AKHURL	18-MAY-11	PU_CLERK	3100	149	30

Lesson Agenda

- Set operators: Types and guidelines
- Tables used in this lesson
- UNION and UNION ALL operators
- INTERSECT operator
- MINUS operator
- Matching the SELECT statements
- Using the ORDER BY clause in set operations

UNION Operator



The UNION operator returns rows from both queries after eliminating duplications.

ORACLE

The UNION operator returns all rows that are selected by either query. Use the UNION operator to return all rows from multiple tables and eliminate any duplicate rows.

Guidelines

- The number of columns being selected must be the same.
- The data types of the columns being selected must be in the same data type group (such as numeric or character).
- The names of the columns need not be identical.
- UNION operates over all the columns being selected.
- NULL values are not ignored during duplicate checking.
- By default, the output is sorted in ascending order of the columns of the SELECT clause.

Using the UNION Operator

Display the job details of all current and retired employees. Display each job only once.

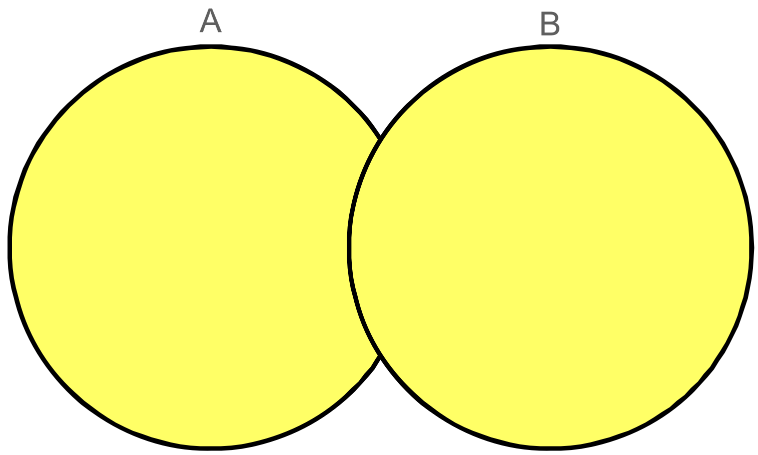
```
SELECT job_id
FROM   employees
UNION
SELECT job_id
FROM   retired_employees
```

	JOB_ID
1	AC_ACCOUNT
2	AC_MGR
3	AD_ASST
4	AD_PRES
5	AD_VP
6	FI_ACCOUNT
7	FI_MGR
8	IT_PROG
9	MK_MAN
10	MK_REP
11	PU_CLERK
12	PU_MAN
13	SA_MAN
14	SA_REP
15	ST_CLERK
16	ST_MAN

ORACLE

The UNION operator eliminates any duplicate records. If records that occur in both the EMPLOYEES and the RETIRED_EMPLOYEES tables are identical, the records are displayed only once.

UNION ALL Operator



The UNION ALL operator returns rows from both queries, including all duplications.

Use the UNION ALL operator to return all rows from multiple queries.

Guidelines

The guidelines for UNION and UNION ALL are the same, with the following two exceptions that pertain to UNION ALL: Unlike UNION, duplicate rows are not eliminated and the output is not sorted by default.

Using the UNION ALL Operator

Display the jobs and departments of all current and previous employees.

```
SELECT job_id, department_id
FROM   employees
UNION ALL
SELECT job_id, department_id
FROM   retired_employees
ORDER BY  job_id;
```

	JOB_ID	DEPARTMENT_ID
1	AC_ACCOUNT	110
2	AC_MGR	110
3	AD_ASST	10
4	AD_PRES	90
5	AD_PRES	90
6	AD_VP	90
7	AD_VP	80
8	AD_VP	90
9	AD_VP	90
10	FI_ACCOUNT	90
11	FI_ACCOUNT	100
12	FI_ACCOUNT	100
13	FI_ACCOUNT	100
14	FI_ACCOUNT	100
15	FI_MGR	100
16	IT_PROG	60
17	IT_PROG	60
18	IT_PROG	60
19	IT_PROG	60
20	IT_PROG	60
21	IT_PROG	60
22	IT_PROG	60
23	MK_MAN	20
24	MK_REP	20
25	PU_CLERK	30
26	PU_MAN	30
27	SA_MAN	80
28	SA_REP	80
29	SA_REP	80
30	SA_REP	(null)
31	ST_CLERK	50
32	ST_CLERK	50
33	ST_CLERK	50
34	ST_CLERK	50
35	ST_MAN	50

...

ORACLE

In the example, 35 rows are selected. The combination of the two tables totals to 35 rows. The UNION ALL operator does not eliminate duplicate rows. UNION returns all distinct rows selected by either query. UNION ALL returns all rows selected by either query, including all duplicates. Consider the query in the slide, now written with the UNION clause:

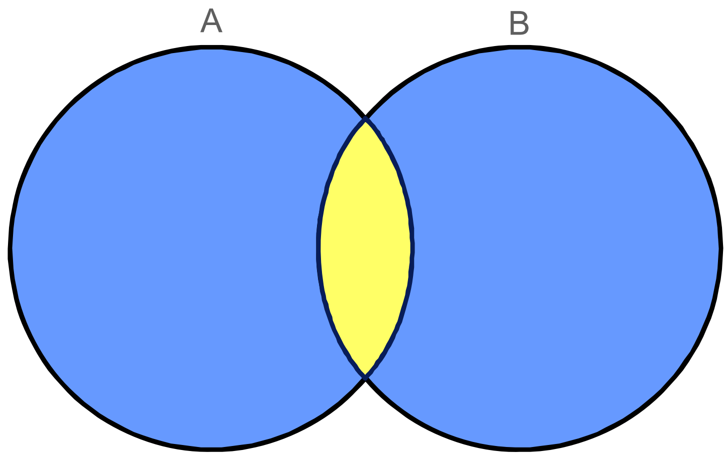
```
SELECT job_id,department_id
FROM     employees
UNION
SELECT job_id,department_id
FROM    retired_employees
ORDER BY job_id;
```

The preceding query returns 19 rows. This is because it eliminates all the duplicate rows.

Lesson Agenda

- Set operators: Types and guidelines
- Tables used in this lesson
- UNION and UNION ALL operators
- INTERSECT operator
- MINUS operator
- Matching the SELECT statements
- Using the ORDER BY clause in set operations

INTERSECT Operator



The INTERSECT operator returns rows that are common to both queries.

Use the INTERSECT operator to return all rows that are common to multiple queries.

Guidelines

- The number of columns and the data types of the columns being selected by the SELECT statements in the queries must be identical in all the SELECT statements used in the query. The names of the columns, however, need not be identical.
- Reversing the order of the intersected tables does not alter the result.
- INTERSECT does not ignore NULL values.

Using the `INTERSECT` Operator

Display the common manager IDs and department IDs of current and previous employees.

```
SELECT  manager_id,department_id
FROM    employees
INTERSECT
SELECT manager_id,department_id
FROM    retired_employees
```

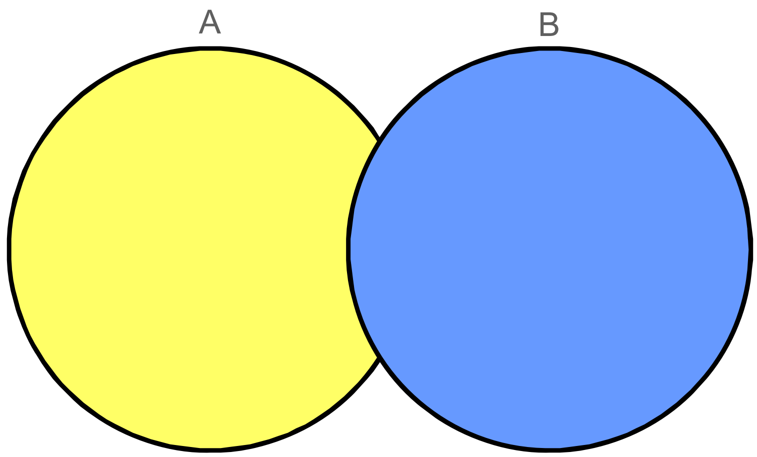
	MANAGER_ID	DEPARTMENT_ID
1	149	80

In the example in the slide, the query returns only those records that have the same values in the selected columns in both tables.

Lesson Agenda

- Set operators: Types and guidelines
- Tables used in this lesson
- `UNION` and `UNION ALL` operators
- `INTERSECT` operator
- `MINUS` operator
- Matching the `SELECT` statements
- Using the `ORDER BY` clause in set operations

`MINUS` Operator



The `MINUS` operator returns all the distinct rows selected by the first query, but not present in the second query result set.

Use the `MINUS` operator to return all distinct rows selected by the first query, but not present in the second query result set (the first `SELECT` statement `MINUS` the second `SELECT` statement).

Note: The number of columns must be the same and the data types of the columns being selected by the `SELECT` statements in the queries must belong to the same data type group in all the `SELECT` statements used in the query. The names of the columns, however, need not be identical.

Using the MINUS Operator

Display the employee IDs and job IDs of those employees who work in the sales department.

```
SELECT employee_id, job_id
FROM employees
WHERE department_id = 80
MINUS
SELECT employee_id, job_id
FROM retired_employees
WHERE department_id = 90;
```

	EMPLOYEE_ID	JOB_ID
1	149	SA_MAN
2	174	SA_REP
3	176	SA_REP

ORACLE

In the example in the slide, the employee IDs in the RETIRED_EMLOYEES table are subtracted from those in the EMPLOYEES table. The results set displays the employees remaining after the subtraction; they are represented by rows that exist in the EMPLOYEES table, but do not exist in the RETIRED_EMPLOYEES table. These are the records of employees who work in the sales department.

Lesson Agenda

- Set operators: Types and guidelines
- Tables used in this lesson
- `UNION` and `UNION ALL` operators
- `INTERSECT` operator
- `MINUS` operator
- Matching the `SELECT` statements
- Using the `ORDER BY` clause in set operations

ORACLE

Matching the SELECT Statements

You must match the data type (using the TO_CHAR function or any other conversion functions) when columns do not exist in one or the other table.

```
SELECT location_id, department_name "Department",
       TO_CHAR(NULL) "Warehouse location"
FROM departments
UNION
SELECT location_id, TO_CHAR(NULL) "Department",
       state_province
FROM locations;
```

ORACLE

Because the expressions in the SELECT lists of the queries must match in number, you can use dummy columns and the data type conversion functions to comply with this rule. To match the column list explicitly, you can insert NULL columns at the missing positions so as to match the count and data type of the selected columns in each SELECT statement. In the slide, the name Warehouse location is given as the dummy column heading. The TO_CHAR function is used in the first query to match the VARCHAR2 data type of the state_province column that is retrieved by the second query. Similarly, the TO_CHAR function in the second query is used to match the VARCHAR2 data type of the department_name column that is retrieved by the first query.

Matching the SELECT Statement: Example

Using the UNION operator, display the employee name, department_id, and location_id of all employees.

```
SELECT  first_name, job_id, TO_DATE(hire_date)"HIRE_DATE"
FROM employees
UNION
SELECT first_name, job_id, TO_DATE(null)"HIRE_DATE"
FROM retired_employees;
```

	FIRST_NAME	JOB_ID	HIRE_DATE
1	Alex	PU_CLERK	(null)
2	Alexander	IT_PROG	03-JAN-14
3	Alexandera	IT_PROG	(null)
4	Bruce	IT_PROG	21-MAY-15
5	Bruk	IT_PROG	(null)
6	Curtis	ST_CLERK	29-JAN-13
7	Dany	FI_ACCOUNT	(null)
8	Del	PU_MAN	(null)
9	Diana	IT_PROG	07-FEB-15
10	Dravid	IT_PROG	(null)
11	Eleni	SA_MAN	29-JAN-16
12	Ellen	SA_REP	11-MAY-12
13	James	FI_ACCOUNT	(null)
14	Jennifer	AD_ASST	17-SEP-11
15	Jonathon	SA_REP	24-MAR-14
16	Kevin	ST_MAN	16-NOV-15
17	Kimberely	SA_REP	24-MAY-15

...

ORACLE

The EMPLOYEES and RETIRED_EMPLOYEES tables have several columns in common (for example, EMPLOYEE_ID, JOB_ID, and DEPARTMENT_ID). But what if you want the query to display the FIRST_NAME, JOB_ID, and HIRE_DATE by using the UNION operator, knowing that HIRE_DATE exists only in the EMPLOYEES table?

The code example in the slide matches the FIRST_NAME and JOB_ID columns in the EMPLOYEES and RETIRED_EMPLOYEES tables. NULL is added to the RETIRED_EMPLOYEES SELECT statement to match the HIRE_DATE column in the EMPLOYEES SELECT statement.

In the results shown in the slide, each row in the output that corresponds to a record from the RETIRED_EMPLOYEES table contains a NULL in the HIRE_DATE column.

Lesson Agenda

- Set operators: Types and guidelines
- Tables used in this lesson
- UNION and UNION ALL operators
- INTERSECT operator
- MINUS operator
- Matching the SELECT statements
- Using the ORDER BY clause in set operations

Using the ORDER BY Clause in Set Operations

- The ORDER BY clause can appear only once at the end of the compound query.
- Component queries cannot have individual ORDER BY clauses.
- The ORDER BY clause recognizes only the columns of the first SELECT query.
- By default, the first column of the first SELECT query is used to sort the output in ascending order.

ORACLE

The ORDER BY clause can be used only once in a compound query. If used, the ORDER BY clause must be placed at the end of the query. The ORDER BY clause accepts the column name or an alias. By default, the output is sorted in ascending order of the first column of the first SELECT query.

Note: The ORDER BY clause does not recognize the column names of the second SELECT query. To avoid confusion over column names, it is a common practice to ORDER BY column positions.

For example, in the following statement, the output will be shown in ascending order of job_id.

```
SELECT employee_id, job_id,salary
FROM   employees
UNION
SELECT employee_id, job_id,0
FROM   retired_employees
ORDER BY 2;
```

If you omit ORDER BY, by default, the output will be sorted in ascending order of employee_id. You cannot use the columns from the second query to sort the output.

Quiz

Identify two set operator guidelines.

a. The expressions in the `SELECT` lists must match in number.

b. Parentheses may not be used to alter the sequence of execution.

c. The data type of each column in the second query must match the data type of its corresponding column in the first query.

d. The `ORDER BY` clause can be used only once in a compound query, unless a `UNION ALL` operator is used.

Answer: a, c

Quiz

Identify the set operator that returns all the distinct rows selected by the first query, but not present in the second query result set.

a. `INTERSECT`

b. `UNION`

c. `MINUS`

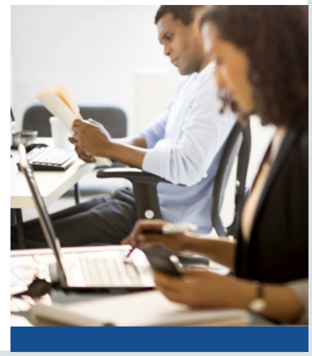
d. `UNION ALL`

Answer: c

Summary

In this lesson, you should have learned how to use:

- `UNION` to return all distinct rows
- `UNION ALL` to return all rows, including duplicates
- `INTERSECT` to return all rows that are shared by both queries
- `MINUS` to return all distinct rows that are selected by the first query, but not by the second
- `ORDER BY` only at the very end of the statement



- The `UNION` operator returns all the distinct rows selected by each query in the compound query. Use the `UNION` operator to return all rows from multiple tables and eliminate any duplicate rows.
- Use the `UNION ALL` operator to return all rows from multiple queries. Unlike with the `UNION` operator, duplicate rows are not eliminated and the output is not sorted by default.
- Use the `INTERSECT` operator to return all rows that are common to multiple queries.
- Use the `MINUS` operator to return rows returned by the first query that are not present in the second query.
- Remember to use the `ORDER BY` clause only at the very end of the compound statement.
- Make sure that the corresponding expressions in the `SELECT` lists match in number and data type.

Practice 12: Overview

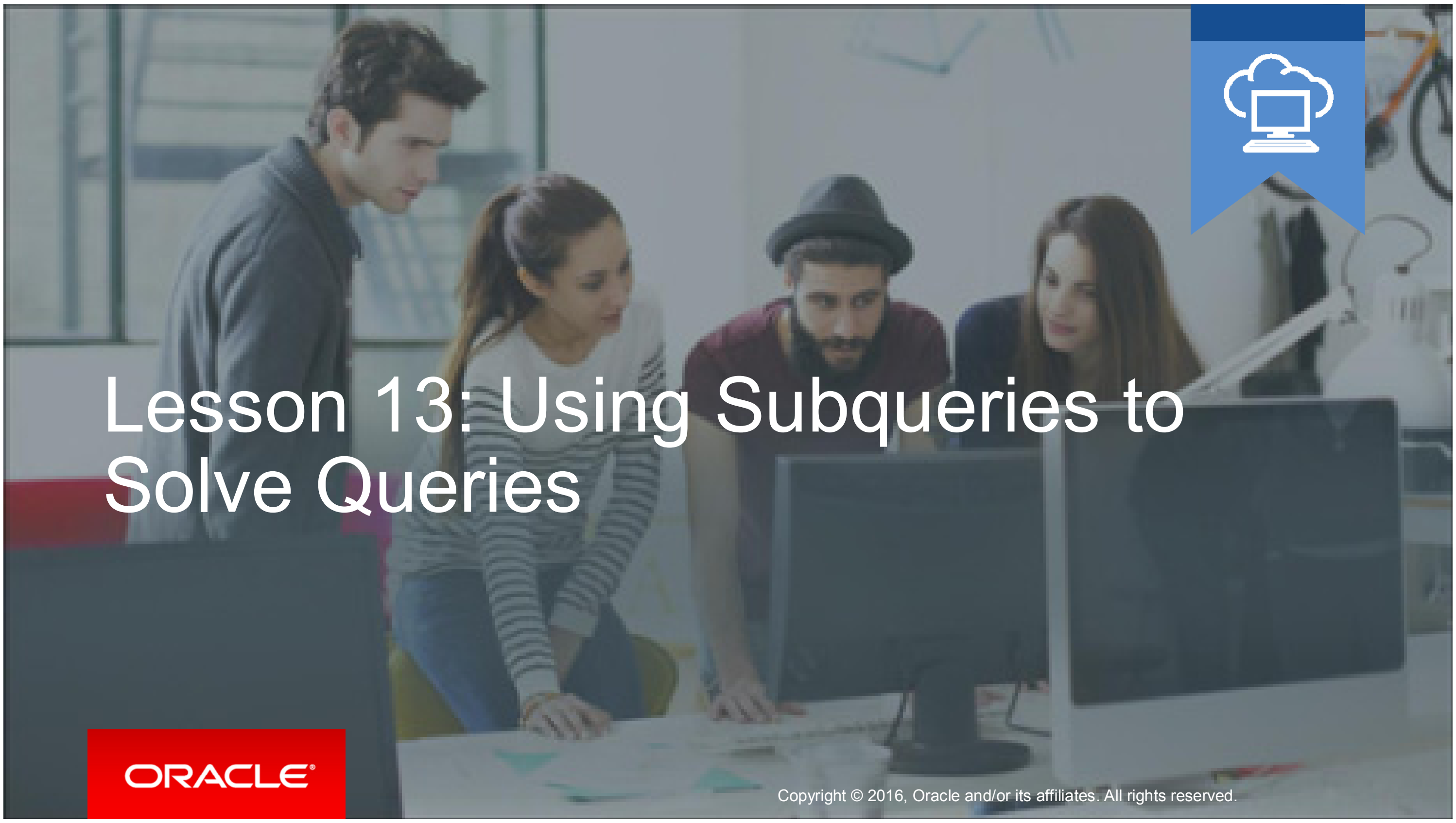
In this practice, you create reports by using:

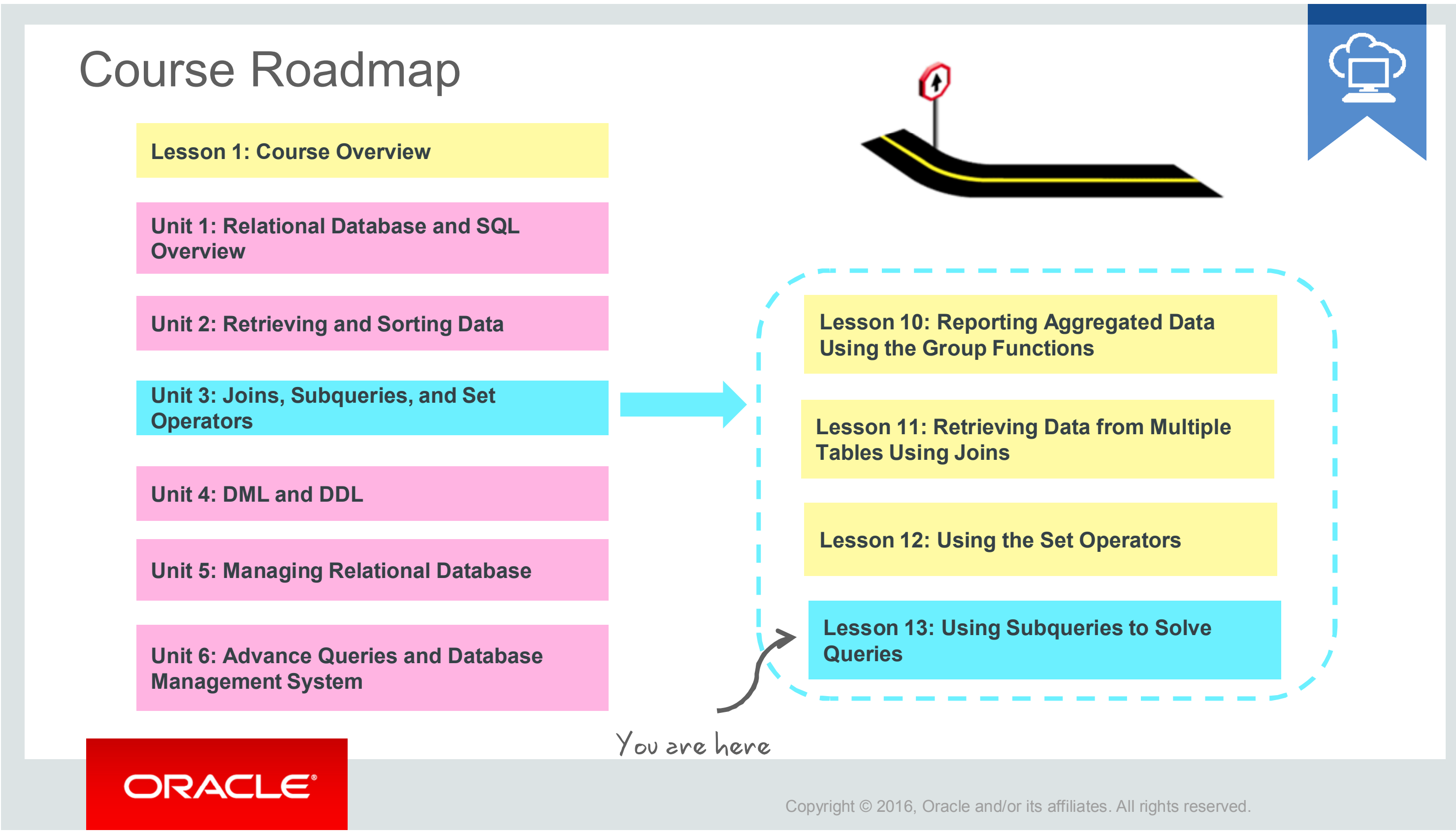
- The `UNION` operator
- The `INTERSECT` operator
- The `MINUS` operator

ORACLE

In this practice, you write queries by using the set operators.

Lesson 13: Using Subqueries to Solve Queries



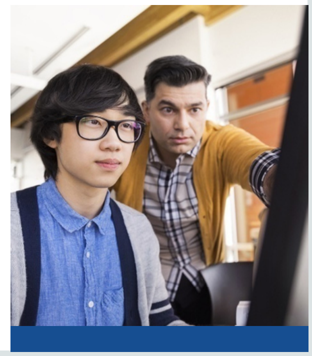


In Unit 3, you will learn about using joins, subqueries, and set operators. You will learn to write compound queries in SQL to generate customized reports using group functions, joins, and subqueries.

Objectives

After completing this lesson, you should be able to:

- Define subqueries
- Describe the types of problems that the subqueries can solve
- List the types of subqueries
- Write single-row, multiple-row, multiple-column subqueries



In this lesson, you learn about the more advanced features of the `SELECT` statement. You can write subqueries in the `WHERE` clause of another SQL statement to obtain values based on an unknown conditional value. This lesson also covers single-row subqueries, multiple-row subqueries, and multiple-column subqueries.

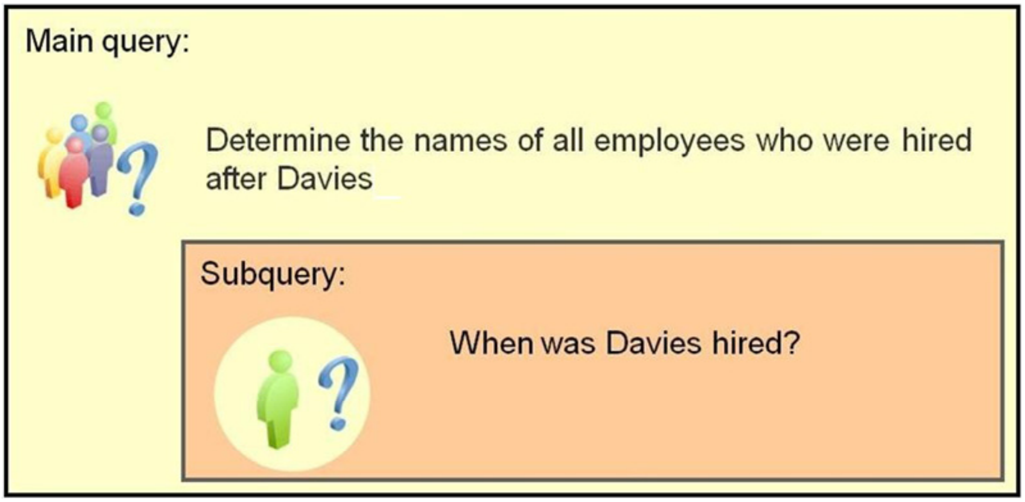
Lesson Agenda

- Subquery: Types, syntax, and guidelines
- Single-row subqueries:
  - Group functions in a subquery
  - `HAVING` clause with subqueries
- Multiple-row subqueries
  - Using `ALL` or `ANY` operator
- Multiple-column subqueries
- Null values in a subquery

ORACLE

Using a Subquery to Solve a Problem

Who is hired after Davies?



Suppose you want to write a query to find out the names of all employees who were hired after Davies.

To solve this problem, you need *two* queries: one query to find when Davies was hired, and a second query to find who were hired after Davies.

You can solve this problem by combining the two queries, placing one query *inside* the other query.

The inner query (or *subquery*) returns a value that is used by the outer query (or *main query*).

Subquery Syntax

- The subquery (inner query) executes *before* the main query (outer query).
- The result of the subquery is used by the main query.

```
SELECT   select_list
FROM     table
WHERE    expr operator
                          (SELECT  select_list
                           FROM              table);
```

In a SELECT statement, subqueries can be placed in:

- WHERE clause
- HAVING clause
- FROM clause

ORACLE

A subquery is a SELECT statement that is embedded in the clause of another SELECT statement. You can build powerful statements out of simple ones by using subqueries. They can be very useful when you need to select rows from a table with a condition that depends on the data in the table itself.

You can place the subquery in a number of SQL clauses, including the following:

- WHERE clause
- HAVING clause
- FROM clause

In the syntax:

operator includes a comparison condition such as >, =, or IN

The subquery is often referred to as a nested SELECT, sub-SELECT, or inner SELECT statement. The subquery generally executes first, and its output is used to complete the query condition for the main (or outer) query.

Using a Subquery

```
SELECT last_name, hire_date
FROM   employees          29-JAN-13
WHERE  hire_date > (SELECT hire_date
                    FROM   employees
                    WHERE  last_name = 'Davies');
```

	LAST_NAME	HIRE_DATE
1	Kochhar	21-SEP-13
2	Hunold	03-JAN-14
3	Ernst	21-MAY-15
4	Lorentz	07-FEB-15
5	Mourgos	16-NOV-15
6	Matos	15-MAR-14
7	Vargas	09-JUL-14
8	Zlotkey	29-JAN-16
9	Taylor	24-MAR-14
10	Grant	24-MAY-15
11	Fay	17-AUG-13

The inner query returns the hire date of 'Davies'. The outer query returns the last name and hire date of employees hired after 'Davies'.

In the slide, the inner query determines the hire date of the employee Davies. The outer query takes the result of the inner query and uses this result to display all the employees who were hired after Davies.

Rules and Guidelines for Using Subqueries

- Enclose subqueries in parentheses.
- Place subqueries on the right side of the comparison condition for readability. (However, the subquery can appear on either side of the comparison operator.)
- Use single-row operators with single-row subqueries and multiple-row operators with multiple-row subqueries.

ORACLE

- A subquery must be enclosed in parentheses.
- Place the subquery on the right side of the comparison condition for readability. However, the subquery can appear on either side of the comparison operator.
- Two classes of comparison conditions are used in subqueries: single-row operators and multiple-row operators.

Types of Subqueries

- Single-row subquery

- Multiple-row subquery

- **Single-row subqueries:** Queries that return only one row from the inner `SELECT` statement
- **Multiple-row subqueries:** Queries that return more than one row from the inner `SELECT` statement

Note: There are also multiple-column subqueries, which are queries that return more than one column from the inner `SELECT` statement. This is not covered in this lesson. For more information, refer to the *Oracle Database SQL Language Reference* for 12*c* database.

Lesson Agenda

- Subquery: Types, syntax, and guidelines
- Single-row subqueries:
  - Group functions in a subquery
  - HAVING clause with subqueries
- Multiple-row subqueries
  - Using ALL or ANY operator
- Multiple-column subqueries
- Null values in a subquery

Single-Row Subqueries

- Return only one row
- Use single-row comparison operators

Operator	Meaning
=	Equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
<>	Not equal to

ORACLE

A single-row subquery is one that returns one row from the inner `SELECT` statement. This type of subquery uses a single-row operator. The slide gives a list of single-row operators.

Single-Row Subqueries: Example

Display the employees whose job ID is the same as that of employee 141.

```
SELECT last_name, job_id
FROM   employees
WHERE  job_id =
               (SELECT job_id
                FROM   employees
                WHERE  employee_id = 141);
```

ST_CLERK

	LAST_NAME	JOB_ID
1	Rajs	ST_CLERK
2	Davies	ST_CLERK
3	Matos	ST_CLERK
4	Vargas	ST_CLERK

ORACLE

The example query uses the equal to (=) operator . The subquery returns the job_id of the employee 141. The main query displays the employees who have the same job_id.

Executing Single-Row Subqueries

```
SELECT last_name, job_id, salary
FROM   employees
WHERE  job_id =                                  SA_REP
                  (SELECT job_id
                   FROM   employees
                   WHERE  last_name = 'Taylor')

AND    salary >                                  8600
                  (SELECT salary
                   FROM   employees
                   WHERE  last_name = 'Taylor');
```

	LAST_NAME	JOB_ID	SALARY
1	Abel	SA_REP	11000

ORACLE

A `SELECT` statement can be considered as a query block. The example in the slide displays employees who do the same job as "Taylor," but earn more salary than him.

The example consists of three query blocks: the outer query and two inner queries. The inner query blocks are executed first, producing the query results `SA_REP` and `8600`, respectively. The outer query block is then processed and uses the values that were returned by the inner queries to complete its search conditions.

Both inner queries return single values (`SA_REP` and `8600`, respectively), so this SQL statement is called a single-row subquery.

Note: The outer and inner queries can get data from different tables.

Using Group Functions in a Subquery

```
SELECT last_name, job_id, salary
FROM   employees
WHERE  salary =
                 (SELECT MIN(salary)
                  FROM   employees);
```

2500

	LAST_NAME	JOB_ID	SALARY
1	Vargas	ST_CLERK	2500

You can display data from a main query by using a group function in a subquery to return a single row. The subquery is in parentheses and is placed after the comparison condition.

The example in the slide displays the employee last name, job `ID`, and salary of all employees whose salary is equal to the minimum salary. The `MIN` group function returns a single value (`2500`) to the outer query.

HAVING Clause with Subqueries

- The Oracle server executes the subqueries first.
- The Oracle server returns results into the HAVING clause of the main query.

```
SELECT    department_id, MIN(salary)
FROM      employees
GROUP BY department_id
HAVING    MIN(salary) >                    4200
                            (SELECT MIN(salary)
                             FROM   employees
                             WHERE  department_id = 60);
```

	DEPARTMENT_ID	MIN(SALARY)
1	(null)	7000
2	90	17000
3	20	6000
4	110	8300
5	80	8600
6	10	4400

ORACLE

You can use subqueries not only in the WHERE clause, but also in the HAVING clause. The Oracle server executes the subquery and the results are returned into the HAVING clause of the main query.

The SQL statement in the slide displays all the departments that have a minimum salary greater than that of department 60.

Example

Find the job with the lowest average salary.

```
SELECT    job_id, AVG(salary)
FROM      employees
GROUP BY job_id
HAVING    AVG(salary) = (SELECT   MIN(AVG(salary))
                         FROM     employees
                         GROUP BY job_id);
```

What Is Wrong with This Statement?

```
SELECT employee_id, last_name
FROM   employees
WHERE  salary =
                  (SELECT   MIN(salary)
                   FROM     employees
                   GROUP BY department_id);
```

```
ORA-01427: single-row subquery returns more than one row
01427. 00000 - "single-row subquery returns more than one row"
*Cause:
*Action:
```

Single-row operator with multiple-row subquery

To correct this error, change the `=` operator to `IN`.

A common error with subqueries occurs when more than one row is returned for a single-row subquery.

In the SQL statement in the slide, the subquery contains a `GROUP BY` clause, which implies that the subquery will return multiple rows, one for each group that it finds. In this case, the results of the subquery are `4400`, `6000`, `2500`, `4200`, `7000`, `17000`, and `8300`.

The outer query takes those results and uses them in its `WHERE` clause. The `WHERE` clause contains an equal (`=`) operator, a single-row comparison operator that expects only one value. The `=` operator cannot accept more than one value from the subquery and, therefore, generates the error.

To correct this error, change the `=` operator to `IN`.

No Rows Returned by the Inner Query

```
SELECT last_name, job_id
FROM   employees
WHERE  job_id =
                 (SELECT job_id
                  FROM   employees
                  WHERE  last_name = 'Haas');
```

The subquery returns no rows because there is no employee named “Haas.”

ORACLE

Another common problem with subqueries occurs when no rows are returned by the inner query.

In the SQL statement in the slide, the subquery contains a `WHERE` clause. Presumably, the intention is to find the employee whose name is Haas. The statement is correct, but it selects no rows when executed because there is no employee named Haas. Therefore, the subquery returns no rows.

The outer query takes the results of the subquery (null) and uses these results in its `WHERE` clause. The outer query finds no employee with a job `ID` equal to `NULL`, and so returns no rows. If a job existed with a value of null, the row is not returned because comparison of two null values yields a null; therefore, the `WHERE` condition is not true.

Lesson Agenda

- Subquery: Types, syntax, and guidelines
- Single-row subqueries:
  - Group functions in a subquery
  - `HAVING` clause with subqueries
- Multiple-row subqueries
  - Use `IN`, `ALL`, or `ANY`
- Multiple-column subqueries
- Null values in a subquery

Multiple-Row Subqueries

- Return more than one row
- Use multiple-row comparison operators

Operator	Meaning
`IN`	Equal to any member in the list
`ANY`	Must be preceded by `=, !=, >, <, <=, >=`. It returns `TRUE` if at least one element exists in the result set of the subquery for which the relation is `TRUE`.
`ALL`	Must be preceded by `=, !=, >, <, <=, >=`. It returns `TRUE` if the relation is `TRUE` for all elements in the result set of the subquery.

ORACLE

Subqueries that return more than one row are called multiple-row subqueries. You use a multiple-row operator, instead of a single-row operator, with a multiple-row subquery. The multiple-row operator expects one or more values.

Using the `IN` Operator in Multiple-Row Subqueries

```
SELECT last_name, salary, department_id
FROM   employees
WHERE  salary IN
                (SELECT   MIN(salary)
                 FROM     employees
                 GROUP BY department_id);
```

The main query appears to the Oracle server as shown below:

```
SELECT last_name, salary, department_id
FROM   employees
WHERE  salary IN
                (2500, 4200, 4400, 6000, 7000,
8300, 8600, 17000);
```

ORACLE

The example finds the employees who earn the same salary as the minimum salary for each department.

The inner query is executed first, producing a query result. The main query block is then processed and uses the values that were returned by the inner query to complete its search condition.

Using the ANY Operator in Multiple-Row Subqueries

```
SELECT employee_id, last_name, job_id, salary
FROM   employees
WHERE  salary < ANY
                    (SELECT salary
                     FROM   employees
                     WHERE  job_id = 'IT_PROG')
AND    job_id <> 'IT_PROG';
```

9000, 6000, 4200

	EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
1	144	Vargas	ST_CLERK	2500
2	143	Matos	ST_CLERK	2600
3	142	Davies	ST_CLERK	3100
4	141	Rajs	ST_CLERK	3500
5	200	Whalen	AD_ASST	4400
...				
9	206	Gietz	AC_ACCOUNT	8300
10	176	Taylor	SA_REP	8600

Displays employees who are not IT programmers and whose salary is less than that of *any* IT programmer

ORACLE

The ANY operator (and its synonym, the SOME operator) compares a value to *each* value returned by a subquery. The slide example displays employees who are not IT programmers and whose salary is less than that of any IT programmer. The maximum salary that a programmer earns is $9,000.

- <ANY means less than the maximum.
- >ANY means more than the minimum.
- =ANY is equivalent to IN.

Using the ALL Operator in Multiple-Row Subqueries

```
SELECT employee_id, last_name, job_id, salary
FROM   employees
WHERE  salary < ALL
                    (SELECT salary
                     FROM   employees
                     WHERE  job_id = 'IT_PROG')
AND    job_id <> 'IT_PROG';
```

9000, 6000, 4200

	EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
1	141	Rajs	ST_CLERK	3500
2	142	Davies	ST_CLERK	3100
3	143	Matos	ST_CLERK	2600
4	144	Vargas	ST_CLERK	2500

Displays employees who are not IT programmers and whose salary is less than *all* the IT programmers

ORACLE

The ALL operator compares a value to *every* value returned by a subquery. The example in the slide displays employees whose salary is less than the salary of all employees with a job ID of IT_PROG and whose job is not IT_PROG.

>ALL means more than the maximum and <ALL means less than the minimum.

The NOT operator can be used with IN, ANY, and ALL operators.

Lesson Agenda

- Subquery: Types, syntax, and guidelines
- Single-row subqueries:
  - Group functions in a subquery
  - `HAVING` clause with subqueries
- Multiple-row subqueries
  - Use IN, ALL, or ANY
- Multiple-column subqueries
- Null values in a subquery

Multiple-Column Subqueries

- A multiple-column subquery returns more than one column to the outer query.
- Column comparisons in multiple column comparisons can be pairwise or nonpairwise.
- A multiple-column subquery can also be used in the `FROM` clause of a `SELECT` statement.

```
SELECT column, column, ...
FROM table
WHERE (column, column, ...) IN
                            (SELECT column, column, ...
                             FROM table
                             WHERE condition);
```

ORACLE

A multiple-column subquery returns more than one column to the outer query and can be listed in the outer query's `FROM`, `WHERE`, or `HAVING` clause.

If you want to compare two or more columns, you must write a compound `WHERE` clause using logical operators. Multiple-column subqueries enable you to combine duplicate `WHERE` conditions into a single `WHERE` clause.

`IN` operator is used to check a value within a set of values. The list of values may come from the results returned by a subquery.

Multiple-Column Subquery: Example

Display all the employees with the lowest salary in each department

```
SELECT first_name, department_id, salary
FROM employees
WHERE (salary, department_id) IN
      (SELECT min(salary), department_id
       FROM employees
       GROUP BY department_id)
ORDER BY department_id;
```

	FIRST_NAME	DEPARTMENT_ID	SALARY
1	Jennifer	10	4400
2	Pat	20	6000
3	Peter	50	2500
4	Diana	60	4200
5	Jonathon	80	8600
6	Neena	90	17000
7	Lex	90	17000
8	William	110	8300

ORACLE

The example in the slide is that of a multiple-column subquery because the subquery returns more than one column.

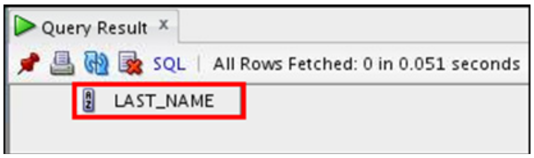
The inner query is executed first, and it returns the lowest `salary` and `department_id` for each department. The main query block is then processed and uses the values that were returned by the inner query to complete its search condition.

Lesson Agenda

- Subquery: Types, syntax, and guidelines
- Single-row subqueries:
  - Group functions in a subquery
  - `HAVING` clause with subqueries
- Multiple-row subqueries
  - Use IN, ALL, or ANY
- Multiple-column subqueries
- Null values in a subquery

Null Values in a Subquery

```
SELECT emp.last_name
FROM   employees emp
WHERE  emp.employee_id NOT IN
                           (SELECT mgr.manager_id
                            FROM   employees mgr);
```



The subquery returns no rows because one of the values returned by the subquery is **null**.

The SQL statement in the slide attempts to display all the employees who do not have any subordinates. Logically, this SQL statement should have returned 12 rows. However, the SQL statement does not return any rows. One of the values returned by the inner query is a null value and, therefore, the entire query returns no rows.

The reason is that all conditions that compare a null value result in a null. So whenever null values are likely to be part of the results set of a subquery, do not use the `NOT IN` operator. The `NOT IN` operator is equivalent to `<> ALL`.

Notice that the null value as part of the results set of a subquery is not a problem if you use the `IN` operator. The `IN` operator is equivalent to `=ANY`. For example, to display the employees who have subordinates, use the following SQL statement:

```
SELECT emp.last_name
FROM   employees emp
WHERE  emp.employee_id  IN
                         (SELECT mgr.manager_id
                          FROM   employees mgr);
```

Alternatively, a `WHERE` clause can be included in the subquery to display all employees who do not have any subordinates:

```
SELECT last_name FROM employees
WHERE  employee_id NOT IN
                    (SELECT manager_id
                     FROM   employees
                     WHERE  manager_id IS NOT NULL);
```

Quiz

Q

Using a subquery is equivalent to performing two sequential queries and using the result of the first query as the search values in the second query.

a. True
b. False

ORACLE

Answer: a

Quiz

In which of the following clauses of the `SELECT` statement can a subquery be nested?

a. `WHERE` clause
b. `HAVING` clause
c. `FROM` clause
d. `ORDER BY` clause

ORACLE

Answer: a, b, c

Quiz

Identify the multiple-row operators.

a. `IN`

b. `ANY`

c. `ALL`

d. `<>`

Answer: a, b, c

Quiz

If a single-row subquery returns a null value to the equal to (=) operator, the main query does not return any rows because comparison of two null values yields a null.

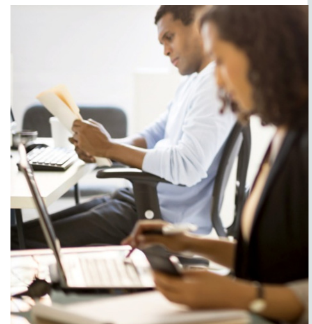
a. True
b. False

Answer: a

Summary

In this lesson, you should have learned how to:

- Define subqueries
- Identify the types of problems that the subqueries can solve
- Write single-row, multiple-row, multiple-column subqueries



ORACLE

In this lesson, you should have learned how to use subqueries. A subquery is a `SELECT` statement that is embedded in the clause of another SQL statement. Subqueries are useful when a query is based on a search criterion with unknown intermediate values.

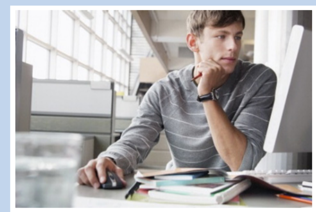
Subqueries have the following characteristics:

- Can pass one row of data to a main statement that contains a single-row operator, such as `=`, `<>`, `>`, `>=`, `<`, or `<=`
- Can pass multiple rows of data to a main statement that contains a multiple-row operator, such as `IN`
- Are processed first by the Oracle server, after which the `WHERE` or `HAVING` clause uses the results
- Can contain group functions

Practice 13: Overview

This practice covers the following topics:

- Creating subqueries to query values based on unknown criteria
- Using subqueries to find out the values that exist in one set of data and not in another

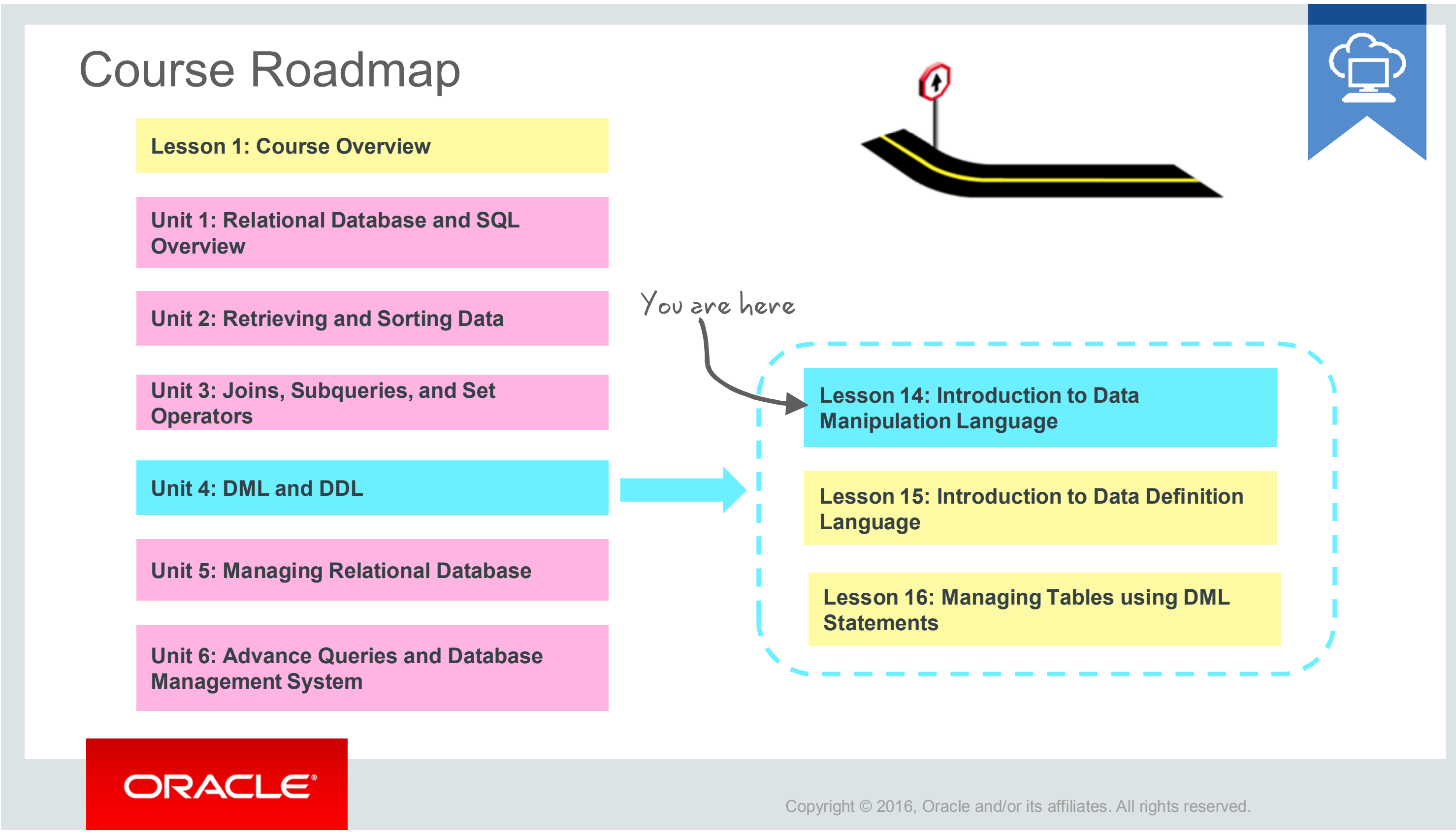


ORACLE

In this practice, you write simple queries using nested `SELECT` statements.

For practice questions, you may want to create the inner query first. Make sure that it runs and produces the data that you anticipate before you code the outer query.


Lesson 14: Introduction to Data Manipulation Language
ORACLE
Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

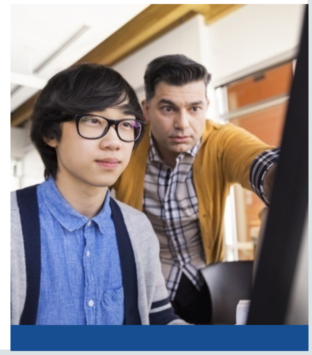


In Unit 4, you will learn about Data Manipulation Language (DML) and Data Definition Language (DDL). Using DML statements, you will learn to update and manage data in the tables. Using DDL statements, you will learn to create tables, remove tables, and so on.

Objectives

After completing this lesson, you should be able to:

- Describe each data manipulation language (DML) statement
- Control database transactions by using the `COMMIT`, `SAVEPOINT`, and `ROLLBACK` statements



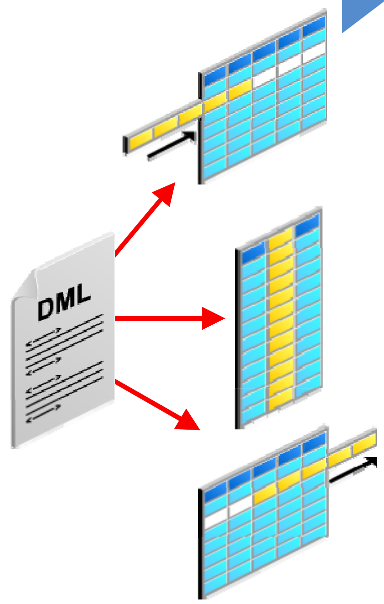
In this lesson, you learn how to use DML statements to insert rows into a table, update existing rows in a table, and delete existing rows from a table. You also learn how to control database transactions with the `COMMIT`, `SAVEPOINT`, and `ROLLBACK` statements.

Lesson Agenda

- Adding new rows in a table
  - `INSERT` statement
- Changing data in a table
  - `UPDATE` statement
- Removing rows from a table
  - `DELETE` statement
  - `TRUNCATE` statement
- Database transaction control using `COMMIT`, `ROLLBACK`, and `SAVEPOINT`
- Read Consistency

DML

- You write and execute a DML statement for:
  – Adding new rows to a table
  – Modifying existing rows in a table
  – Removing existing rows from a table
- A *transaction* consists of a collection of DML statements that forms a logical unit of work.



ORACLE

DML is a core part of SQL. When you want to add, update, or delete data in the database, you execute a DML statement. A collection of DML statements that forms a logical unit of work is called a *transaction*.

Consider a banking database. When a bank customer transfers money from a savings account to a checking account, the transaction might consist of three separate operations: decreasing the savings account, increasing the checking account, and recording the transaction in the transaction journal. The Oracle server must guarantee that all the three SQL statements are performed to maintain the accounts in proper balance. When something prevents one of the statements in the transaction from executing, the other statements of the transaction must be undone.

Note

- Most of the DML statements in this lesson assume that no constraints on the table are violated. Constraints are discussed later in this course.
- In SQL Developer, click the Run Script icon or press [F5] to run the DML statements. The feedback messages will be shown in the Script Output pane.

Adding a New Row to a Table

70 Public Relations	100	1700

New row

DEPARTMENTS

	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	10	Administration	200	1700
2	20	Marketing	201	1800
3	50	Shipping	124	1500
4	60	IT	103	1400
5	80	Sales	149	2500
6	90	Executive	100	1700
7	110	Accounting	205	1700
8	190	Contracting	(null)	1700

Insert a new row into the `DEPARTMENTS` table.

	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	70	Public Relations	100	1700
2	10	Administration	200	1700
3	20	Marketing	201	1800
4	50	Shipping	124	1500
5	60	IT	103	1400
6	80	Sales	149	2500
7	90	Executive	100	1700
8	110	Accounting	205	1700
9	190	Contracting	(null)	1700

INSERT Statement Syntax

- Add new rows to a table by using the INSERT statement:

Names of the columns to populate

```
INSERT INTO     table [(column [, column...])]
VALUES          (value [, value...]);
```

Name of the table (should already exist)

Corresponding values for the columns

With this syntax, only one row is inserted at a time.

ORACLE

You can add new rows to a table by issuing the INSERT statement. The syntax is explained in the slide. The table should already exist before you run the INSERT statement to insert rows in it.

Note: This statement with the VALUES clause adds only one row at a time to a table.

Inserting New Rows

- Insert a new row containing values for each column.
- List values in the default order of the columns in the table.
- Optionally, list the columns in the INSERT clause.

```
INSERT INTO departments(department_id,
       department_name, manager_id, location_id)
VALUES (70, 'Public Relations', 100, 1700);
```

```
1 rows inserted
```

- Enclose character and date values within single quotation marks.

Because you can insert a new row that contains values for each column, the column list is not required in the INSERT clause. However, if you do not use the column list, the values must be listed according to the default order of the columns in the table, and a value must be provided for each column. To view the column order of the table, use the following command:

```
DESCRIBE departments
```

For clarity, use the column list in the INSERT clause.

Enclose character and date values within single quotation marks; however, it is not recommended that you enclose numeric values within single quotation marks.

Inserting Rows with Null Values

- Implicit method: Omit the column from the column list.

```
INSERT INTO       departments (department_id,
                               department_name)
VALUES            (30, 'Purchasing');
```

```
1 rows inserted
```

- Explicit method: Specify the NULL keyword in the VALUES clause. Or enter empty string ('') in the VALUES list for character strings and dates.

```
INSERT INTO       departments
VALUES            (100, 'Finance', NULL, NULL);
```


Be sure that you can use null values in the targeted column by verifying the `Null` status with the `DESCRIBE` command.

The Oracle server automatically enforces all data types, data ranges, and data integrity constraints. Any column that is not listed explicitly obtains a null value in the new row, unless you have default values for the missing columns that are used.

Common errors that can occur during user input are checked in the following order:

- Mandatory value missing for a `NOT NULL` column
- Duplicate value violating any unique or primary key constraint
- `Any` value violating a `CHECK` constraint
- Referential integrity maintained for foreign key constraint
- Data type mismatches or values too wide to fit in a column

Note: Use of the column list is recommended because it makes the `INSERT` statement more readable and reliable, and less prone to mistakes.

Inserting Special Values

- The SYSDATE function records the current date and time.
- The CURRENT_DATE function records the current date in the session time zone.

```
INSERT INTO employees (employee_id,
                  first_name, last_name,
                  email, phone_number,
                  hire_date, job_id, salary,
                  commission_pct, manager_id,
                  department_id)
VALUES                (113,
                  'Louis', 'Popp',
                  'LPOPP', '515.124.4567',
                  CURRENT_DATE, 'AC_ACCOUNT', 6900,
                  NULL, 205, 110);
```

```
1 rows inserted
```

ORACLE

You can use functions to enter special values in your table.

The slide example records information for employee Popp in the EMPLOYEES table. It supplies the current date and time in the HIRE_DATE column. It uses the CURRENT_DATE function that returns the current date in the session time zone (if you use SYSDATE, it records the current date and time set at the database server). You can also use the USER function when inserting rows in a table. The USER function records the current username.

Confirming Additions to the Table

```
SELECT employee_id, last_name, job_id, hire_date, commission_pct
FROM   employees
WHERE  employee_id = 113;
```

Inserting Specific Date and Time Values

- Add a new employee.

```
INSERT INTO employees
VALUES      (114,
             'Den', 'Raphealy',
             'DRAPHEAL', '515.127.4561',
             TO_DATE('FEB 3, 2012', 'MON DD, YYYY'),
             'SA_REP', 11000, 0.2, 100, 60);
```

```
1 rows inserted
```

- Verify your addition.

21	114	Den	Raphealy	DRAPHEAL	515.127.4561	03-FEB-12	SA_REP	11000	0.2	100	60

The DD-MON-RR format is generally used to insert a date value. With the RR format, the system provides the correct century automatically.

You may also supply the date value in the DD-MON-YYYY or in the MON DD, YYYY format. This is recommended because it clearly specifies the century and does not depend on the internal RR format logic of specifying the correct century.

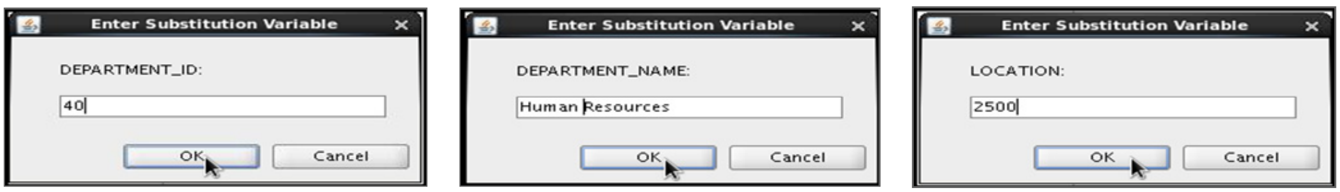
If a date must be entered in a format other than the default format (for example, with another century or a specific time), you must use the TO_DATE function.

The example in the slide records information for employee Raphealy in the EMPLOYEES table. It sets the HIRE_DATE column to be February 3, 2012.

Creating a Script

- Use the `&` substitution in a SQL statement to prompt for values.
- `&` is a placeholder for the variable value.

```
INSERT INTO departments
            (department_id, department_name, location_id)
VALUES      (&department_id, '&department_name',&location);
```



You can save commands with substitution variables to a file and execute the commands in the file. The example in the slide records information for a department in the `DEPARTMENTS` table.

Run the script file and you are prompted for input for each of the ampersand (`&`) substitution variables. After entering a value for the substitution variable, click the OK button. The values that you input are then substituted into the statement. This enables you to run the same script file over and over, but supply a different set of values each time you run it.

Copying Rows from Another Table

- To add rows derived from an existing table:

```
INSERT INTO table [ column (, column) ] subquery;
```

Name of the table

Columns to populate data

Subquery that returns rows to the table

- To create a copy of all the rows of a table:

```
INSERT INTO copy_emp
SELECT *
FROM EMPLOYEES;
```


You can use the `INSERT` statement to add rows to a table where the values are derived from existing tables.

In place of the `VALUES` clause, use a subquery. Zero or more rows are added depending on the number of rows returned by the subquery.
In the example, the `copy_emp` table must have been created before running the `INSERT` statement.

Copying Rows from Another Table

- 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%';
```

```
5 rows inserted.
```

- Do not use the VALUES clause.
- Match the number of columns and their data types in the INSERT clause to those in the subquery.
- Inserts all the rows returned by the subquery in the sales_reps table.

ORACLE

In the example in the slide, for the INSERT INTO statement to work, you must have already created the sales_reps table using the CREATE TABLE statement. CREATE TABLE is discussed in the next lesson titled "Introduction to Data Definition Language."

Note that in this INSERT statement, you do not use the VALUES clause. Instead, you add a subquery.

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. Zero or more rows are added depending on the number of rows returned by the subquery.

Lesson Agenda

- Adding new rows in a table
  - `INSERT` statement
- Changing data in a table
  - `UPDATE` statement
- Removing rows from a table
  - `DELETE` statement
  - `TRUNCATE` statement
- Database transaction control using `COMMIT`, `ROLLBACK`, and `SAVEPOINT`
- Read Consistency

ORACLE

Changing Data in a Table

EMPLOYEES

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	SALARY	MANAGER_ID	COMMISSION_PCT	DEPARTMENT_ID
100	Steven	King	24000	(null)	(null)	90
101	Neena	Kochhar	17000	100	(null)	90
102	Lex	De Haan	17000	100	(null)	90
103	Alexander	Hunold	9000	102	(null)	60
104	Bruce	Ernst	6000	103	(null)	60
107	Diana	Lorentz	4200	103	(null)	60
124	Kevin	Mourgos	5800	100	(null)	50

Change the department number for employees in department 60 to department 80.

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	SALARY	MANAGER_ID	COMMISSION_PCT	DEPARTMENT_ID
100	Steven	King	24000	(null)	(null)	90
101	Neena	Kochhar	17000	100	(null)	90
102	Lex	De Haan	17000	100	(null)	90
103	Alexander	Hunold	9000	102	(null)	80
104	Bruce	Ernst	6000	103	(null)	80
107	Diana	Lorentz	4200	103	(null)	80
124	Kevin	Mourgos	5800	100	(null)	50

ORACLE

UPDATE Statement Syntax

- Modify existing values in a table with the `UPDATE` statement:

Table in which you want to make changes

Value to change to

```
UPDATE      table
SET         column = value [, column = value, ...]
[WHERE      condition];
```

Column that needs to be updated

Identifies the rows to be updated and is composed of column names, expressions, constants, subqueries, and comparison operators

You can modify the existing values in a table by using the `UPDATE` statement. You can update more than one row at a time (if required). You specify a *`condition`* to identify the rows to be updated. The `condition` is composed of column names, expressions, constants, subqueries, and comparison operators.

Confirm the update operation by querying the table to display the updated rows.

For more information, see the section on "`UPDATE`" in *Oracle Database SQL Language Reference* for 12*c* database.

Note: In general, use the primary key column in the `WHERE` clause to identify a single row for update. Using other columns can unexpectedly cause several rows to be updated. For example, identifying a single row in the `EMPLOYEES` table by name may return more than one employee having the same name.

Updating Rows in a Table

- Values for a specific row or rows are modified if you specify the WHERE clause:

```
UPDATE employees
SET    department_id = 50
WHERE  employee_id = 113;
```

```
1 rows updated
```

- Values for all the rows in the table are modified if you omit the WHERE clause:

```
UPDATE   copy_emp
SET      department_id = 110;
```

```
22 rows updated
```


The UPDATE statement modifies the values of a specific row or rows if the WHERE clause is specified. The example in the slide shows the transfer of employee 113 (Popp) to department 50.

If you omit the WHERE clause, values for all the rows in the table are modified. Examine the updated rows in the COPY_EMP table.

```
SELECT last_name, department_id
FROM   copy_emp;
```

Note: The COPY_EMP table has the same data as the EMPLOYEES table.

Updating Rows in a Table

- Specify `SET column_name = NULL` to update a column value to `NULL`.

```
UPDATE employees
SET    job_id = 'IT_PROG', commission_pct = NULL
WHERE  employee_id = 114;
```

```
1 rows updated
```

Employee 114 ‘s `JOB_ID` is updated to `IT_PROG` and the commission field is set to `NULL`.

ORACLE

The slide example shows how you can update multiple columns for a single record and also shows how you can set `NULL` value for a column.

Updating Two Columns with a Subquery

Update employee 103's job and salary to match those of employee 205.

```
UPDATE   employees
SET      (job_id,salary)   = (SELECT  job_id,salary
                              FROM    employees
                              WHERE   employee_id = 205)
WHERE    employee_id     =  103;
```

```
1 rows updated
```

ORACLE

You can update multiple columns in the SET clause of an UPDATE statement by writing multiple subqueries. The syntax is as follows:

```
  UPDATE table
  SET     column   =
                   (SELECT column
                    FROM table
                    WHERE condition)
        [ ,
         column   =
                   (SELECT column
                    FROM table
                    WHERE condition)]
 [WHERE   condition ] ;
```

Updating Rows Based on Another Table

Use the subqueries in the UPDATE statements to update row values in a table based on values from another table:

```
UPDATE   copy_emp
SET      department_id  =  (SELECT department_id
                            FROM employees
                            WHERE employee_id = 100)
WHERE    job_id         =  (SELECT job_id
                            FROM employees
                            WHERE employee_id = 200);
```

```
1 rows updated
```

ORACLE

You can use subqueries in the UPDATE statements to update values in a table. The example in the slide updates the COPY_EMP table based on the values from the EMPLOYEES table. It changes the department number of all employees with employee 200's job ID to employee 100's current department number.

Lesson Agenda

- Adding new rows in a table
  - `INSERT` statement
- Changing data in a table
  - `UPDATE` statement
- Removing rows from a table
  - `DELETE` statement
  - `TRUNCATE` statement
- Database transaction control using `COMMIT`, `ROLLBACK`, and `SAVEPOINT`
- Read Consistency

Removing a Row from a Table

DEPARTMENTS

	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	10	Administration	200	1700
2	20	Marketing	201	1800
3	50	Shipping	124	1500
4	60	IT	103	1400
5	80	Sales	149	2500
6	90	Executive	100	1700
7	110	Accounting	205	1700
8	190	Contracting	(null)	1700

Delete a row from the DEPARTMENTS table:

	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	10	Administration	200	1700
2	20	Marketing	201	1800
3	50	Shipping	124	1500
4	60	IT	103	1400
5	80	Sales	149	2500
6	90	Executive	100	1700
7	110	Accounting	205	1700

ORACLE

The Contracting department has been removed from the DEPARTMENTS table (assuming no constraints on the DEPARTMENTS table are violated), as shown by the graphic in the slide.

DELETE Statement

You can remove existing rows from a table by using the DELETE statement:

```
DELETE [FROM]  table
[WHERE    condition];
```

Identifies the rows to be deleted using column names, expressions, constants, subqueries, and comparison operators

ORACLE

You can remove existing rows from a table by using the DELETE statement.

Note: If no rows are deleted, the message “0 rows deleted” is returned (in the Script Output pane in SQL Developer).

For more information, see the section on “DELETE” in *Oracle Database SQL Language Reference* for 12*c* database.

Deleting Rows from a Table

- Specific rows are deleted if you specify the WHERE clause:

```
DELETE FROM departments
WHERE  department_name = 'Finance';
1 rows deleted
```

- All rows in the table are deleted if you omit the WHERE clause:

```
DELETE FROM  copy_emp;
22 rows deleted
```

ORACLE

You can delete specific rows by specifying the WHERE clause in the DELETE statement. The first example in the slide deletes the Finance department from the DEPARTMENTS table. You can confirm the delete operation by displaying the deleted rows using the SELECT statement.

```
SELECT   *
FROM     departments
WHERE    department_name = 'Finance';
```

However, if you omit the WHERE clause, all rows in the table are deleted. The second example in the slide deletes all rows from the COPY_EMP table, because no WHERE clause was specified.

Example

Remove rows identified in the WHERE clause.

```
DELETE FROM  employees WHERE employee_id = 114;

DELETE FROM  departments WHERE department_id IN (30, 40);
```

Deleting Rows Based on Another Table

Use the subqueries in the `DELETE` statements to remove rows from a table based on values from another table:

```
DELETE FROM employees
WHERE   department_id IN
                (SELECT department_id
                 FROM   departments
                 WHERE  department_name
                        LIKE '%Public%');
```

```
1 rows deleted
```

ORACLE

You can use the subqueries to delete rows from a table based on values from another table. The example in the slide deletes all the employees in a department, where the department name contains the string `Public`.

The subquery searches the `DEPARTMENTS` table to find the department number based on the department name containing the string `Public`. The subquery then feeds the department number to the main query, which deletes rows of data from the `EMPLOYEES` table based on this department number.

TRUNCATE Statement

- Removes all rows from a table, leaving the table empty and the table structure intact
- Is a data definition language (DDL) statement rather than a DML statement; cannot easily be undone
- Syntax:

```
TRUNCATE TABLE table_name;
```

- Example:

```
TRUNCATE TABLE copy_emp;
```

ORACLE

A more efficient method of emptying a table is by using the TRUNCATE statement.
You can use the TRUNCATE statement to quickly remove all rows from a table or cluster. Removing rows with the TRUNCATE statement is faster than removing them with the DELETE statement because the TRUNCATE statement is a DDL statement and generates no rollback information. Rollback information is covered later in this lesson.

If the table is the parent of a referential integrity constraint, you cannot truncate the table. You need to disable the constraint before issuing the TRUNCATE statement. Disabling constraints is covered in the lesson titled "Introduction to DDL Statements."

Lesson Agenda

- Adding new rows in a table
  - `INSERT` statement
- Changing data in a table
  - `UPDATE` statement
- Removing rows from a table
  - `DELETE` statement
  - `TRUNCATE` statement
- Database transaction control using `COMMIT`, `ROLLBACK`, and `SAVEPOINT`
- Read Consistency

ORACLE

Database Transaction: Example

Withdraw cash.
If the ATM machine dispenses the cash, debit the account; commit it.
If it fails to dispense the cash, roll back the debit account transaction.

The Oracle server ensures data consistency based on transactions. Transactions give you more flexibility and control when changing data, and they ensure data consistency in the event of user process failure or system failure.

Transactions consist of DML statements that constitute one consistent change to the data. For example, a transfer of funds between two accounts should include the debit in one account and the credit to another account of the same amount. Both actions should either fail or succeed together; the credit should not be committed without the debit.

Database Transactions

A database transaction can be of the following types:

Type of transaction	Description
DML	Consists of any number of DML statements that the Oracle server treats as a single entity or a logical unit of work
DDL	Consists of only one DDL statement
Data control language (DCL)	Consists of only one DCL statement

The table lists the three types of database transactions. A DML database transaction is a set of DML operations that form a logical unit of work. A DDL database transaction consists of only one statement because when a DDL statement is executed, it is automatically committed as a single transaction. A DCL database transaction consists of creating users, roles, and privileges.

Database Transactions: Start and End

- Begin when the first DML SQL statement is executed.
- End with one of the following events:
  - A `COMMIT` or `ROLLBACK` statement is issued.
  - A DDL or DCL statement executes (automatic commit).
  - The user exits SQL Developer or SQL*Plus.
  - The system crashes.

ORACLE

When does a database transaction start and end?

A transaction begins when the first DML statement is encountered and ends when one of the following occurs:

- A `COMMIT` or `ROLLBACK` statement is issued.
- A DDL statement, such as `CREATE`, is issued.
- A DCL statement is issued.
- The user exits SQL Developer or SQL*Plus.
- The machine fails or the system crashes.

After one transaction ends, the next executable SQL statement automatically starts the next transaction.

A DDL statement or a DCL statement is automatically committed and, therefore, implicitly ends a transaction.

Advantages of COMMIT and ROLLBACK Statements

With COMMIT and ROLLBACK statements, you can:

- Ensure data consistency
- Preview data changes before making the changes permanent
- Group logically related operations

COMMIT	End your current transaction and make permanent all changes performed in the transaction
ROLLBACK	Undo work done in the current transaction

ORACLE

With the COMMIT and ROLLBACK statements, you have control over making changes to the data permanent.

- COMMIT ends the current transaction by making all pending data changes permanent.
- ROLLBACK ends the current transaction by discarding all pending data changes.

Explicit Transaction Control Statements

You can control the logic of transactions by using the `COMMIT`, `SAVEPOINT`, and `ROLLBACK` statements.

`SAVEPOINT` *`name`*	`SAVEPOINT` *`name`* marks a savepoint within the current transaction.
`ROLLBACK TO` *`SAVEPOINT name`*	`ROLLBACK TO <savepoint>` rolls back the current transaction to the specified savepoint.

ORACLE

You can control the logic of transactions by using the `COMMIT`, `SAVEPOINT`, and `ROLLBACK` statements. A SAVEPOINT identifies a point in a transaction to which you can later roll back.

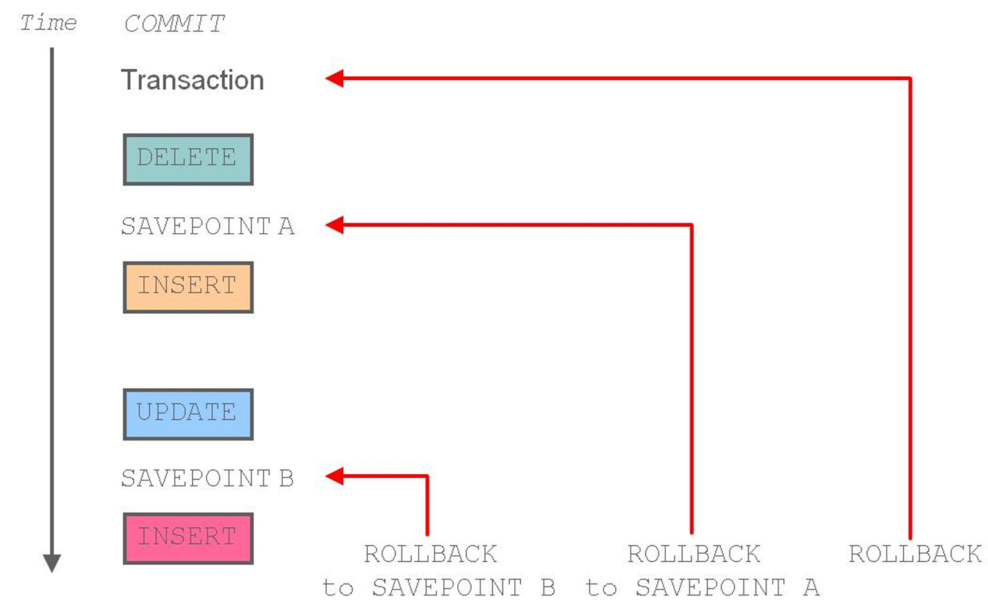
`SAVEPOINT` *`name`* marks a savepoint within the current transaction.

`ROLLBACK TO <savepoint>` rolls back the current transaction to the specified savepoint, thereby discarding any changes and/or savepoints that were created after the savepoint to which you are rolling back.

If you omit the `TO SAVEPOINT` clause, the `ROLLBACK` statement rolls back the entire transaction. Because savepoints are logical, there is no way to list the savepoints that you have created.

Note: You cannot `COMMIT` to a `SAVEPOINT`. `SAVEPOINT` is not ANSI-standard SQL.

Explicit Transaction Control Statements



You can control the logic of transactions by using the `COMMIT`, `SAVEPOINT`, and `ROLLBACK` statements. After creating a `SAVEPOINT A`, if you performed an `INSERT` and an `UPDATE`, and then realized you wanted to undo the change, you can roll back the transactions to `SAVEPOINT A`. If you use only `ROLLBACK`, it will undo all the transactions up to the last `COMMIT`.

Rolling Back Changes to a Marker

- Create a marker in the current transaction by using the `SAVEPOINT` statement.
- Roll back to that marker by using the `ROLLBACK TO SAVEPOINT` statement.

```
UPDATE...
SAVEPOINT update_done;
SAVEPOINT update_done succeeded.
INSERT...
ROLLBACK TO update_done;
ROLLBACK TO succeeded.
```

ORACLE

You can create a marker in the current transaction by using the `SAVEPOINT` statement, which divides the transaction into smaller sections. You can discard the changes done after the `SAVEPOINT` marker by using the `ROLLBACK TO SAVEPOINT` statement.

Note that if you create a second savepoint with the same name as an earlier savepoint, the earlier savepoint is deleted.

Implicit Transaction Processing

- An automatic commit occurs in the following circumstances:
  - A DDL statement is issued.
  - A DCL statement is issued.
- An automatic rollback occurs when there is an abnormal termination of SQL Developer or a system failure.

ORACLE

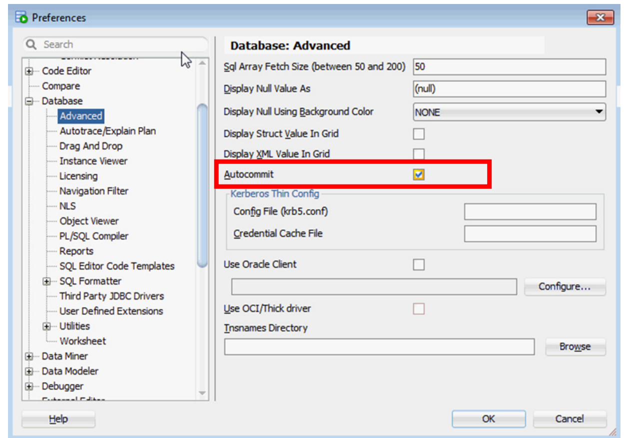
System Failures

When a transaction is interrupted by a system failure, the entire transaction is automatically rolled back. This prevents the error from causing unwanted changes to the data and returns the tables to the state at the time of the last commit. In this way, the Oracle server protects the integrity of the tables.

In SQL Developer, a normal exit from the session is accomplished by selecting Exit from the File menu. In SQL*Plus, a normal exit is accomplished by entering the `EXIT` command at the prompt.

Setting AutoCommit in SQL Developer

- Go to Tools > Preferences > Database > Advanced.
  - Select the Autocommit check box.
- If this option is selected, a commit operation is automatically performed after each DML statement executed using the SQL Worksheet.
- If this option is not selected, a commit operation is not performed until you execute a COMMIT statement.



ORACLE

To enable Autocommit, perform the following:

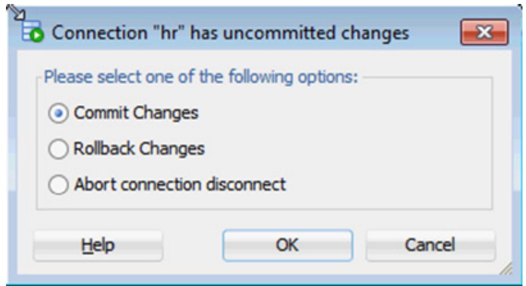
- In the **Tools** menu, select **Preferences**. In the Preferences dialog box, expand **Database** and select **Advanced**.
- In the right pane, select the **Autocommit** check box. Click **OK**.

If this option is selected, a commit operation is automatically performed after each INSERT, UPDATE, or DELETE statement executed using the SQL Worksheet. If this option is not selected, a commit operation is not performed until you execute a COMMIT statement.

Note: In SQL*Plus, the `AUTOCOMMIT` command can be toggled `ON` or `OFF`. If set to `ON`, each individual `DML` statement is committed as soon as it is executed. You cannot roll back the changes. If set to `OFF`, the `COMMIT` statement can still be issued explicitly.

Commit/Rollback on Exiting SQL Developer

- When the Autocommit option is not selected (default) and you exit SQL Developer by selecting File > Exit or by closing the window, the following pop-up window appears:



- You can decide to commit or roll back the changes.

When the Autocommit option is not selected (default) and you exit SQL Developer either by using the File > Exit option or by closing the window, a pop-up window is displayed. It prompts you to select either rollback or commit.

If the Autocommit option is selected, any DML operations are committed as soon as they are executed, so this pop-up window does not show up when you exit SQL Developer.

State of Data Before `COMMIT` or `ROLLBACK`

- Data manipulation operations primarily affect the database buffer; therefore, the previous state of the data can be recovered.
- The current session can review the results of the DML operations by using the `SELECT` statement.
- Other sessions *cannot* view the results of the DML statements issued by the current session.
- The affected rows are *locked*; other session cannot change the data in the affected rows.

Every data change made during the transaction is temporary until the transaction is committed.

The state of the data before `COMMIT` or `ROLLBACK` statements are issued can be described as follows:

- Data manipulation operations primarily affect the database buffer; therefore, the previous state of the data can be recovered.
- The current session can review the results of the data manipulation operations by querying the tables.
- Other sessions cannot view the results of the data manipulation operations made by the current session. The Oracle server institutes read consistency to ensure that each session sees data as it existed at the last commit.
- The affected rows are locked; other session cannot change the data in the affected rows.

State of Data After COMMIT

- Data changes are saved in the database.
- The previous state of the data is overwritten.
- All sessions can view the results.
- Locks on the affected rows are released; those rows are available for other sessions to manipulate.
- All savepoints are erased.

ORACLE®

Make all pending changes permanent by using the `COMMIT` statement. Here is what happens after a `COMMIT` statement:

- Data changes are written to the database.
- The previous state of the data is no longer available with normal SQL queries.
- All sessions can view the results of the transaction.
- The locks on the affected rows are released; the rows are now available for other sessions to perform new data changes.
- All savepoints are erased.

COMMIT: Example

- Make the changes:

```
DELETE FROM EMPLOYEES
WHERE employee_id=113;
1 rows deleted
INSERT INTO departments
VALUES (290, 'Corporate Tax', NULL, 1700);
1 rows inserted
```

- Commit the changes:

```
COMMIT;
commited.
```

ORACLE

In the example in the slide, a row is deleted from the EMPLOYEES table and a new row is inserted into the DEPARTMENTS table. The changes are saved by issuing the COMMIT statement.

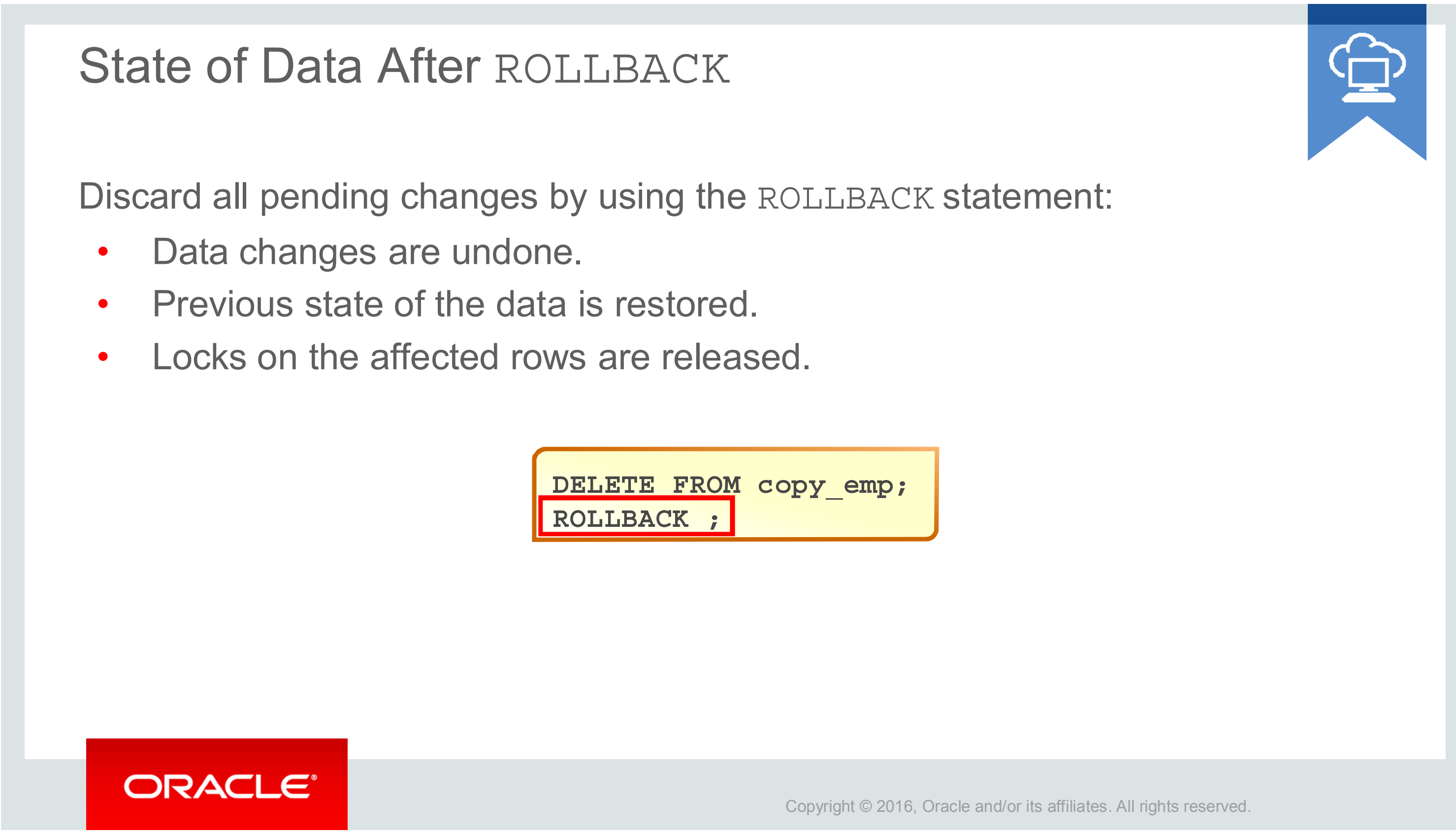
Example

Remove departments 290 and 300 from the DEPARTMENTS table and update a row in the EMPLOYEES table. Save the data change.

```
DELETE FROM departments
WHERE  department_id IN (290, 300);

UPDATE  employees
  SET   department_id = 80
  WHERE employee_id = 206;

COMMIT;
```



Discard all pending changes by using the `ROLLBACK` statement, which results in the following:

- Data changes are undone.
- The previous state of the data is restored.
- Locks on the affected rows are released.

ROLLBACK: Example

```
DELETE FROM test;
4 rows deleted.

ROLLBACK;
Rollback complete.

DELETE FROM test WHERE  id = 100;
1 row deleted.

SELECT * FROM   test WHERE  id = 100;
No rows selected.

COMMIT;
Commit complete.
```


While attempting to remove a record from the TEST table, you may accidentally empty the table. However, you can correct the mistake, reissue a proper statement, and make the data change permanent.

Statement-Level Rollback

- If a single DML statement fails during execution, only that statement is rolled back.
- The Oracle server implements an implicit savepoint.
- All other changes are retained.
- The user should terminate transactions explicitly by executing a `COMMIT` or `ROLLBACK` statement.

ORACLE

A part of a transaction can be discarded through an implicit rollback if a statement execution error is detected. If a single DML statement fails during execution of a transaction, its effect is undone by a statement-level rollback. Note that in such a case, the changes made by the previous DML statements in the transaction are not discarded. They can be committed or rolled back explicitly by the user.

The Oracle server issues an implicit commit before and after any DDL statement. So, even if your DDL statement does not execute successfully, you cannot roll back the previous statement because the server issued a commit.

Terminate your transactions explicitly by executing a `COMMIT` or `ROLLBACK` statement.

Lesson Agenda

- Adding new rows in a table
  - INSERT statement
- Changing data in a table
  - UPDATE statement
- Removing rows from a table
  - DELETE statement
  - TRUNCATE statement
- Database transaction control using COMMIT, ROLLBACK, and SAVEPOINT
- Read Consistency

ORACLE

Read Consistency

- Read consistency guarantees a consistent view of the data at all times.
- Changes made by one user do not conflict with the changes made by another user.
- Read consistency ensures that, on the same data:
  - Readers do not wait for writers
  - Writers do not wait for readers
  - Writers wait for writers
- Read consistency is automatically implemented by:
  - Keeping a partial copy of the database in the undo segments.

ORACLE

Database users access the database in two ways:

- Read operations (`SELECT` statement)
- Write operations (`INSERT`, `UPDATE`, `DELETE` statements)

You need read consistency so that the following occur:

- The database reader and writer are ensured a consistent view of the data.
- Readers do not view data that is in the process of being changed.
- Writers are ensured that the changes to the database are done in a consistent manner.
- Changes made by one writer do not disrupt or conflict with the changes being made by another writer.

The purpose of read consistency is to ensure that each user sees data as it existed at the last commit, before a DML operation started.

Note: The same user can log in to different sessions. Each session maintains read consistency in the manner described above, even if they are the same users.

Read consistency is an automatic implementation. It keeps a partial copy of the database in the undo segments. The read-consistent image is constructed from the committed data in the table and the old data that is being changed and is not yet committed from the undo segment.

When an insert, update, or delete operation is made on the database, the Oracle server takes a copy of the data before it is changed and writes it to an *undo segment*.

All readers, except the one who issued the change, see the database as it existed before the changes started; they view the undo segment's "snapshot" of the data.

Before the changes are committed to the database, only the user who is modifying the data sees the database with the alterations. Everyone else sees the snapshot in the undo segment. This guarantees that readers of the data read consistent data that is not currently undergoing change.

When a DML statement is committed, the change made to the database becomes visible to anyone issuing a `SELECT` statement *after* the commit is done. The space occupied by the *old* data in the undo segment file is freed for reuse.

If the transaction is rolled back, the changes are undone:

- The original, older version of the data in the undo segment is written back to the table.
- All users see the database as it existed before the transaction began.

`FOR UPDATE` clause in a `SELECT` statement

By default, Oracle implicitly (automatically) locks many data structures for you. However, you can request specific data locks on rows or tables when you need to override default locking. Explicit locking lets you share or deny access to a table for the duration of a transaction or ensure multitable and multiquery read consistency.

When you issue a `SELECT` statement against the database to query some records, no locks are placed on the selected rows. In general, this is required because the number of records locked at any given time is (by default) kept to the absolute minimum: only those records that have been changed but not yet committed are locked. Even then, others will be able to read those records as they appeared before the change (the "before image" of the data). There are times, however, when you may want to lock a set of records even before you change them in your program. Oracle offers the `FOR UPDATE` clause of the `SELECT` statement to perform this locking. This is mainly useful when you are selecting rows within a PL/SQL code.

When you issue a `SELECT...FOR UPDATE` statement, the relational database management system (RDBMS) automatically obtains exclusive row-level locks on all the rows identified by the `SELECT` statement, thereby holding the records "for your changes only." No one else will be able to change any of these records until you perform a `ROLLBACK` or a `COMMIT`.

You can append the optional keyword `NOWAIT` to the `FOR UPDATE` clause to tell the Oracle server not to wait if the table has been locked by another user. In this case, control will be returned immediately to your program or to your SQL Developer environment so that you can perform other work, or simply wait for a period of time before trying again. Without the `NOWAIT` clause, your process will block until the table is available, when the locks are released by the other user through the issue of a `COMMIT` or a `ROLLBACK` command.

Quiz

Which of the following steps do you need to perform to copy rows from an existing table to a new table?

a. In place of the `VALUES` clause, you write a subquery.
b. Make sure the table is already created.
c. Match the number of columns and their data types in the `INSERT` clause to those in the subquery.
d. Use * if you want to copy all the rows.
e. All of the above.

Answer: e

Quiz

Q

The following statements produce the same results:

```
DELETE FROM copy_emp;
```

```
TRUNCATE TABLE copy_emp;
```

a. True
b. False

Answer: b

Quiz

Which two of the following statements, when executed, lead to an implicit `COMMIT`?

a. `CREATE TABLE` statement

b. `CREATE USER` statement

c. `INSERT` statement

d. `UPDATE` statement

ORACLE

Answer: a, b

Quiz

Q

To enable Autocommit, you should go to Tools > Preferences > Database > Advanced, and select the Autocommit check box.

a. True
b. False

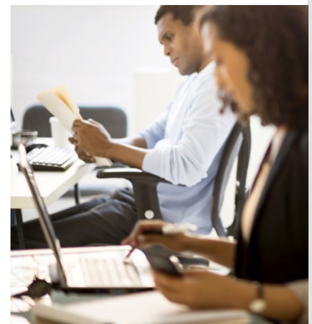
ORACLE

Answer: a

Summary

In this lesson, you should have learned how to use the following statements:

Function	Description
INSERT	Adds a new row to the table
UPDATE	Modifies existing rows in the table
DELETE	Removes existing rows from the table
TRUNCATE	Removes all rows from the table
COMMIT	Makes all pending changes permanent
SAVEPOINT	Rolls back the changes to the savepoint
ROLLBACK	Discards all pending data changes
FOR UPDATE clause in SELECT	Locks rows identified by the SELECT query



ORACLE

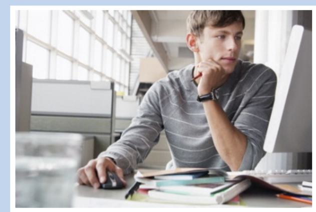
In this lesson, you should have learned how to manipulate data in the Oracle database by using the INSERT, UPDATE, DELETE, and TRUNCATE statements. You should have also learned how to control data changes by using the COMMIT, SAVEPOINT, and ROLLBACK statements. You also learned how to use the FOR UPDATE clause of the SELECT statement to lock rows for your changes only.

Remember that the Oracle server guarantees a consistent view of data at all times.

Practice 14: Overview

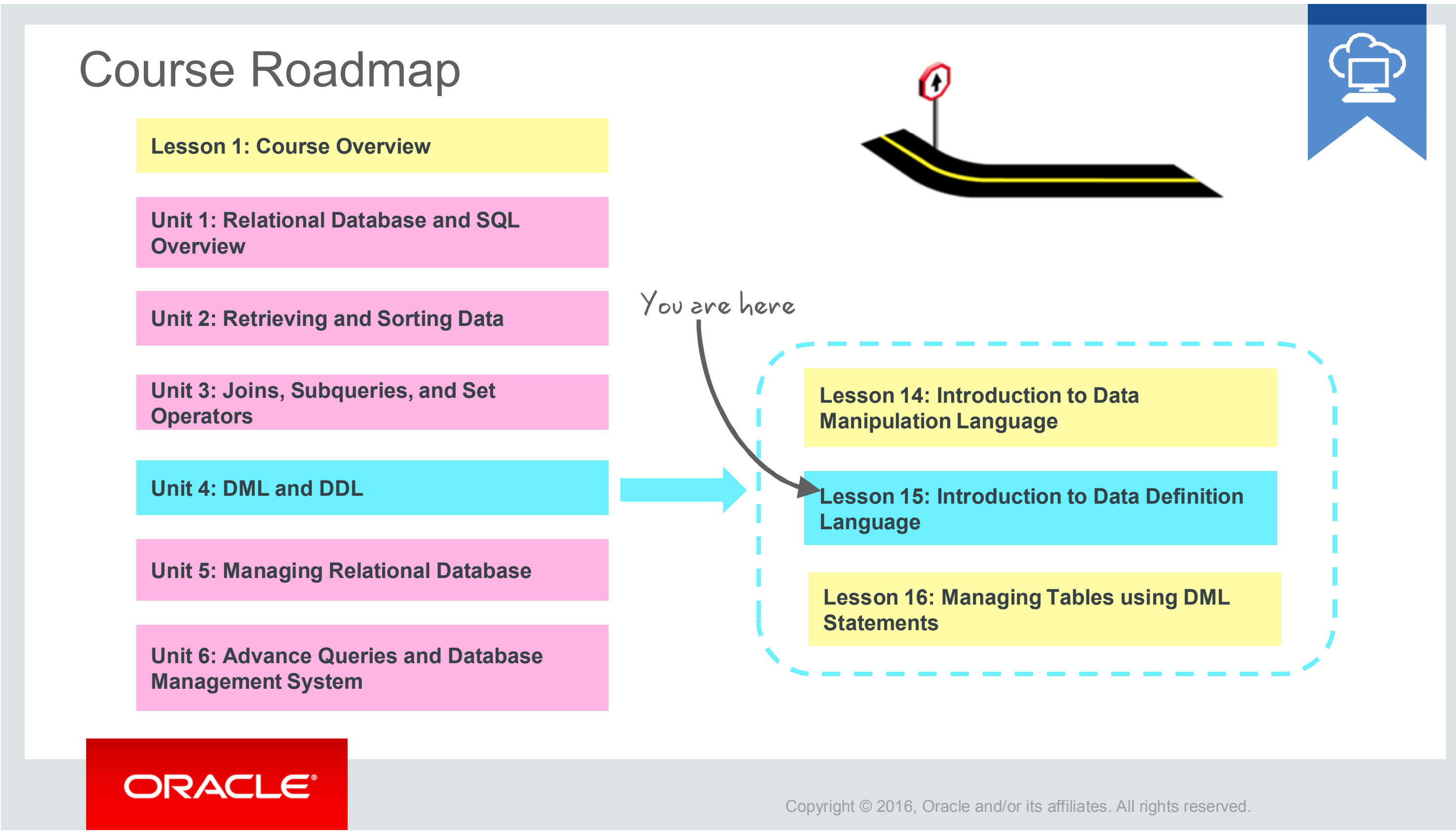
This practice covers the following topics:

- Inserting rows into a table
- Updating and deleting rows in the table
- Controlling database transactions



In this practice, you add rows to a table, update and delete data from the table, and control your transactions.


Lesson 15: Introduction to Data Definition Language
ORACLE
Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

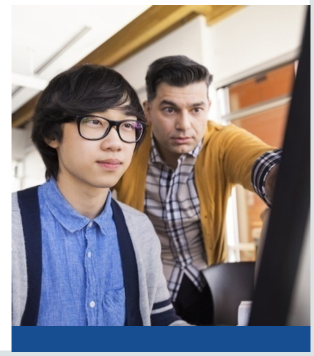


In Unit 4, you will learn about Data Manipulation Language (DML) and Data Definition Language (DDL). Using DML statements, you will learn to update and manage data in the tables. Using DDL statements, you will learn to create tables, remove tables, etc.

Objectives

After completing this lesson, you should be able to:

- Categorize the main database objects
- Review the table structure
- List the data types that are available for columns
- Create a simple table
- Explain how constraints are created at the time of table creation



ORACLE

In this lesson, you are introduced to the data definition language (DDL) statements. You learn the basics of how to create simple tables. The data types available in Oracle database are shown and schema concepts are introduced. Constraints are discussed in this lesson. Exception messages that are generated from violating constraints during DML operations are shown and explained.

Lesson Agenda

- Database objects
  - Naming rules
- `CREATE TABLE` statement
- Data types
- Overview of constraints: `NOT NULL`, `UNIQUE`, `PRIMARY KEY`, `FOREIGN KEY`, and `CHECK`

Database Objects

Object	Description
Table	Is the basic unit of storage; composed of rows
View	Logically represents subsets of data from one or more tables
Sequence	Generates numeric values
Index	Improves the performance of some queries
Synonym	Gives alternative names to objects

ORACLE

The Oracle database can contain multiple data structures. Each structure should be outlined in the database design so that it can be created during the build stage of database development.

- **Table:** Stores data
- **View:** Is a subset of data from one or more tables
- **Sequence:** Generates numeric values
- **Index:** Improves the performance of some queries
- **Synonym:** Gives alternative names to objects

Oracle Table Structures

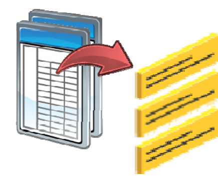
- Tables can be created at any time, even when users are using the database.
- You do not need to specify the size of a table. The size is ultimately defined by the amount of space allocated to the database as a whole. It is important, however, to estimate how much space a table will use over time.
- The table structure can be modified online.

Note: More database objects are available, but are not covered in this course.

Naming Rules

Table names and column names must:

- Begin with a letter
- Be 1–30 characters long
- Contain only A–Z, a–z, 0–9, _, $, and #
- Not duplicate the name of another object owned by the same user
- Not be an Oracle server–reserved word



You name database tables and columns according to the standard rules for naming any Oracle database object.

- Table names and column names must begin with a letter and be 1–30 characters long.
- Names must contain only the characters A–Z, a–z, 0–9, _ (underscore), $, and # (legal characters, but their use is discouraged).
- Names must not duplicate the name of another object owned by the same Oracle server user.
- Names must not be an Oracle server–reserved word.
    - You may also use quoted identifiers to represent the name of an object. A quoted identifier begins and ends with double quotation marks (“”). If you name a schema object using a quoted identifier, you must use the double quotation marks whenever you refer to that object. Quoted identifiers can be reserved words, although this is not recommended.

Naming Guidelines

Use descriptive names for tables and other database objects.

Note: Names are not case-sensitive. For example, `EMPLOYEES` is treated the same as `eMPloyees` or `eMpLOYEES`. However, quoted identifiers are case-sensitive.

For more information, see the “Schema Object Names and Qualifiers” section in the *Oracle Database SQL Language Reference* for 12*c* database.

Lesson Agenda

- Database objects
  - Naming rules
- CREATE TABLE statement
- Data types
- Overview of constraints: NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY, and CHECK

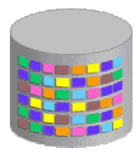
CREATE TABLE Statement

- You must have:
  - The CREATE TABLE privilege
  - A storage area

Schema is the same as the owner's name; specify the table name.

```
CREATE TABLE [schema.]table
          (column datatype [DEFAULT expr][, ...]);
```

Specify the column name, data type, and size.



Specifies default value if omitted in the INSERT statement

ORACLE

You create tables to store data by executing the SQL CREATE TABLE statement. This statement is one of the DDL statements that are a subset of the SQL statements used to create, modify, or remove Oracle Database structures. These statements have an immediate effect on the database and they also record information in the data dictionary.

To create a table, a user must have the CREATE TABLE privilege and a storage area in which to create objects. The database administrator (DBA) uses data control language (DCL) statements to grant privileges to users.

Note: The CREATE ANY TABLE privilege is needed to create a table in any schema other than the user's schema.

Creating Tables

- Create the table:

```
CREATE TABLE dept
        (deptno       NUMBER(2),
         dname        VARCHAR2(14),
         loc          VARCHAR2(13),
         create_date DATE DEFAULT SYSDATE);
```

```
table DEPT created.
```

- Confirm table creation:

```
DESCRIBE dept
```

```
Name        Null Type
----------- ---- ------------
DEPTNO           NUMBER(2)
DNAME            VARCHAR2(14)
LOC              VARCHAR2(13)
CREATE_DATE      DATE
```

ORACLE

The example in the slide creates the DEPT table with four columns: DEPTNO, DNAME, LOC, and CREATE_DATE. The CREATE_DATE column has a default value. If a value is not provided for an INSERT statement, the system date is automatically inserted.

To confirm that the table was created, run the DESCRIBE command.

Because creating a table is a DDL statement, an automatic commit takes place when this statement is executed.

Note: You can view the list of tables that you own by querying the data dictionary. Example:

```
select table_name from user_tables;
```

Using data dictionary views, you can also find information about other database objects such as views and indexes.

Lesson Agenda

- Database objects
  - Naming rules
- `CREATE TABLE` statement
- Data types
- Overview of constraints: `NOT NULL`, `UNIQUE`, `PRIMARY KEY`, `FOREIGN KEY`, and `CHECK`

Data Types

Data Type	Description
`VARCHAR2(size)`	Variable-length character data
`CHAR(size)`	Fixed-length character data
`NUMBER(p, s)`	Variable-length numeric data
`DATE`	Date and time values
`LONG`	Variable-length character data (up to 2 GB)
`CLOB`	Maximum size is (4 gigabytes - 1) * (`DB_BLOCK_SIZE`).
`RAW and LONG RAW`	Raw binary data
`BLOB`	Maximum size is (4 gigabytes - 1) * (`DB_BLOCK_SIZE` initialization parameter (8 TB to 128 TB)).
`BFILE`	Binary data stored in an external file (up to 4 GB)
`ROWID`	A base-64 number system representing the unique address of a row in its table

ORACLE

When you identify a column for a table, you need to provide a data type for the column. There are several data types available:

Data Type	Description
`VARCHAR2(`*`size`*`)`	Variable-length character data (A maximum *size* must be specified; minimum *size* is 1.) Maximum size is: • 32767 bytes if `MAX_SQL_STRING_SIZE = EXTENDED` • 4000 bytes if `MAX_SQL_STRING_SIZE = LEGACY`
`CHAR [(`*`size`*`)]`	Fixed-length character data of length *size* bytes (Default and minimum *size* is 1; maximum *size* is 2,000.)
`NUMBER [(`*`p,s`*`)]`	Number having precision *p* and scale *s* (Precision is the total number of decimal digits and scale is the number of digits to the right of the decimal point; precision can range from 1 to 38, and scale can range from –84 to 127.)
`DATE`	Date and time values to the nearest second between January 1, 4712 B.C., and December 31, 9999 A.D.

Data Type	Description
LONG	Variable-length character data (up to 2 GB)
CLOB	A character large object containing single-byte or multibyte characters. Maximum size is (4 gigabytes - 1) * (DB_BLOCK_SIZE); stores national character set data.
NCLOB	A character large object containing Unicode characters. Both fixed-width and variable-width character sets are supported, both using the database national character set. Maximum size is (4 gigabytes - 1) * (database block size); stores national character set data.
RAW(size)	Raw binary data of length *size* bytes. You must specify *size* for a RAW value. Maximum *size* is: 32767 bytes if MAX_SQL_STRING_SIZE = EXTENDED 4000 bytes if MAX_SQL_STRING_SIZE = LEGACY
LONG RAW	Raw binary data of variable length up to 2 gigabytes
BLOB	A binary large object. Maximum size is (4 gigabytes - 1) * (DB_BLOCK_SIZE initialization parameter (8 TB to 128 TB)).
BFILE	Binary data stored in an external file (up to 4 GB)
ROWID	Base 64 string representing the unique address of a row in its table. This data type is primarily for values returned by the ROWID pseudocolumn.
TIMESTAMP	This is a datetime data type. Enables storage of time as a date with fractional seconds. It stores the year, month, day, hour, minute, and second value of the DATE data type, as well as the fractional seconds value. There are several variations of this data type such as WITH TIMEZONE and WITH LOCALTIMEZONE.

Guidelines

- A LONG column is not copied when a table is created using a subquery.
- A LONG column cannot be included in a GROUP BY or an ORDER BY clause.
- Only one LONG column can be used per table.
- No constraints can be defined on a LONG column.
- You might want to use a CLOB column rather than a LONG column.

DEFAULT Option

- Specify a default value for a column when specifying the CREATE TABLE statement.

```
... hire_date DATE DEFAULT SYSDATE, ...
```

- Literal values, expressions, or SQL functions are legal values.
- Another column's name or a pseudocolumn are illegal values.
- The default data type must match the column data type.

```
CREATE TABLE hire_dates
        (id              NUMBER(8),
         hire_date DATE DEFAULT SYSDATE);
```

```
table HIRE_DATES created.
```

ORACLE

When you define a table, you can specify that a column should be given a default value by using the DEFAULT option. This option prevents null values from entering the columns when a row is inserted without a value for the column. The default value can be a literal, an expression, or a SQL function (such as SYSDATE or USER), but the value cannot be the name of another column or a pseudocolumn (such as NEXTVAL or CURRVAL). The default expression must match the data type of the column.

Consider the following examples:

```
INSERT INTO hire_dates values(45, NULL);
```

The preceding statement will insert the null value rather than the default value.

```
INSERT INTO hire_dates(id) values(35);
```

The preceding statement will insert SYSDATE for the HIRE_DATE column.

Note: In SQL Developer, click the Run Script icon or press F5 to run the DDL statements. The feedback messages will be shown in the Script Output pane.

Lesson Agenda

- Database objects
  - Naming rules
- `CREATE TABLE` statement
- Data types
- Overview of constraints: `NOT NULL`, `UNIQUE`, `PRIMARY KEY`, `FOREIGN KEY`, and `CHECK`

Including Constraints

- Constraints enforce rules at the table level.
- Constraints ensure the consistency and integrity of the database.

Constraint	Description
NOT NULL	Specifies that the column cannot contain a null value
UNIQUE	Specifies a column or combination of columns whose values must be unique for all rows in the table
PRIMARY KEY	Uniquely identifies each row of the table

ORACLE

The Oracle server uses constraints to prevent invalid data entry into tables.

You can use constraints to do the following:

- Enforce rules on the data in a table whenever a row is inserted, updated, or deleted from that table. The constraint must be satisfied for the operation to succeed.
- Prevent the dropping of a table if there are dependencies from other tables.
- Provide rules for Oracle tools, such as Oracle Developer.

Including Constraints

Constraint	Description
FOREIGN KEY	Establishes and enforces a referential integrity between the column and a column of the referenced table such that values in one table match the values in another table
CHECK	Specifies a condition that must be true

Constraint Guidelines

- You can name a constraint or the Oracle server generates a name by using the `SYS_Cn` format.
- Create a constraint at either of the following times:
  - At the same time as the creation of the table
  - After the creation of the table
- Define a constraint at the column or table level.
- View a constraint in the data dictionary.

All constraints are stored in the data dictionary. Constraints are easy to reference if you give them a meaningful name. Constraint names must follow the standard object-naming rules, except that the name cannot be the same as another object owned by the same user. If you do not name your constraint, the Oracle server generates a name with the format `SYS_Cn`, where *n* is an integer so that the constraint name is unique.

Constraints can be defined at the time of table creation or after the creation of the table. You can define a constraint at the column or table level. Functionally, a table-level constraint is the same as a column-level constraint.

For more information, see the section on “Constraints” in *Oracle Database SQL Language Reference* for 12*c* database.

Defining Constraints

- Syntax:

```
CREATE TABLE [schema.]table
      (column datatype [DEFAULT expr]
      [column_constraint],
      ...
      [table_constraint][,...]);
```

Is an integrity constraint as part of the column definition

Is an integrity constraint as part of the table definition

Functionally, a column-level constraint is the same as a table-level constraint.

NOT NULL constraints can be defined only at the column level.

ORACLE

The slide gives the syntax for defining constraints when creating a table. You can create constraints at either the column level or the table level. Constraints defined at the column level are included when the column is defined. Table-level constraints are defined at the end of the table definition, and must refer to the column or columns to which the constraint pertains in a set of parentheses. It is mainly the syntax that differentiates the two; otherwise, functionally, a column-level constraint is the same as a table-level constraint.

NOT NULL constraints can be defined only at the column level.

Constraints that apply to more than one column must be defined at the table level.

Defining Constraints

- Column-level constraint syntax:

```
column [CONSTRAINT constraint_name] constraint_type,
```

- Table-level constraint syntax:

```
column,...
  [CONSTRAINT constraint_name] constraint_type
  (column, ...),
```


The slide shows the syntax for defining the constraints at column and table level.

Defining Constraints

- Example of a column-level constraint:

```
CREATE TABLE employees(
  employee_id  NUMBER(6)
    CONSTRAINT emp_emp_id_pk PRIMARY KEY,
  first_name   VARCHAR2(20),
  ...);
```

- Example of a table-level constraint:

```
CREATE TABLE employees(
  employee_id  NUMBER(6),
  first_name   VARCHAR2(20),
  ...
  job_id       VARCHAR2(10) NOT NULL,
  CONSTRAINT emp_emp_id_pk
    PRIMARY KEY (EMPLOYEE_ID));
```

ORACLE

Constraints are usually created at the same time as the table. Constraints can be added to a table after its creation and also be temporarily disabled.

Both examples in the slide create a primary key constraint on the EMPLOYEE_ID column of the EMPLOYEES table.

1. The first example uses the column-level syntax to define the constraint.
2. The second example uses the table-level syntax to define the constraint.

NOT NULL Constraint

Ensures that null values are not permitted for the column:

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID
100	Steven	King	SKING	515.123.4567	17-JUN-11	AD_PRES	24000	(null)	(null)	90
101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-SEP-13	AD_VP	17000	(null)	100	90
102	Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-09	AD_VP	17000	(null)	100	90
103	Alexander	Hunold	AHUNOLD	590.423.4567	03-JAN-14	IT_PROG	9000	(null)	102	60
104	Bruce	Ernst	BERNST	590.423.4568	21-MAY-15	IT_PROG	6000	(null)	103	60
107	Diana	Lorentz	DLORENTZ	590.423.5567	07-FEB-15	IT_PROG	4200	(null)	103	60
124	Kevin	Mourgos	KMOURGOS	650.123.5234	16-NOV-15	ST_MAN	5800	(null)	100	50
141	Trenna	Rajs	TRAJS	650.121.8009	17-OCT-11	ST_CLERK	3500	(null)	124	50
142	Curtis	Davies	CDAVIES	650.121.2994	29-JAN-13	ST_CLERK	3100	(null)	124	50
143	Randall	Matos	RMATOS	650.121.2874	15-MAR-14	ST_CLERK	2600	(null)	124	50
144	Peter	Vargas	PVARGAS	650.121.2004	09-JUL-14	ST_CLERK	2500	(null)	124	50
149	Eleni	Zlotkey	EZLOTKEY	011.44.1344.429018	29-JAN-16	SA_MAN	10500	0.2	100	80
174	Ellen	Abel	EABEL	011.44.1644.429267	11-MAY-12	SA_REP	11000	0.3	149	80
176	Jonathon	Taylor	JTAYLOR	011.44.1644.429265	24-MAR-14	SA_REP	8600	0.2	149	80
178	Kimberely	Grant	KGRANT	011.44.1644.429263	24-MAY-15	SA_REP	7000	0.15	149	(null)
200	Jennifer	Whalen	JWHALEN	515.123.4444	17-SEP-11	AD_ASST	4400	(null)	101	10
201	Michael	Hartstein	MHARTSTE	515.123.5555	17-FEB-12	MK_MAN	13000	(null)	100	20
202	Pat	Fay	PFAY	603.123.6666	17-AUG-13	MK_REP	6000	(null)	201	20
205	Shelley	Higgins	SHIGGINS	515.123.8080	07-JUN-10	AC_MGR	12008	(null)	101	110
206	William	Gietz	WGIETZ	515.123.8181	07-JUN-10	AC_ACCOUNT	8300	(null)	205	110

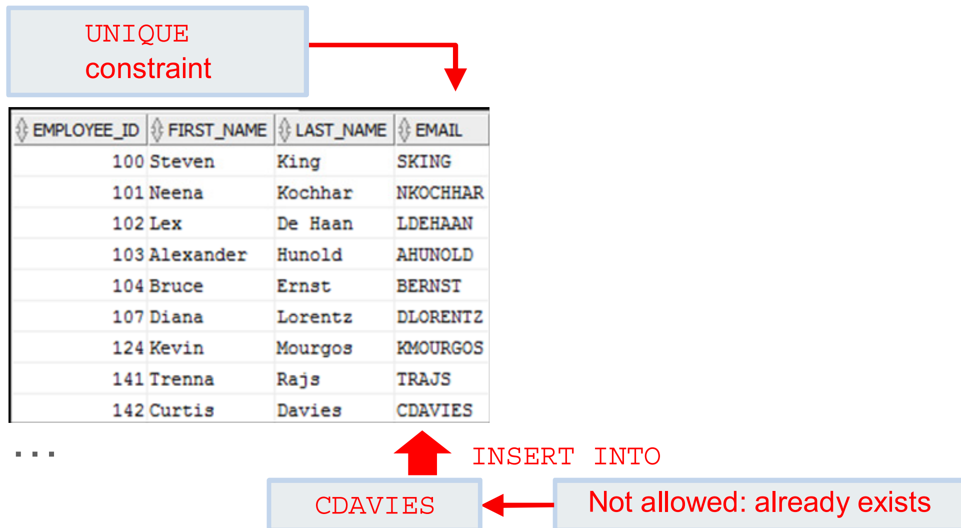
Primary Key enforces NOT NULL constraint

NOT NULL constraint

Absence of NOT NULL constraint (Any row can contain a null value for this column.)

The NOT NULL constraint ensures that the column contains no null values. Columns without the NOT NULL constraint can contain null values by default. NOT NULL constraints must be defined at the column level. In the EMPLOYEES table, the EMPLOYEE_ID column inherits a NOT NULL constraint because it is defined as a primary key; otherwise, the LAST_NAME, EMAIL, HIRE_DATE, and JOB_ID columns have the NOT NULL constraint enforced on them.

UNIQUE Constraint



EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL
100	Steven	King	SKING
101	Neena	Kochhar	NKOCHHAR
102	Lex	De Haan	LDEHAAN
103	Alexander	Hunold	AHUNOLD
104	Bruce	Ernst	BERNST
107	Diana	Lorentz	DLORENTZ
124	Kevin	Mourgos	KMOURGOS
141	Trenna	Rajs	TRAJS
142	Curtis	Davies	CDAVIES

...

A UNIQUE key integrity constraint requires that every value in a column or a set of columns (key) be unique—that is, no two rows of a table can have duplicate values in a specified column or a set of columns. The column (or set of columns) included in the definition of the UNIQUE key constraint is called the *unique key*. If the UNIQUE constraint comprises more than one column, that group of columns is called a *composite unique key*.

UNIQUE constraints enable the input of nulls unless you also define NOT NULL constraints for the same columns. In fact, any number of rows can include nulls for columns without the NOT NULL constraints because nulls are not considered equal to anything. A null in a column (or in all columns of a composite UNIQUE key) always satisfies a UNIQUE constraint.

Note: Because of the search mechanism for the UNIQUE constraints on more than one column, you cannot have identical values in the non-null columns of a partially null composite UNIQUE key constraint.

UNIQUE Constraint

Defined at either the table level or the column level:

```
CREATE TABLE employees(
    employee_id      NUMBER(6),
    last_name        VARCHAR2(25) NOT NULL,
    email            VARCHAR2(25),
    salary           NUMBER(8,2),
    commission_pct   NUMBER(2,2),
    hire_date        DATE NOT NULL,
...
    CONSTRAINT emp_email_uk UNIQUE(email));
```


UNIQUE constraints can be defined at the column level or the table level. You define the constraint at the table level when you want to create a composite unique key. A composite key is defined when a single attribute cannot uniquely identify a row. In that case, you can have a unique key that is composed of two or more columns, the combined value of which is always unique and can identify rows.

The example in the slide applies the UNIQUE constraint to the EMAIL column of the EMPLOYEES table. The name of the constraint is EMP_EMAIL_UK.

Note: The Oracle server enforces the UNIQUE constraint by implicitly creating a unique index on the unique key column or columns.

PRIMARY KEY Constraint

PRIMARY KEY

	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	10	Administration	200	1700
2	20	Marketing	201	1800
3	50	Shipping	124	1500
4	60	IT	103	1400
5	80	Sales	149	2500
6	90	Executive	100	1700
7	110	Accounting	205	1700
8	190	Contracting	(null)	1700

INSERT INTO

50 ← Not allowed: already exists

null ← Not allowed: cannot contain a null value

ORACLE

A PRIMARY KEY constraint creates a primary key for the table. Only one primary key can be created for each table. The PRIMARY KEY constraint is a column or a set of columns that uniquely identifies each row in a table. This constraint enforces the uniqueness of the column or column combination, and ensures that no column that is part of the primary key can contain a null value.

Note: Because uniqueness is part of the primary key constraint definition, the Oracle server enforces the uniqueness by implicitly creating a unique index on the primary key column or columns.

FOREIGN KEY Constraint

DEPARTMENTS: PRIMARY KEY

	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	10	Administration	200	1700
2	20	Marketing	201	1800
3	50	Shipping	124	1500
4	60	IT	103	1400
5	80	Sales	149	2500
6	90	Executive	100	1700
7	110	Accounting	205	1700
8	190	Contracting	(null)	1700

EMPLOYEES:FOREIGN KEY

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	DEPARTMENT_ID
100	Steven	King	90
101	Neena	Kochhar	90
102	Lex	De Haan	90
103	Alexander	Hunold	60
104	Bruce	Ernst	60
107	Diana	Lorentz	60
124	Kevin	Mourgos	50
141	Trenna	Rajs	50
142	Curtis	Davies	50
143	Randall	Matos	50
144	Peter	Vargas	50
149	Eleni	Zlotkey	80
174	Ellen	Abel	80
176	Jonathon	Taylor	80
178	Kimberely	Grant	(null)

INSERT INTO

Not allowed: DOES NOT EXIST IN PARENT TABLE ← DEPARTMENT_ID: 30

Allowed ← DEPARTMENT_ID: 60

The FOREIGN KEY (or referential integrity) constraint designates a column or a combination of columns as a foreign key, and establishes a relationship with a primary key or a unique key in the same table or a different table.

In the example in the slide, DEPARTMENT_ID has been defined as the foreign key in the EMPLOYEES table (dependent or child table); it references the DEPARTMENT_ID column of the DEPARTMENTS table (the referenced or parent table).

Guidelines

- A foreign key value must match an existing value in the parent table or be NULL.
- Foreign keys are based on data values and are purely logical, rather than physical, pointers.

FOREIGN KEY Constraint

Defined at either the table level or the column level:

```
CREATE TABLE employees(
    employee_id       NUMBER(6),
    last_name         VARCHAR2(25) NOT NULL,
    email             VARCHAR2(25),
    salary            NUMBER(8,2),
    commission_pct    NUMBER(2,2),
    hire_date         DATE NOT NULL,
...
    department_id     NUMBER(4),
    CONSTRAINT emp_dept_fk FOREIGN KEY (department_id)
      REFERENCES departments(department_id),
    CONSTRAINT emp_email_uk UNIQUE(email));
```

ORACLE

FOREIGN KEY constraints can be defined at the column or table level. A composite foreign key must be created by using the table-level definition.

The example in the slide defines a FOREIGN KEY constraint on the DEPARTMENT_ID column of the EMPLOYEES table, using table-level syntax. The name of the constraint is EMP_DEPT_FK.

The foreign key can also be defined at the column level, provided that the constraint is based on a single column. The syntax differs in that the keywords FOREIGN KEY do not appear. Example:

```
CREATE TABLE employees
(...
department_id NUMBER(4) CONSTRAINT emp_deptid_fk
REFERENCES departments(department_id),
...
)
```

FOREIGN KEY Constraint: Keywords

- FOREIGN KEY: Defines the column in the child table at the table-constraint level
- REFERENCES: Identifies the table and column in the parent table
- ON DELETE CASCADE: Deletes the dependent rows in the child table when a row in the parent table is deleted
- ON DELETE SET NULL: Converts dependent foreign key values to null

The foreign key is defined in the child table, and the table containing the referenced column is the parent table. The foreign key is defined using a combination of the following keywords:

- `FOREIGN KEY` is used to define the column in the child table at the table-constraint level.
- `REFERENCES` identifies the table and the column in the parent table.
- `ON DELETE CASCADE` indicates that when a row in the parent table is deleted, the dependent rows in the child table are also deleted.
- `ON DELETE SET NULL` indicates that when a row in the parent table is deleted, the foreign key values are set to null.

The default behavior is called the *restrict rule*, which disallows the update or deletion of referenced data.

Without the `ON DELETE CASCADE` or the `ON DELETE SET NULL` options, a row in the parent table cannot be deleted if it is referenced in the child table. Additionally, these keywords cannot be used in column-level syntax.

CHECK Constraint

- Defines a condition that each row must satisfy
- Cannot reference columns from other tables

```
..., salary       NUMBER(2)
     CONSTRAINT emp_salary_min
            CHECK (salary > 0),...
```

The CHECK constraint defines a condition that each row must satisfy. To satisfy the constraint, each row in the table must make the condition either TRUE or unknown (due to a null).

The condition can use the same constructs as the query conditions except the queries that refer to other values in other rows.

A single column can have multiple CHECK constraints that refer to the column in its definition. There is no limit to the number of CHECK constraints that you can define on a column.

CHECK constraints can be defined at the column level or the table level.

```
CREATE TABLE employees
   (...
    salary NUMBER(8,2) CONSTRAINT emp_salary_min
                       CHECK (salary > 0),
   ...
```

CREATE TABLE: Example

```
CREATE TABLE teach_emp (
         empno      NUMBER(5) PRIMARY KEY,
         ename      VARCHAR2(15) NOT NULL,
         job        VARCHAR2(10),
         mgr        NUMBER(5),
         hiredate   DATE DEFAULT (sysdate),
         photo      BLOB,
         sal        NUMBER(7,2),
         deptno     NUMBER(3) NOT NULL
                     CONSTRAINT admin_dept_fkey REFERENCES
                      departments(department_id));
```

ORACLE

Violating Constraints

```
UPDATE employees
SET    department_id = 55
WHERE  department_id = 110;
```

```
Error starting at line : 1 in command -
UPDATE employees
SET    department_id = 55
WHERE  department_id = 110
Error report -
SQL Error: ORA-02291: integrity constraint (ORA01.EMP_DEPT_FK) violated - parent key not found
02291. 00000 - "integrity constraint (%s.%s) violated - parent key not found"
*Cause:    A foreign key value has no matching primary key value.
*Action:   Delete the foreign key or add a matching primary key.
```

ORACLE

When you have constraints in place on columns, an error is returned if you try to violate the constraint rule. For example, if you try to update a record with a value that is tied to an integrity constraint, an error is returned.

In the example in the slide, department 55 does not exist in the parent table, `DEPARTMENTS`, and so you receive the "parent key not found" violation `ORA-02291`.

Violating Constraints

You cannot delete a row containing a primary key that is used as a foreign key in another table.

```
DELETE FROM departments
WHERE department_id = 60;
```

```
DELETE FROM departments
WHERE department_id = 60
Error report -
SQL Error: ORA-02292: integrity constraint (ORA01.EMP_DEPT_FK) violated - child record found
02292. 00000 - "integrity constraint (%s.%s) violated - child record found"
*Cause:    attempted to delete a parent key value that had a foreign
           dependency.
*Action:   delete dependencies first then parent or disable constraint.
```

ORACLE

If you attempt to delete a record with a value that is tied to an integrity constraint, an error is returned.

The example in the slide tries to delete department 60 from the `DEPARTMENTS` table, but it results in an error because that department number is used as a foreign key in the `EMPLOYEES` table. If the parent record that you attempt to delete has child records, you receive the “child record found” violation `ORA-02292`.

The following statement works because there are no employees in department 70:

```
DELETE FROM  departments
WHERE department_id = 70;
```

Quiz

Q

While creating a table, you can specify the default value for the columns, which is the value to be inserted if the value is omitted in the INSERT statement.

a. True
b. False

ORACLE

Answer: a

Quiz

Which constraint can only be specified at the column level?

a. `PRIMARY KEY`
b. `NOT NULL`
c. `CHECK`
d. `FOREIGN KEY`

Answer: b

Quiz

Q

A foreign key value must match an existing value in the parent table or be `NULL`.

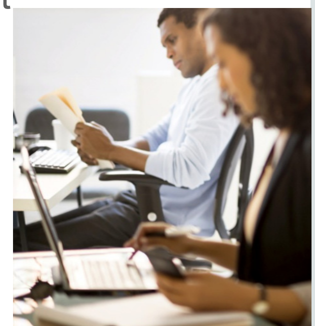
a. True
b. False

Answer: a

Summary

In this lesson, you should have learned how to:

- Categorize the main database objects
- Review the table structure
- List the data types that are available for columns
- Create a simple table by using the `CREATE TABLE` statement
- Explain how constraints are added at the time of table creation



In this lesson, you should have learned how to use the `CREATE TABLE` statement to create a table and include constraints.

Practice 15: Overview

This practice covers the following topics:

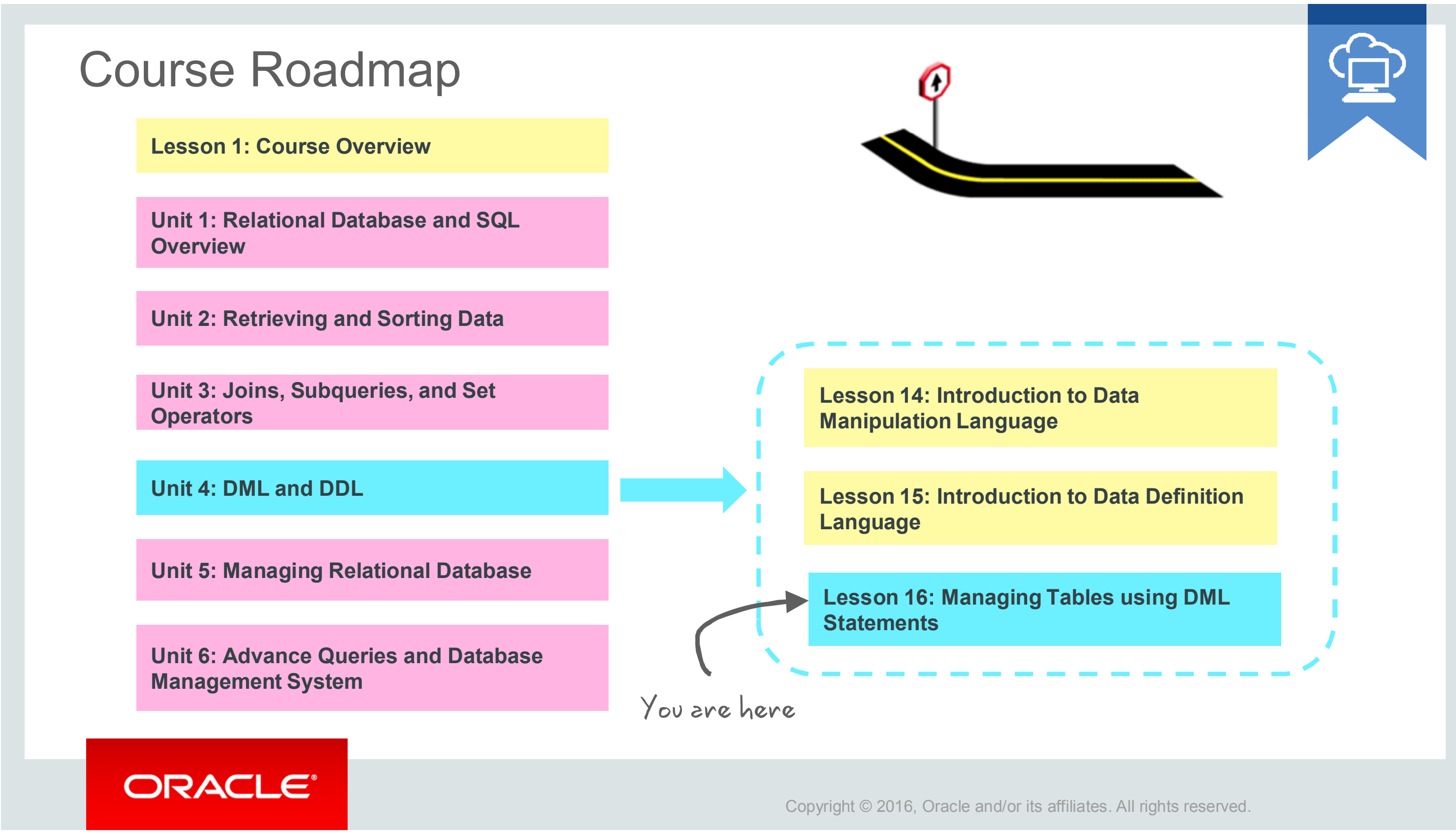
- Creating new tables
- Verifying that tables exist
- Defining various table and column constraints

ORACLE

You create new tables by using the `CREATE TABLE` statement and confirm that the new tables were added to the database.

Note: For all DDL and DML statements, click the Run Script icon (or press F5) to execute the query in SQL Developer. Thus, you get to see the feedback messages in the Script Output pane. For `SELECT` queries, continue to click the Execute Statement icon or press F9 to get the formatted output in the Results pane.


Lesson 16: Managing Tables Using DML Statements
ORACLE
Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

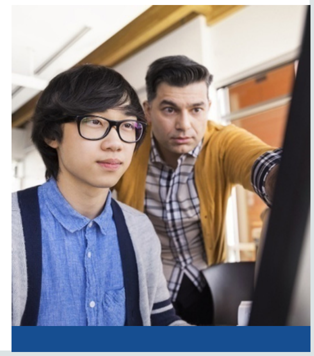


In Unit 4, you will learn about Data Manipulation Language (DML) and Data Definition Language (DDL). Using DML statements, you will learn to update and manage data in the tables. Using DDL statements, you will learn to create tables, remove tables, etc.

Objectives

After completing this lesson, you should be able to:

- Create a table using the `CREATE TABLE AS` statement
- Add, modify, and drop columns by using the `ALTER TABLE` statement
- Drop tables using the `DROP TABLE` statement



In this lesson, you learn more about managing database objects. You learn how to create a table using a subquery. You learn the basics of altering and dropping tables.

Lesson Agenda

- Creating a table using a subquery
- `ALTER TABLE` statement
- `DROP TABLE` statement

Creating a Table Using a Subquery

- Create a table and insert rows by combining the `CREATE TABLE` statement and the `AS subquery` option.

```
CREATE TABLE table
          [(column, column...)]
AS subquery;
```

SELECT statement that defines the set of rows to be inserted into the new table

- Match the number of specified columns to the number of subquery columns.
- Define columns with column names and default values.

ORACLE

A second method for creating a table is to apply the `AS` *`subquery`* clause, which both creates the table and inserts rows returned from a subquery.

Guidelines

- The table is created with the specified column names, and the rows retrieved by the `SELECT` statement are inserted into the table.
- The column definition can contain only the column name and default value.
- If column specifications are given, the number of columns must equal the number of columns in the subquery `SELECT` list.
- If no column specifications are given, the column names of the table are the same as the column names in the subquery. To specify different column names, enclose them in a parenthesis following the table name. For example:
  - `CREATE TABLE  emp (id, name) AS` *subquery*
- The column data type definitions and the `NOT NULL` constraint are passed to the new table. Note that only the explicit `NOT NULL` constraint will be inherited. The `PRIMARY KEY` column will not pass the `NOT NULL` feature to the new column. Any other constraint rules are not passed to the new table. However, you can add constraints in the column definition.

Creating a Table Using a Subquery

```
CREATE TABLE     dept80
  AS
    SELECT  employee_id, last_name,
            salary*12 ANNSAL,
            hire_date
    FROM    employees
    WHERE   department_id = 80;
```

```
table DEPT80 created.
```

```
DESCRIBE dept80
```

```
Name        Null     Type
----------- -------- ------------
EMPLOYEE_ID          NUMBER(6)
LAST_NAME   NOT NULL VARCHAR2(25)
ANNSAL               NUMBER
HIRE_DATE   NOT NULL DATE
```


The example in the slide creates a table named DEPT80, which contains details of all the employees working in department 80. Notice that the data for the DEPT80 table comes from the EMPLOYEES table.

You can verify the existence of a database table and check the column definitions by using the DESCRIBE command.

However, be sure to provide a column alias when selecting an expression. The expression SALARY*12 is given the alias ANNSAL. Without the alias, the following error is generated:

```
Error starting at line 1 in command:
CREATE TABLE  dept80
  AS
    SELECT  employee_id, last_name,
            salary*12,
            hire_date
    FROM    employees
    WHERE   department_id = 80
Error at Command Line:4 Column:18
Error report:
SQL Error: ORA-00998: must name this expression with a column alias
00998. 00000 -  "must name this expression with a column alias"
*Cause:
*Action:
```

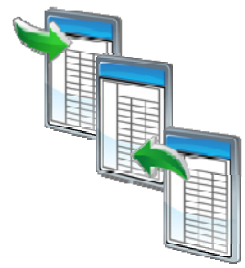
Lesson Agenda

- Creating a table using a subquery
- `ALTER TABLE` statement
- `DROP TABLE` statement

ALTER TABLE Statement

Use the ALTER TABLE statement to:

- Add a new column
- Modify an existing column definition
- Define a default value for a new column
- Drop a column
- Change table to read-only status



After you create a table, you may need to change the table structure for any of the following reasons:

- You omitted a column.
- Your column definition or its name needs to be changed.
- You need to remove columns.
- You want to put the table into read-only mode.

You can do this by using the ALTER TABLE statement.

ALTER TABLE Statement

Use the ALTER TABLE statement to add, modify, or drop columns:

Specify the default value for a column.

```
ALTER TABLE table
ADD|MODIFY    (column datatype [DEFAULT expr]
              [, column datatype]...)
DROP (column [, column] …);
```

ADD, MODIFY, or DROP specify the type of modification.

When adding or modifying, specify the column and its data type and size.

You can add columns to a table, modify columns, and drop columns from a table by using the ALTER TABLE statement.

Adding a Column

- You use the ADD clause to add columns:

```
ALTER TABLE dept80
ADD  (job_id VARCHAR2(9));
```

- The new column becomes the last column:

	EMPLOYEE_ID	LAST_NAME	ANNSAL	HIRE_DATE	JOB_ID
1	149	Zlotkey	126000	29-JAN-16	(null)
2	174	Abel	132000	11-MAY-12	(null)
3	176	Taylor	103200	24-MAR-14	(null)

Guidelines for Adding a Column

- You can add or modify columns.
- You cannot specify where the column should appear. The new column becomes the last column.

The example in the slide adds a column named JOB_ID to the DEPT80 table. The JOB_ID column becomes the last column in the table.

Note: If a table already contains rows when a column is added, the new column is initially null or takes the default value for all the rows. You can add a mandatory NOT NULL column to a table that contains data in the other columns only if you specify a default value. You can add a NOT NULL column to an empty table without the default value.

Modifying a Column

- You can change a column's data type, size, and default value.

```
ALTER TABLE dept80
MODIFY(last_name VARCHAR2(30));
```

- A change to the default value affects only subsequent insertions to the table.

You can modify a column definition by using the ALTER TABLE statement with the MODIFY clause. Column modification can include changes to a column's data type, size, and default value.

Guidelines

- You can increase the width or precision of a numeric column.
- You can increase the width of character columns.
- You can decrease the width of a column if:
  - The column contains only null values
  - The table has no rows
  - The decrease in column width is not less than the existing values in that column
- You can change the data type if the column contains only null values. The exception to this is CHAR-to-VARCHAR2 conversions, which can be done with data in the columns.
- You can convert a CHAR column to the VARCHAR2 data type or convert a VARCHAR2 column to the CHAR data type only if the column contains null values or if you do not change the size.
- A change to the default value of a column affects only subsequent insertions to the table.

Dropping a Column

Use the DROP clause to drop columns that you no longer need from the table:

```
ALTER TABLE  dept80
DROP (job_id);
```

	EMPLOYEE_ID	LAST_NAME	ANNSAL	HIRE_DATE
1	149	Zlotkey	126000	29-JAN-16
2	174	Abel	132000	11-MAY-12
3	176	Taylor	103200	24-MAR-14

ORACLE

You can drop a column from a table by using the ALTER TABLE statement with the DROP clause.

Guidelines

- The column may or may not contain data.
- Using the ALTER TABLE DROP statement, only one column can be dropped at a time.
- The table must have at least one column remaining in it after it is altered.
- After a column is dropped, it cannot be recovered.
- A primary key that is referenced by another column cannot be dropped, unless the cascade option is added.

Dropping a column can take a while if the column has a large number of values. In this case, it may be better to set it as unused and drop it when there are fewer users on the system to avoid extended locks. You use the SET UNUSED option to mark one or more columns as unused. You use the DROP UNUSED COLUMNS option to remove the columns that are marked as unused.

DROP UNUSED COLUMNS Option

You can use this statement when you want to reclaim the extra disk space from the unused columns in the table. If the table contains no unused columns, the statement returns with no errors.

```
ALTER TABLE  dept80
SET   UNUSED (last_name);
ALTER TABLE  dept80
DROP  UNUSED COLUMNS;
```

Note: Certain columns can never be dropped, such as columns that form part of the partitioning key of a partitioned table or columns that form part of the PRIMARY KEY of an index-organized table.

Read-Only Tables

You can use the ALTER TABLE syntax to:

- Put a table in read-only mode, which prevents DDL or DML changes during table maintenance
- Put the table back into read/write mode

```
ALTER TABLE employees READ ONLY;

-- perform table maintenance and then
-- return table back to read/write mode

ALTER TABLE employees READ WRITE;
```


You can specify READ ONLY to place a table in read-only mode. When the table is in READ ONLY mode, you cannot issue any DML statements that affect the table or any SELECT ... FOR UPDATE statements. You can issue DDL statements as long as they do not modify any data in the table. Operations on indexes associated with the table are allowed when the table is in READ ONLY mode.

Specify READ/WRITE to return a read-only table to read/write mode.

Note: You can drop a table that is in READ ONLY mode. The DROP command is executed only in the data dictionary, so access to the table contents is not required. The space used by the table will not be reclaimed until the tablespace is made read/write again, and the required changes can be made to the block segment headers, and so on.

Lesson Agenda

- Creating a table using a subquery
- `ALTER TABLE` statement
- `DROP TABLE` statement

Dropping a Table

- Moves a table to the recycle bin
- Removes the table and all its data entirely if the `PURGE` clause is specified
- Invalidates dependent objects and removes object privileges on the table

```
DROP TABLE dept80;
```

```
Table DEPT80 dropped.
```


The `DROP TABLE` statement moves a table to the recycle bin or removes the table and all its data from the database entirely. Unless you specify the `PURGE` clause, the `DROP TABLE` statement does not result in space being released back to the tablespace for use by other objects, and the space continues to count toward the user's space quota. Dropping a table invalidates the dependent objects and removes object privileges on the table.

When you drop a table, the database loses all the data in the table and all the indexes associated with it.

Syntax

```
DROP TABLE table [PURGE]
```

In the syntax, *`table`* is the name of the table.

Guidelines

- All data is deleted from the table.
- Any views and synonyms remain, but are invalid.
- Any pending transactions are committed.
- Only the creator of the table or a user with the `DROP ANY TABLE` privilege can remove a table.

Note: You cannot rollback the `DROP TABLE` statement. Use the `FLASHBACK TABLE` statement to restore a dropped table from the recycle bin.

Quiz

What does the following statement do?

```
CREATE TABLE     dept20
  AS
    SELECT  employee_id, last_name,
            salary*0.5 BONUS,
            hire_date
    FROM    employees
    WHERE   department_id = 20;
```

a. Creates a table named DEPT20, which contains details of all the employees working in department 20, with their bonus salary details
b. Gives an error because the table does not exist
c. Creates the DEPT20 table and inserts the query statement as the data

Answer: a

Quiz

Which one of the following statements describes the difference between `ALTER TABLE DROP` and `DROP TABLE`?

a. `ALTER TABLE DROP` is used to drop an unused column while `DROP TABLE` is used to delete the whole table.

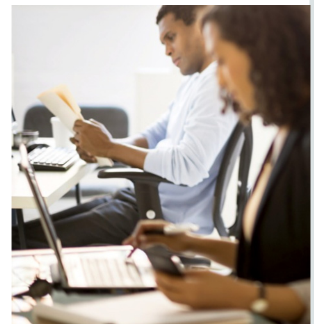
b. `ALTER TABLE DROP` drops the data from a column while `DROP TABLE` invalidates a table.

Answer: a

Summary

In this lesson, you should have learned how to:

- Create a table by using a subquery
- Use the `ALTER TABLE` and `DROP TABLE` statements



In this lesson, you should have learned how to:

- Create a table using a subquery
- Add, modify, and delete a column from a table
- Drop a table

Practice 16: Overview

This practice covers the following topics:

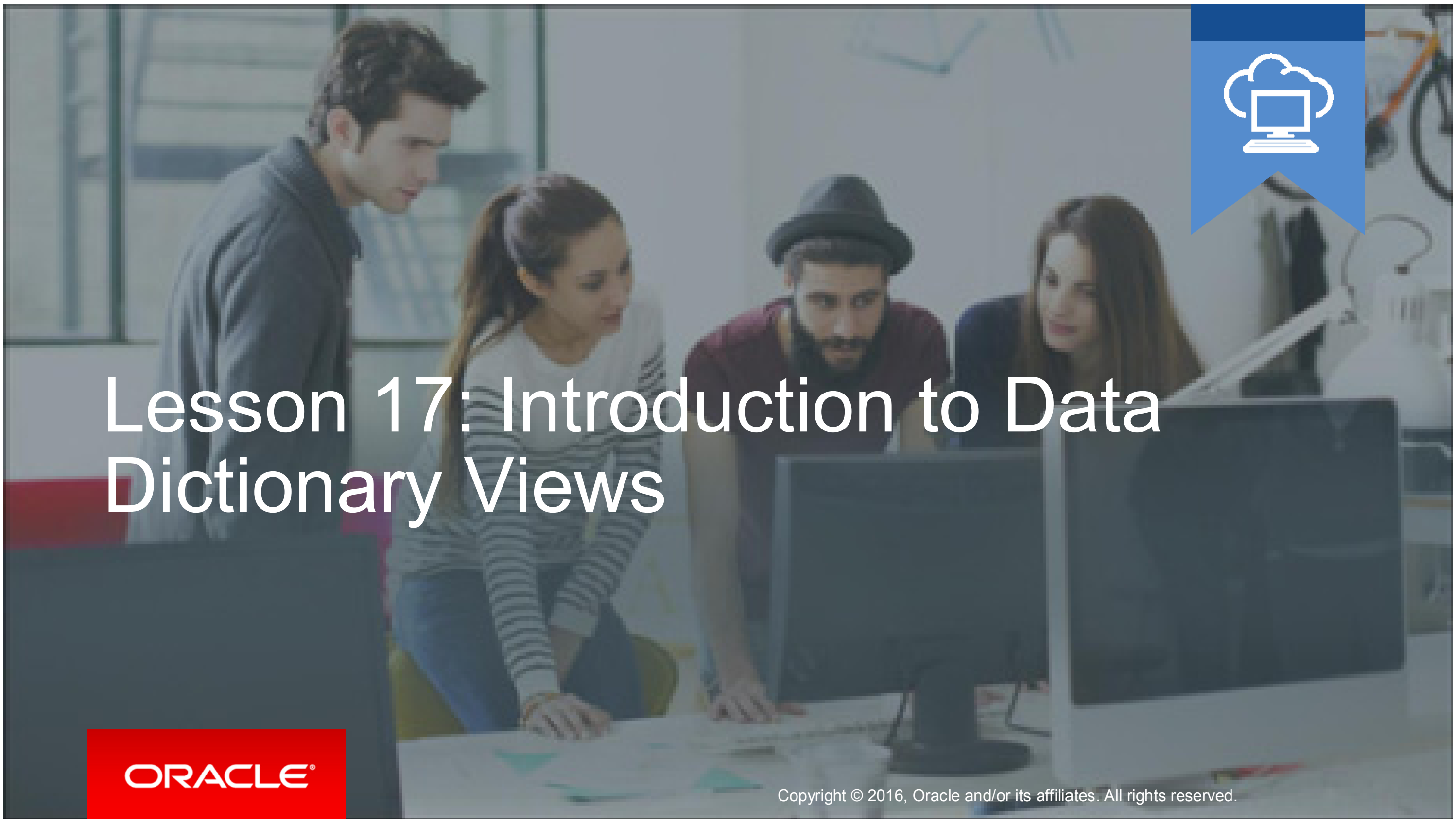
- Creating a new table by using the `CREATE TABLE AS` syntax
- Verifying that tables exist
- Altering tables
  - Adding columns
  - Dropping columns
- Setting a table to read-only status
- Dropping tables

ORACLE

In this practice, you create new tables by using the `CREATE TABLE AS` statement. You also alter and drop tables.

Note: For all DDL and DML statements, click the Run Script icon (or press F5) to execute the query in SQL Developer. Thus, you get to see the feedback messages in the Script Output pane. For `SELECT` queries, continue to click the Execute Statement icon or press F9 to get the formatted output in the Results pane.

Lesson 17: Introduction to Data Dictionary Views



Course Roadmap

- Lesson 1: Course Overview
- Unit 1: Relational Database and SQL Overview
- Unit 2: Retrieving and Sorting Data
- Unit 3: Joins, Subqueries, and Set Operators
- Unit 4: DML and DDL
- Unit 5: Managing Relational Database
- Unit 6: Advance Queries and Database Management System

You are here

- Lesson 17: Introduction to Data Dictionary Views
- Lesson 18: Creating Views
- Lesson 19: Creating Sequences, Synonyms and Indexes
- Lesson 20: Managing Constraints, Temporary Tables and External Tables

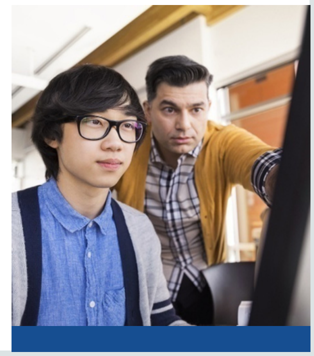
ORACLE

In Unit 5, you will be introduced to views. You will also learn to query data dictionary views. You will learn to create sequences, synonyms and indexes. You will also learn to manage constraints and tables.

Objectives

After completing this lesson, you should be able to:

- Use the data dictionary views to research data on your objects
- Query various data dictionary views



ORACLE

In this lesson, you are introduced to data dictionary views. You learn that the dictionary views can be used to retrieve metadata and create reports about your schema objects.

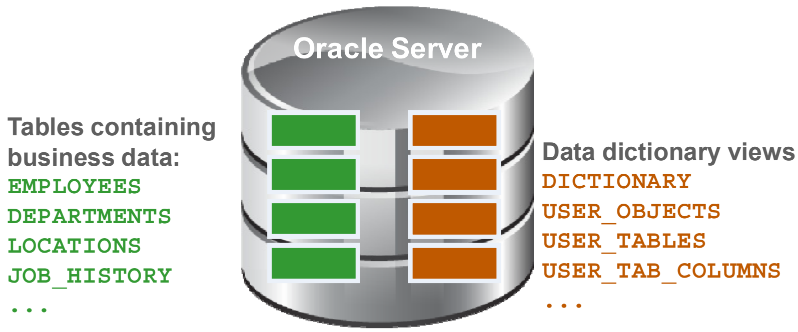
Lesson Agenda

- Introduction to data dictionary
- Querying the dictionary views for the following:
  - Table information
  - Column information
  - Constraint information
- Adding a comment to a table and querying the dictionary views for comment information

ORACLE

Data Dictionary

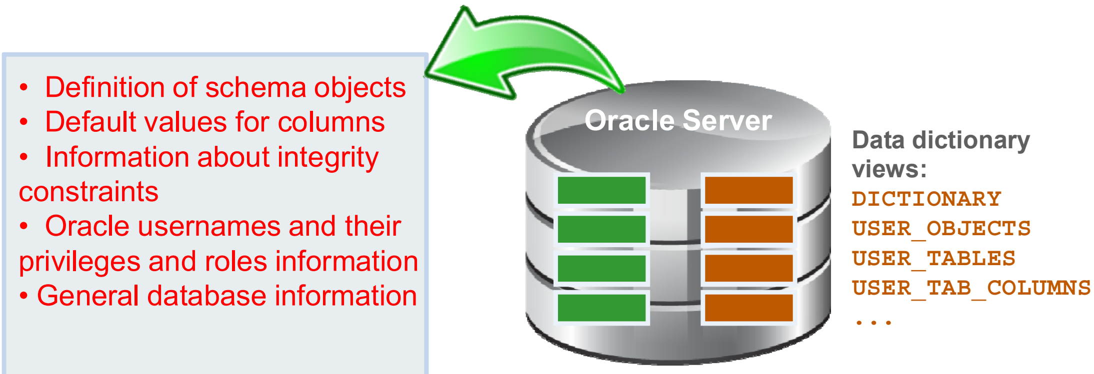
- Users create and maintain business data in user tables.
- Oracle Database creates and maintains data *about* users' data in a collection of tables or views known as the Data Dictionary.
- The data dictionary is structured in tables and views, just like other database data.



User tables are tables created by the users and contain business data, such as `EMPLOYEES`. There is another collection of tables and views in the Oracle database known as the *data dictionary*. This collection is created and maintained by the Oracle Server and contains information about the database. The data dictionary is structured in tables and views, just like other database data.

Data Dictionary

- Important tool for end users, application developers, and database administrators to find information about their data
- You use SQL statements to query the read-only dictionary tables and views.



ORACLE

Not only is the data dictionary central to every Oracle database, but it is also an important tool for all users, from end users to application designers and database administrators.

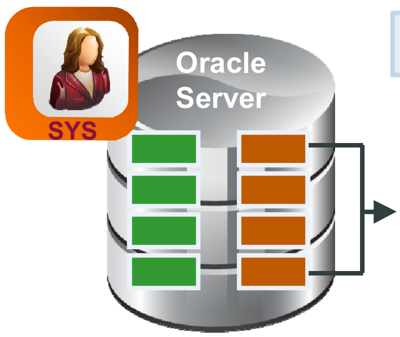
You use SQL statements to access the data dictionary. Because the data dictionary is read-only, you can issue only queries against its tables and views.

You can query the dictionary views that are based on the dictionary tables to find information such as:

- Definitions of all schema objects in the database (tables, views, indexes, synonyms, sequences, procedures, functions, packages, triggers, and so on)
- Default values for columns
- Integrity constraint information
- Names of Oracle users
- Privileges and roles that each user has been granted
- Other general database information

Data Dictionary Structure

- The Oracle server writes and reads from a set of base tables.
- Users access the views that provide useful information decoded from these base tables; views hide the complexity from the users.



- SYS owns the data dictionary.

Consists of:
- **Base tables**
- **User-accessible views**

ORACLE

Underlying base tables store information about the associated database. Only the Oracle Server should write to and read from these tables. You rarely access them directly.

There are several views that summarize and display the information stored in the base tables of the data dictionary. These views decode the base table data into useful information (such as user or table names) using joins and `WHERE` clauses to simplify the information. Most users are given access to the views rather than the base tables.

The Oracle user `SYS` owns all base tables and user-accessible views of the data dictionary. No Oracle user should *ever* alter (`UPDATE`, `DELETE`, or `INSERT`) any rows or schema objects contained in the `SYS` schema, because such activity can compromise data integrity.

Data Dictionary Structure

View naming convention:

View Prefix	Purpose
USER	User's view (what is in your schema; what you own)
ALL	Expanded user's view (what you can access)
DBA	Database administrator's view (what is in everyone's schemas)
V$	Performance-related data

ORACLE

The data dictionary consists of sets of views. In many cases, a set consists of three views containing similar information and distinguished from each other by their prefixes. For example, there is a view named `USER_OBJECTS`, another named `ALL_OBJECTS`, and a third named `DBA_OBJECTS`.

These three views contain similar information about objects in the database, except that the scope is different. `USER_OBJECTS` contains information about objects that you own or you created. `ALL_OBJECTS` contains information about all objects to which you have access. `DBA_OBJECTS` contains information about all objects that are owned by all users. For views that are prefixed with `ALL` or `DBA`, there is usually an additional column in the view named `OWNER` to identify who owns the object.

There is also a set of views that is prefixed with `v$`. These views are dynamic in nature and hold information about performance. Dynamic performance tables are not true tables, and they should not be accessed by most users. However, database administrators can query and create views on the tables and grant access to those views to other users. This course does not go into details about these views.

How to Use the Dictionary Views

Start with `DICTIONARY`. It contains the names and descriptions of the dictionary tables and views.

```
DESCRIBE DICTIONARY
```

```
Name       Null Type
---------- ---- --------------
TABLE_NAME      VARCHAR2(128)
COMMENTS        VARCHAR2(4000)
```

```
SELECT *
FROM   dictionary
WHERE  table_name = 'USER_OBJECTS';
```

	TABLE_NAME	COMMENTS
1	USER_OBJECTS	(null)

ORACLE

To familiarize yourself with the dictionary views, you can use the dictionary view named DICTIONARY. It contains the name and short description of each dictionary view to which you have access.

You can write queries to search for information about a particular view name, or you can search the COMMENTS column for a word or phrase.

The SELECT statement retrieves information about the dictionary view named USER_OBJECTS. The USER_OBJECTS view contains information about all the objects that you own.

You can write queries to search the COMMENTS column for a word or phrase. For example, the following query returns the names of the tables that you have access to in which the COMMENTS column contains the word *employees*:

```
SELECT table_name,comments
FROM user_tab_comments
WHERE LOWER(comments) LIKE '%employees%';
```

Note: The names in the data dictionary are in uppercase.

You will learn to add comments to a table or column by using the COMMENT statement later in this lesson.

USER_OBJECTS and ALL_OBJECTS Views

USER_OBJECTS:

- Query USER_OBJECTS to see all the objects that you own.
- Query USER_OBJECTS to obtain a listing of all object names and types in your schema, as well as the following information:
  - Date created
  - Date of last modification
  - Status (valid or invalid)

ALL_OBJECTS:

- Query ALL_OBJECTS to see all the objects to which you have access.

ORACLE

You can query the USER_OBJECTS view to see the names and types of all the objects in your schema. There are several columns in this view:

- **OBJECT_NAME:** Name of the object
- **OBJECT_ID:** Dictionary object number of the object
- **OBJECT_TYPE:** Type of object (such as TABLE, VIEW, INDEX, SEQUENCE)
- **CREATED:** Time stamp for the creation of the object
- **LAST_DDL_TIME:** Time stamp for the last modification of the object resulting from a data definition language (DDL) command
- **STATUS:** Status of the object (VALID, INVALID, or N/A)
- **GENERATED:** Was the name of this object system-generated? (Y|N)

Note: This is not a complete listing of the columns. For a complete listing, see "USER_OBJECTS" in the *Oracle® Database Reference 12c Release 1*.

You can also query the ALL_OBJECTS view to see a listing of all objects to which you have access.

USER_OBJECTS View

```
SELECT object_name, object_type, created, status
FROM   user_objects
ORDER BY object_type;
```

EMP_EMP_ID_PK	INDEX	10-FEB-16	VALID
DEPT_LOCATION_IX	INDEX	10-FEB-16	VALID
EMP_NAME_IX	INDEX	10-FEB-16	VALID
JHIST_EMP_ID_ST_DATE_PK	INDEX	10-FEB-16	VALID
EMP_DEPARTMENT_IX	INDEX	10-FEB-16	VALID
DEPARTMENTS_SEQ	SEQUENCE	10-FEB-16	VALID
LOCATIONS_SEQ	SEQUENCE	10-FEB-16	VALID
EMPLOYEES_SEQ	SEQUENCE	10-FEB-16	VALID
JOB_HISTORY	TABLE	10-FEB-16	VALID
JOB_GRADES	TABLE	10-FEB-16	VALID
EMPLOYEES	TABLE	10-FEB-16	VALID
JOBS	TABLE	10-FEB-16	VALID
DEPARTMENTS	TABLE	10-FEB-16	VALID
LOCATIONS	TABLE	10-FEB-16	VALID
COUNTRIES	TABLE	10-FEB-16	VALID
REGIONS	TABLE	10-FEB-16	VALID
EMP_DETAILS_VIEW	VIEW	10-FEB-16	VALID

...

	TABLE_NAME	TABLE_TYPE
1	REGIONS	TABLE
2	COUNTRIES	TABLE
3	LOCATIONS	TABLE
4	LOCATIONS_SEQ	SEQUENCE
5	DEPARTMENTS	TABLE
6	DEPARTMENTS_SEQ	SEQUENCE
7	JOBS	TABLE
8	EMPLOYEES	TABLE
9	EMPLOYEES_SEQ	SEQUENCE
10	JOB_HISTORY	TABLE
11	EMP_DETAILS_VIEW	VIEW
12	JOB_GRADES	TABLE

The CAT view displays names of all your INDEX, TABLE, CLUSTER, VIEW, SYNONYM, SEQUENCE, or UNDEFINED objects.

ORACLE

The example shows the names, types, dates of creation, and status of all objects that are owned by this user.

The OBJECT_TYPE column holds the values of either TABLE, VIEW, SEQUENCE, INDEX, PROCEDURE, FUNCTION, PACKAGE, or TRIGGER.

The STATUS column holds a value of VALID, INVALID, or N/A. Although tables are always valid, the views, procedures, functions, packages, and triggers may be invalid.

The CAT View

For a simplified query and output, you can query the CAT view. This view contains only two columns: TABLE_NAME and TABLE_TYPE. It provides the names of all your INDEX, TABLE, CLUSTER, VIEW, SYNONYM, SEQUENCE, or UNDEFINED objects.

Note: CAT is a synonym for USER_CATALOG—a view that lists tables, views, synonyms, and sequences owned by the user.

Lesson Agenda

- Introduction to data dictionary
- Querying the dictionary views for the following:
  - Table information
  - Column information
  - Constraint information
- Adding a comment to a table and querying the dictionary views for comment information

ORACLE

Table Information

USER_TABLES: Provides information about your tables

```
DESCRIBE user_tables
```

```
TABLE_NAME              NOT NULL VARCHAR2(128)
TABLESPACE_NAME                  VARCHAR2(30)
CLUSTER_NAME                     VARCHAR2(128)
IOT_NAME                         VARCHAR2(128)
STATUS                           VARCHAR2(8)
PCT_FREE                         NUMBER
PCT_USED                         NUMBER
INI_TRANS                        NUMBER
MAX_TRANS                        NUMBER
INITIAL_EXTENT                   NUMBER
NEXT_EXTENT                      NUMBER
MIN_EXTENTS                      NUMBER
MAX_EXTENTS                      NUMBER
...
```

```
SELECT table_name
FROM   user_tables;
```

	TABLE_NAME
1	REGIONS
2	COUNTRIES
3	LOCATIONS
4	DEPARTMENTS
5	JOBS
6	EMPLOYEES
7	JOB_HISTORY
8	JOB_GRADES

The TAB view is a synonym of the USER_TABLES view.

ORACLE

You can use the USER_TABLES view to obtain the names of all your tables. The USER_TABLES view contains information about your tables. In addition to providing the table name, it contains detailed information about the storage.

The TABS view is a synonym of the USER_TABLES view. You can query it to see a listing of tables that you own:

```
SELECT table_name
FROM   tabs;
```

Note: For a complete listing of the columns in the USER_TABLES view, see "USER_TABLES" in the *Oracle® Database Reference 12c Release 1.*

You can also query the ALL_TABLES view to see a listing of all tables to which you have access.

Column Information

USER_TAB_COLUMNS: Provides detailed information about the columns in your tables

```
DESCRIBE user_tab_columns
```

```
Name                 Null     Type
-------------------- -------- --------------
TABLE_NAME           NOT NULL VARCHAR2(128)
COLUMN_NAME          NOT NULL VARCHAR2(128)
DATA_TYPE                     VARCHAR2(128)
DATA_TYPE_MOD                 VARCHAR2(3)
DATA_TYPE_OWNER               VARCHAR2(128)
DATA_LENGTH          NOT NULL NUMBER
DATA_PRECISION                NUMBER
DATA_SCALE                    NUMBER
NULLABLE                      VARCHAR2(1)
COLUMN_ID                     NUMBER
DEFAULT_LENGTH                NUMBER
```

. . .

ORACLE

You can query the USER_TAB_COLUMNS view to find detailed information about the columns in your tables. Although the USER_TABLES view provides information about your table names and storage, detailed column information is found in the USER_TAB_COLUMNS view.

This view contains information such as:

- Column names
- Column data types
- Length of data types
- Precision and scale for NUMBER columns
- Whether nulls are allowed (Is there a NOT NULL constraint on the column?)
- Default value

Note: For a complete listing and description of the columns in the USER_TAB_COLUMNS view, see "USER_TAB_COLUMNS" in the *Oracle® Database Reference 12c Release 1*.

Column Information

```
SELECT column_name, data_type, data_length,
       data_precision, data_scale, nullable
FROM   user_tab_columns
WHERE  table_name = 'EMPLOYEES';
```

	COLUMN_NAME	DATA_TYPE	DATA_LENGTH	DATA_PRECISION	DATA_SCALE	NULLABLE
1	EMPLOYEE_ID	NUMBER	22	6	0	N
2	FIRST_NAME	VARCHAR2	20	(null)	(null)	Y
3	LAST_NAME	VARCHAR2	25	(null)	(null)	N
4	EMAIL	VARCHAR2	25	(null)	(null)	N
5	PHONE_NUMBER	VARCHAR2	20	(null)	(null)	Y
6	HIRE_DATE	DATE	7	(null)	(null)	N
7	JOB_ID	VARCHAR2	10	(null)	(null)	N
8	SALARY	NUMBER	22	8	2	Y
9	COMMISSION_PCT	NUMBER	22	2	2	Y
10	MANAGER_ID	NUMBER	22	6	0	Y
11	DEPARTMENT_ID	NUMBER	22	4	0	Y

ORACLE

By querying the USER_TAB_COLUMNS table, you can find details about your columns such as the names, data types, data type lengths, null constraints, and default value for a column.

The example shown in the slide displays the columns, data types, data lengths, and null constraints for the EMPLOYEES table. Note that this information is similar to the output from the DESCRIBE command.

To view information about columns set as unused, you use the USER_UNUSED_COL_TABS dictionary view.

Note: Names of the objects in Data Dictionary are in uppercase.

Constraint Information

- USER_CONSTRAINTS describes the constraint definitions on your tables.
- USER_CONS_COLUMNS describes columns that are owned by you and that are specified in constraints.

USER_CONSTRAINTS

Name	Null	Type
OWNER		VARCHAR2(128)
CONSTRAINT_NAME		VARCHAR2(128)
CONSTRAINT_TYPE		VARCHAR2(1)
TABLE_NAME		VARCHAR2(128)
SEARCH_CONDITION		LONG
SEARCH_CONDITION_VC		VARCHAR2(4000)
R_OWNER		VARCHAR2(128)
R_CONSTRAINT_NAME		VARCHAR2(128)
DELETE_RULE		VARCHAR2(9)
STATUS		VARCHAR2(8)
DEFERRABLE		VARCHAR2(14)
DEFERRED		VARCHAR2(9)
VALIDATED		VARCHAR2(13)
GENERATED		VARCHAR2(14)
BAD		VARCHAR2(3)
RELY		VARCHAR2(4)
LAST_CHANGE		DATE
INDEX_OWNER		VARCHAR2(128)
INDEX_NAME		VARCHAR2(128)
INVALID		VARCHAR2(7)

Name	Null	Type
OWNER	NOT NULL	VARCHAR2(128)
CONSTRAINT_NAME	NOT NULL	VARCHAR2(128)
TABLE_NAME	NOT NULL	VARCHAR2(128)
COLUMN_NAME		VARCHAR2(4000)
POSITION		NUMBER

USER_CONS_COLUMNS

ORACLE

You can find out the names of your constraints, the type of constraint, the table name to which the constraint applies, the condition for check constraints, foreign key constraint information, deletion rule for foreign key constraints, the status, and many other types of information about your constraints.

Note: For a complete listing and description of the columns in the USER_CONSTRAINTS view, see "USER_CONSTRAINTS" in the *Oracle® Database Reference 12c Release 1.*

USER_CONSTRAINTS: Example

```
SELECT constraint_name, constraint_type,
       search_condition, r_constraint_name,
       delete_rule, status
FROM   user_constraints
WHERE  table_name = 'EMPLOYEES';
```

	CONSTRAINT_NAME	CONSTRAINT_TYPE	SEARCH_CONDITION	R_CONSTRAINT_NAME	DELETE_RULE	STATUS
1	EMP_LAST_NAME_NN	C	"LAST_NAME" IS NOT NULL	(null)	(null)	ENABLED
2	EMP_EMAIL_NN	C	"EMAIL" IS NOT NULL	(null)	(null)	ENABLED
3	EMP_HIRE_DATE_NN	C	"HIRE_DATE" IS NOT NULL	(null)	(null)	ENABLED
4	EMP_JOB_NN	C	"JOB_ID" IS NOT NULL	(null)	(null)	ENABLED
5	EMP_SALARY_MIN	C	salary > 0	(null)	(null)	ENABLED
6	EMP_EMAIL_UK	U	(null)	(null)	(null)	ENABLED
7	EMP_EMP_ID_PK	P	(null)	(null)	(null)	ENABLED
8	EMP_DEPT_FK	R	(null)	DEPT_ID_PK	NO ACTION	ENABLED
9	EMP_JOB_FK	R	(null)	JOB_ID_PK	NO ACTION	ENABLED
10	EMP_MANAGER_FK	R	(null)	EMP_EMP_ID_PK	NO ACTION	ENABLED

In the example shown in the slide, the USER_CONSTRAINTS view is queried to find the names, types, check conditions, name of the unique constraint that the foreign key references, deletion rule for a foreign key, and status for constraints on the EMPLOYEES table.

The CONSTRAINT_TYPE can be:

- C (check constraint on a table, or NOT NULL)
- P (primary key)
- U (unique key)
- R (referential integrity)
- V (with check option, on a view)
- O (with read-only, on a view)

The DELETE_RULE can be:

- CASCADE: If the parent record is deleted, the child records are also deleted.
- SET NULL: If the parent record is deleted, change the respective child records to null.
- NO ACTION: A parent record can be deleted only if no child records exist.

The STATUS can be:

- ENABLED: Constraint is active.
- DISABLED: Constraint is not active.

Querying USER_CONS_COLUMNS

```
SELECT constraint_name, column_name
FROM   user_cons_columns
WHERE  table_name = 'EMPLOYEES';
```

	CONSTRAINT_NAME	COLUMN_NAME
1	EMP_MANAGER_FK	MANAGER_ID
2	EMP_JOB_FK	JOB_ID
3	EMP_DEPT_FK	DEPARTMENT_ID
4	EMP_EMP_ID_PK	EMPLOYEE_ID
5	EMP_EMAIL_UK	EMAIL
6	EMP_SALARY_MIN	SALARY
7	EMP_JOB_NN	JOB_ID
8	EMP_HIRE_DATE_NN	HIRE_DATE
9	EMP_EMAIL_NN	EMAIL
10	EMP_LAST_NAME_NN	LAST_NAME

ORACLE

To find the names of the columns to which a constraint applies, query the USER_CONS_COLUMNS dictionary view. This view tells you the name of the owner of a constraint, the name of the constraint, the table that the constraint is on, the names of the columns with the constraint, and the original position of the column or attribute in the definition of the object.

Note: A constraint may apply to more than one column.

You can also write a join between USER_CONSTRAINTS and USER_CONS_COLUMNS to create customized output from both tables.

Lesson Agenda

- Introduction to data dictionary
- Querying the dictionary views for the following:
  - Table information
  - Column information
  - Constraint information
- Adding a comment to a table and querying the dictionary views for comment information

Adding Comments to a Table

- You can add comments to a table or column by using the COMMENT statement:

```
COMMENT ON TABLE employees
IS 'Employee Information';
```

```
COMMENT ON COLUMN employees.first_name
IS 'First name of the employee';
```

- Comments can be viewed through the data dictionary views:
  - ALL_COL_COMMENTS
  - USER_COL_COMMENTS
  - ALL_TAB_COMMENTS
  - USER_TAB_COMMENTS

ORACLE

You can add a comment of up to 4,000 bytes about a column, table, view, or snapshot by using the COMMENT statement. The comment is stored in the data dictionary and can be viewed in one of the following data dictionary views in the COMMENTS column:

- ALL_COL_COMMENTS
- USER_COL_COMMENTS
- ALL_TAB_COMMENTS
- USER_TAB_COMMENTS

Syntax

```
COMMENT ON {TABLE table | COLUMN table.column}
    IS 'text';
```

In the syntax:

- *table* Is the name of the table
- *column* Is the name of the column in a table
- *text* Is the text of the comment

You can drop a comment from the database by setting it to empty string (''):

```
COMMENT ON TABLE employees IS '';
```

Quiz

Oracle users should read and write row or schema information to the base tables contained in the SYS schema.

a. True
b. False

Answer: b

Quiz

Which of the following types of information do the dictionary views that are based on the dictionary tables contain?

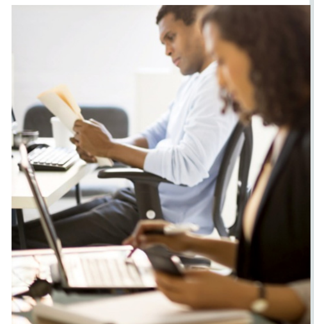
a. Definitions of all the schema objects in the database
b. Default values for the columns
c. Integrity constraint information
d. Privileges and roles that each user has been granted
e. All of the above

Answer: e

Summary

In this lesson, you should have learned how to find information about your objects through the following dictionary views:

- DICTIONARY
- USER_OBJECTS
- USER_TABLES
- USER_TAB_COLUMNS
- USER_CONSTRAINTS
- USER_CONS_COLUMNS

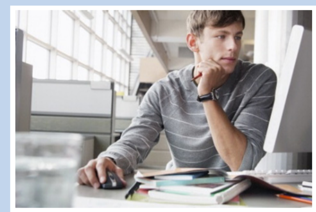


In this lesson, you learned about some of the dictionary views that are available to you. You can use these dictionary views to find information about your tables, constraints, views, sequences, and synonyms.

Practice 17: Overview

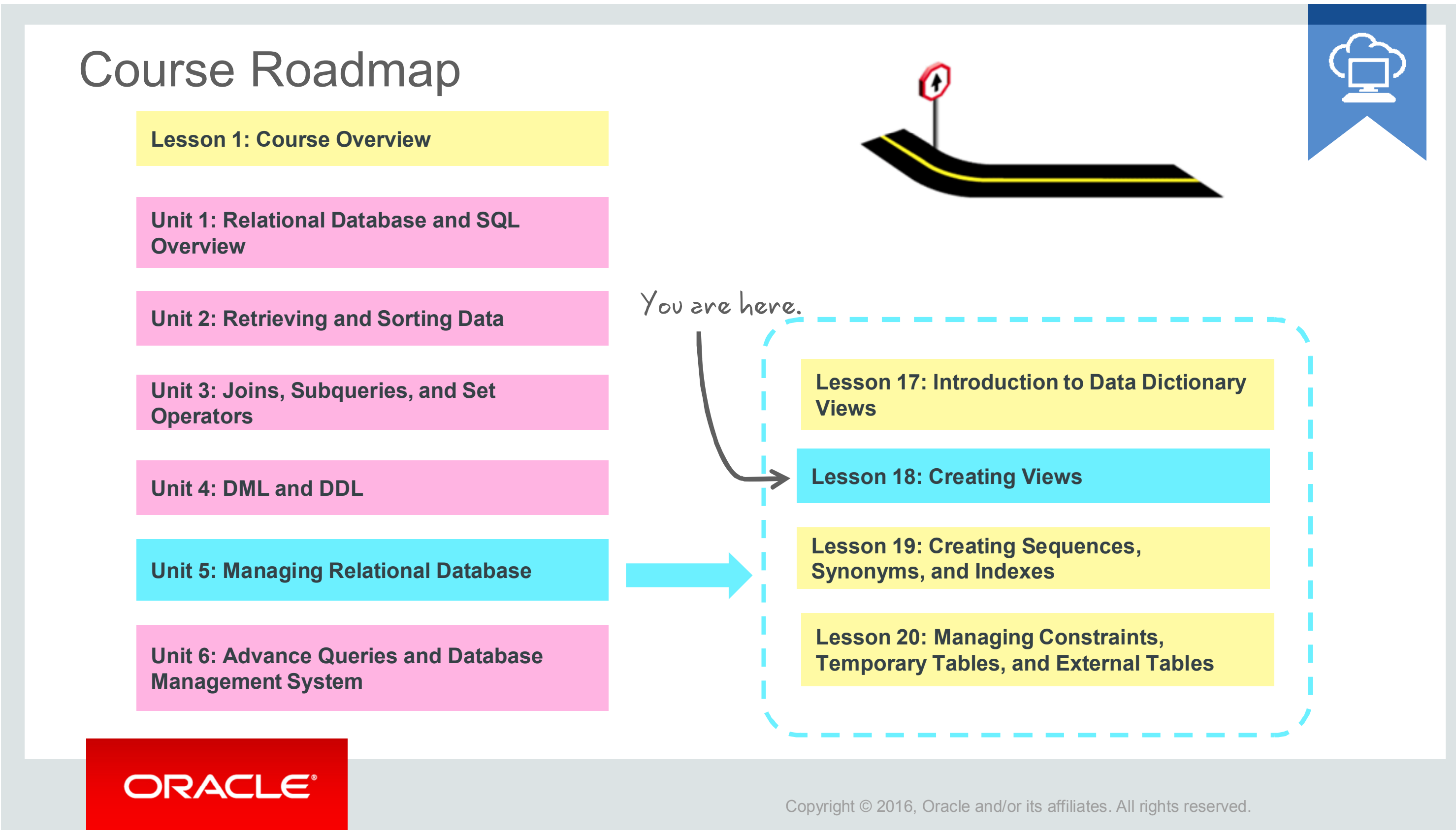
This practice covers the following topics:

- Querying the dictionary views for table and column information
- Querying the dictionary views for constraint information



In this practice, you query the dictionary views to find information about objects in your schema.

Lesson 18: Creating Views

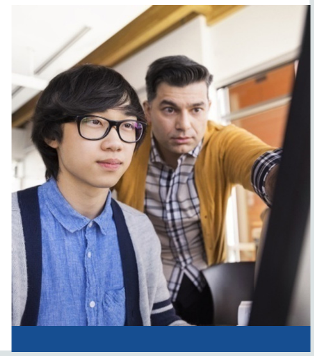


In Unit 5, you are introduced to views. You learn to:

- Query data dictionary views
- Create sequences, synonyms, and indexes
- Manage constraints and tables

Objectives

After completing this lesson, you should be able to:
- Create simple and complex views
- Retrieve data from views
- Query the data dictionary for view information



ORACLE

In this lesson, you are introduced to views, and you learn the basics of creating and using views.

Lesson Agenda

- Overview of views
- Creating, modifying, and retrieving data from a view
- Data Manipulation Language (DML) operations on a view
- Dropping a view

ORACLE

Database Objects

Object	Description
Table	Is the basic unit of storage; composed of rows
View	Logically represents subsets of data from one or more tables
Sequence	Generates numeric values
Index	Improves the performance of some queries
Synonym	Gives alternative names to objects

ORACLE

There are several other objects in a database in addition to tables.

With views, you can present and hide data from the tables.

Many applications require the use of unique numbers as primary key values. You can either build code into the application to handle this requirement or use a sequence to generate unique numbers.

If you want to improve the performance of data retrieval queries, you should consider creating an index. You can also use indexes to enforce uniqueness on a column or a collection of columns.

You can provide alternative names for objects by using synonyms.

Views

A view:

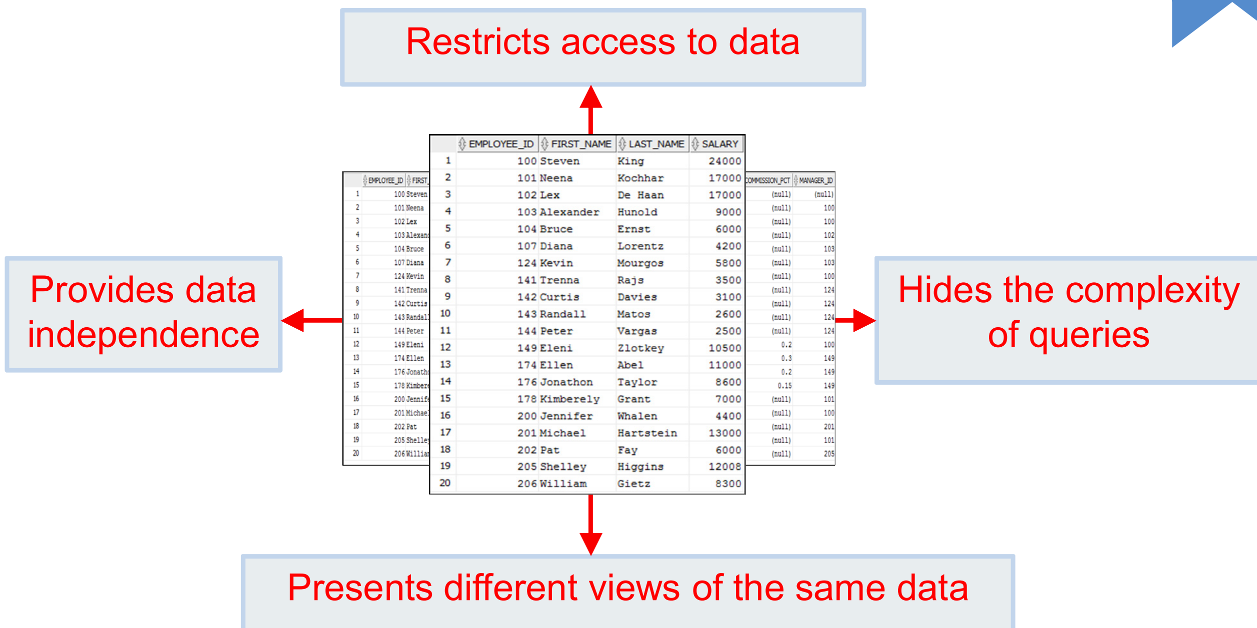
- Is a schema object
- Presents logical subsets of data
- Is a stored `SELECT` statement based on a table or another view
- Contains no data of its own

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	SALARY
1	100	Steven	King	24000
2	101	Neena	Kochhar	17000
3	102	Lex	De Haan	17000
4	103	Alexander	Hunold	9000
5	104	Bruce	Ernst	6000
6	107	Diana	Lorentz	4200
7	124	Kevin	Mourgos	5800
8	141	Trenna	Rajs	3500
9	142	Curtis	Davies	3100
10	143	Randall	Matos	2600
11	144	Peter	Vargas	2500
12	149	Eleni	Zlotkey	10500
13	174	Ellen	Abel	11000
14	176	Jonathon	Taylor	8600
15	178	Kimberely	Grant	7000
16	200	Jennifer	Whalen	4400
17	201	Michael	Hartstein	13000
18	202	Pat	Fay	6000
19	205	Shelley	Higgins	12008
20	206	William	Gietz	8300

ORACLE

You can present logical subsets or combinations of data by creating views of tables. A view is a schema object, a stored `SELECT` statement based on a table or another view. A view contains no data of its own, but is like a window through which data from tables can be viewed or changed. The tables on which a view is based are called *base tables*. The view is stored as a `SELECT` statement in the data dictionary.

Advantages of Views



- Views restrict access to data because they display selected columns from the table.
- Views can be used to make simple queries to retrieve the results of complicated queries. For example, views can be used to query information from multiple tables without the user knowing how to write a join statement.
- Views provide data independence for ad hoc users and application programs. One view can be used to retrieve data from several tables.
- Views provide groups of users access to data according to their particular criteria.

For more information, see the "`CREATE VIEW`" section in *Oracle Database SQL Language Reference* for Oracle Database 12*c*.

Simple Views and Complex Views

Feature	Simple Views	Complex Views
Number of tables	One	One or more
Contain functions	No	Yes
Contain groups of data	No	Yes
DML operations through a view	Yes	Not always

ORACLE

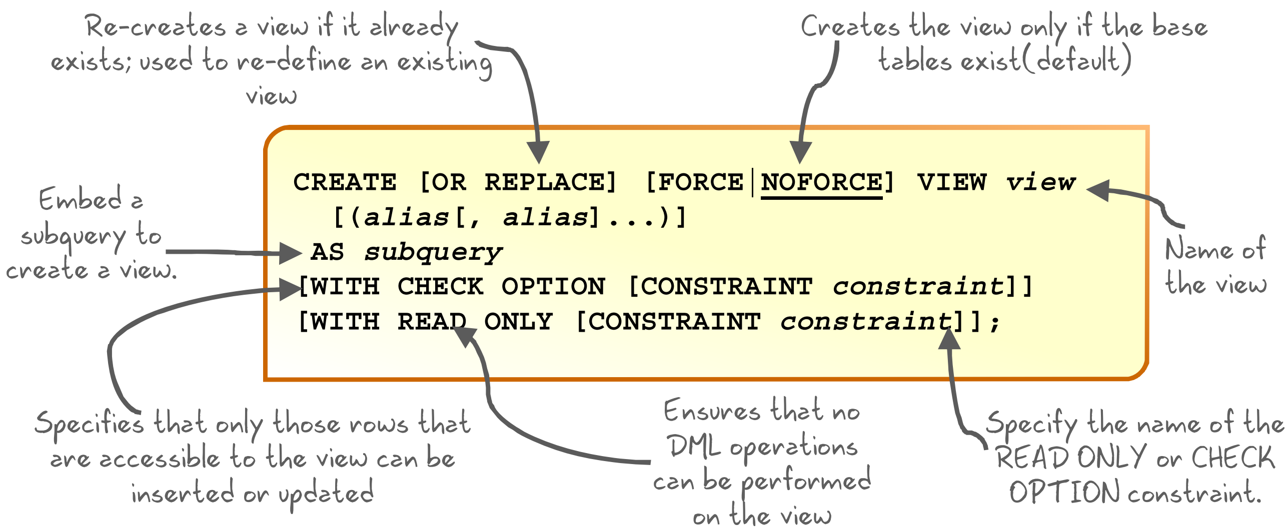
There are two classifications for views: simple and complex. The basic difference is related to the DML (`INSERT`, `UPDATE`, and `DELETE`) operations.

- A simple view is one that:
  - Derives data from only one table
  - Contains no functions or groups of data
  - Usually performs DML operations through the view
- A complex view is one that:
  - Derives data from many tables
  - Contains functions or groups of data
  - Does not always allow DML operations through the view

Lesson Agenda

- Overview of views
- Creating, modifying, and retrieving data from a view
- Data Manipulation Language (DML) operations on a view
- Dropping a view

Creating a View



You can create a view by embedding a subquery in the `CREATE VIEW` statement.

In the syntax:

`OR REPLACE`	Re-creates the view if it already exists. You can use this clause to change the definition of an existing view without dropping, re-creating, and regranting the object privileges previously granted on it.
`FORCE`	Creates the view whether or not the base tables exist
`NOFORCE`	Creates the view only if the base tables exist (This is the default.)
`view`	Is the name of the view
`alias`	Specifies names for the expressions selected by the view's query (The number of aliases must match the number of expressions selected by the view.)
`subquery`	Is a complete `SELECT` statement (You can use aliases for the columns in the `SELECT` list.)
`WITH CHECK OPTION`	Specifies that only those rows that are accessible to the view can be inserted or updated
`Constraint`	Is the name assigned to the `CHECK OPTION` or `READ ONLY` constraint. If you do not specify this, the system automatically assigns the constraint a name of the form `SYS_Cn`, where *n* is an integer that makes the constraint name unique within the database.
`WITH READ ONLY`	Ensures that no DML operations can be performed on this view

Note: In SQL Developer, click the Run Script icon or press F5 to run the data definition language (DDL) statements. The feedback messages will be shown on the Script Output tabbed page.

Creating a View

- Create the `EMPVU80` view, which contains details of the employees in department 80:

```
CREATE VIEW       empvu80
 AS SELECT  employee_id, last_name, salary
    FROM    employees
    WHERE   department_id = 80;
```

```
view EMPVU80 created.
```

- Describe the structure of the view:

```
DESCRIBE empvu80;
```

```
Name        Null     Type
----------- -------- ------------
EMPLOYEE_ID NOT NULL NUMBER(6)
LAST_NAME   NOT NULL VARCHAR2(25)
SALARY               NUMBER(8,2)
```

ORACLE

The example in the slide creates a view that contains the employee number, last name, and salary for each employee in department 80.

You can display the structure of the view by using the `DESCRIBE` command.

Guidelines

- The subquery that defines a view can contain complex `SELECT` syntax, including joins, groups, and subqueries.
- If you do not specify a constraint name for the view that is created with the `WITH CHECK OPTION`, the system assigns a default name in the `SYS_Cn` format.
- You can use the `OR REPLACE` option to change the definition of the view without dropping and re-creating it, or regranting the object privileges previously granted on it.

Creating a View

- Create a view by using column aliases in the subquery:

```
CREATE VIEW salvu50
 AS SELECT  employee_id ID_NUMBER, last_name NAME,
            salary*12 ANN_SALARY
    FROM    employees
    WHERE   department_id = 50;
```

```
view SALVU50 created.
```

- Select the columns from this view by using the given alias names.

ORACLE

You can control the column names by including column aliases in the subquery.

The example in the slide creates a view containing the employee number (EMPLOYEE_ID) with the alias ID_NUMBER, name (LAST_NAME) with the alias NAME, and annual salary (SALARY) with the alias ANN_SALARY for every employee in department 50.

Alternatively, you can use an alias after the CREATE statement and before the SELECT subquery. The number of aliases listed must match the number of expressions selected in the subquery.

```
CREATE OR REPLACE VIEW salvu50 (ID_NUMBER, NAME, ANN_SALARY)
  AS SELECT  employee_id, last_name, salary*12
     FROM    employees
     WHERE   department_id = 50;
```

Retrieving Data from a View

```
SELECT * FROM salvu50;
```

	ID_NUMBER	NAME	ANN_SALARY
1	124	Mourgos	69600
2	141	Rajs	42000
3	142	Davies	37200
4	143	Matos	31200
5	144	Vargas	30000

```
SELECT name, ann_salary FROM salvu50;
```

	NAME	ANN_SALARY
1	Mourgos	69600
2	Rajs	42000
3	Davies	37200
4	Matos	31200
5	Vargas	30000

ORACLE

You can retrieve data from a view as you would from any table. You can display either the contents of the entire view or just specific rows and columns. You select the columns from the view by using the given alias names.

Modifying a View

- Modify the `EMPVU80` view by using a `CREATE OR REPLACE VIEW` clause.
- Add an alias for each column name in the same order as the columns in the subquery.

```
CREATE OR REPLACE VIEW empvu80
  (id_number, name, sal, department_id)
AS SELECT  employee_id, first_name || ' '
           || last_name, salary, department_id
   FROM    employees
   WHERE   department_id = 80;
view EMPVU80 created.
```

	ID_NUMBER	NAME	SAL	DEPARTMENT_ID
1	149	Eleni Zlotkey	10500	80
2	174	Ellen Abel	11000	80
3	176	Jonathon Taylor	8600	80

Note that the employee's first_name and last_name columns are concatenated as the name column in the view.

ORACLE

With the `REPLACE` option, a view can be created even if one exists with this name already, thus replacing the old version of the view for its owner. This means that the view can be altered without dropping, re-creating, and regranting object privileges.

Note: When assigning column aliases in the `CREATE OR REPLACE VIEW` clause, remember that the aliases are listed in the same order as the columns in the subquery.

Creating a Complex View

Create a complex view that contains group functions to display values from two tables:

```
CREATE OR REPLACE VIEW dept_sum_vu
  (name, minsal, maxsal, avgsal)
AS SELECT   d.department_name, MIN(e.salary),
            MAX(e.salary),AVG(e.salary)
   FROM     employees e JOIN departments d
   USING    (department_id)
   GROUP BY d.department_name;
```

```
view DEPT_SUM_VU created.
```

Structure of the view

```
Name   Null     Type
------ -------- ------------
NAME   NOT NULL VARCHAR2(30)
MINSAL          NUMBER
MAXSAL          NUMBER
AVGSAL          NUMBER
```

Data from the view

	NAME	MINSAL	MAXSAL	AVGSAL
1	Administration	4400	4400	4400
2	Accounting	8300	12008	10154
3	IT	4200	9000	6400
4	Executive	17000	24000	19333.33333333333333333333333333333333
5	Shipping	2500	5800	3500
6	Sales	8600	11000	10033.33333333333333333333333333333333
7	Marketing	6000	13000	9500

ORACLE®

The example in the slide creates a complex view of department names, minimum salaries, maximum salaries, and the average salaries by department. Note that alternative names have been specified for the view. This is a requirement if any column of the view is derived from a function or an expression.

You can view the structure of the view by using the `DESCRIBE` command. Display the contents of the view by issuing a `SELECT` statement.

```
SELECT  * FROM   dept_sum_vu;
```

View Information

1

```
DESCRIBE user_views
```

```
Name                           Null     Type
------------------------------ -------- ------------
VIEW_NAME                      NOT NULL VARCHAR2(30)
TEXT_LENGTH                             NUMBER
TEXT                                    LONG()
...
```

2

```
SELECT view_name FROM user_views;
```

	VIEW_NAME
1	EMP_DETAILS_VIEW
2	SALVU50
3	EMPVU80
4	DEPT_SUM_VU

3

```
SELECT text FROM user_views
WHERE view_name = 'EMP_DETAILS_VIEW';
```

```
  TEXT
1 SELECT  e.employee_id,   e.job_id,   e.manager_id,   e.department_id,  d.location_id,  l.country_id,  e.first_name,  e.last_name,  e.salary,  e.commission_pct,  d.department_name,  j.job_title,  l.city,
l.state_province,  c.country_name,  r.region_nameFROM  employees e,  departments d,  jobs j,  locations l,  countries c,  regions rWHERE e.department_id = d.department_id  AND d.location_id = l.location_id
AND l.country_id = c.country_id  AND c.region_id = r.region_id  AND j.job_id = e.job_id WITH READ ONLY
```

ORACLE®

After your view is created, you can query the data dictionary view called USER_VIEWS to see the name of the view and the view definition. The text of the SELECT statement that constitutes your view is stored in a LONG column. The TEXT_LENGTH column is the number of characters in the SELECT statement. By default, when you select from a LONG column, only the first 80 characters of the column's value are displayed. To see more than 80 characters in SQL*Plus, use the SET LONG command:

```
SET LONG 1000
```

In the examples in the slide:

1. The USER_VIEWS columns are displayed. Note that this is a partial listing.
2. The names of your views are retrieved
3. The SELECT statement for the EMP_DETAILS_VIEW is displayed from the dictionary

Data Access Using Views

When you access data by using a view, the Oracle server performs the following operations:

- It retrieves the view definition from the data dictionary table USER_VIEWS.
- It checks access privileges for the view base table.
- It converts the view query into an equivalent operation on the underlying base table or tables. That is, data is retrieved from, or an update is made to, the base tables.

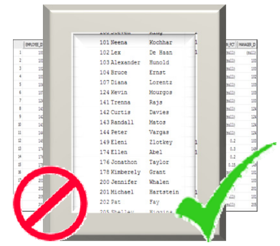
Lesson Agenda

- Overview of views
- Creating, modifying, and retrieving data from a view
- Data Manipulation Language (DML) operations on a view
- Dropping a view

ORACLE

Rules for Performing DML Operations on a View

- You can usually perform DML operations on simple views.
- You cannot remove a row if the view contains the following:
  - Group functions
  - A `GROUP BY` clause
  - The `DISTINCT` keyword
  - The pseudocolumn `ROWNUM` keyword



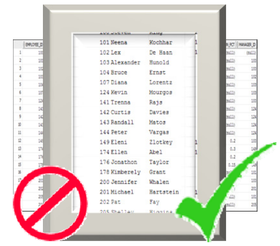
ORACLE

- You can perform DML operations on data through a view if those operations follow certain rules.
- You can remove a row from a view unless it contains any of the following:
  - Group functions
  - A `GROUP BY` clause
  - The `DISTINCT` keyword
  - The pseudocolumn `ROWNUM` keyword

Rules for Performing DML Operations on a View

You cannot modify data in a view if it contains:

- Group functions
- A GROUP BY clause
- The DISTINCT keyword
- The pseudocolumn ROWNUM keyword
- Expressions

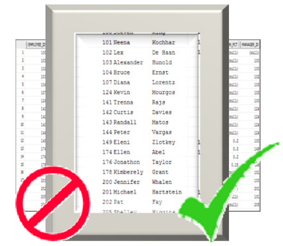


You can modify data through a view unless it contains any of the conditions mentioned in the slide.

Rules for Performing DML Operations on a View

You cannot add data through a view if the view includes:

- Group functions
- A `GROUP BY` clause
- The `DISTINCT` keyword
- The pseudocolumn `ROWNUM` keyword
- Columns defined by expressions
- `NOT NULL` columns without a default value in the base tables that are not selected by the view



ORACLE

You can add data through a view unless it contains any of the items listed in the slide. You cannot add data to a view if the view contains `NOT NULL` columns without default values in the base table. All the required values must be present in the view. Remember that you are adding values directly to the underlying table *through* the view.

For more information, see the "`CREATE VIEW`" section in *Oracle Database SQL Language Reference* for Oracle Database 12*c*.

Using the WITH CHECK OPTION Clause

- You can ensure that DML operations performed on the view stay in the domain of the view by using the WITH CHECK OPTION clause:

```
CREATE OR REPLACE VIEW empvu20
AS SELECT   *
   FROM     employees
   WHERE    department_id = 20
   WITH CHECK OPTION CONSTRAINT empvu20_ck ;
```

```
view EMPVU20 created.
```

- Any attempt to INSERT a row with a department_id other than 20 or to UPDATE the department number for any row in the view fails because it violates the WITH CHECK OPTION.

ORACLE

It is possible to perform referential integrity checks through views. You can also enforce constraints at the database level. The view can be used to protect data integrity, but the use is very limited.

The WITH CHECK OPTION clause specifies that INSERTs and UPDATEs performed through the view cannot create rows that the view cannot select. Therefore, it enables integrity constraints and data validation checks to be enforced on data being inserted or updated. If there is an attempt to perform DML operations on rows that the view has not selected, an error is displayed, along with the constraint name if it has been specified.

Using the WITH CHECK OPTION Clause

- Attempt to update department_id to 10 for employee_id 201 returns an error:

```
UPDATE empvu20
SET    department_id = 10
WHERE  employee_id = 201;
```

```
Error starting at line : 6 in command -
UPDATE empvu20
SET    department_id = 10
WHERE  employee_id = 201
Error report -
SQL Error: ORA-01402: view WITH CHECK OPTION where-clause violation
01402. 00000 -  "view WITH CHECK OPTION where-clause violation"
*Cause:
*Action:
```

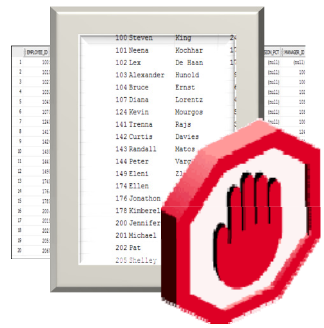
- Error is returned because if the department number were to change to 10, the view would no longer be able to see that employee.

ORACLE

With the WITH CHECK OPTION clause, the view can see only the employees in department 20 and does not allow the department number for those employees to be changed through the view.

Denying DML Operations

- You can ensure that no DML operations occur by adding the `WITH READ ONLY` option to your view definition.
- Any attempt to perform a DML operation on any row in the view results in an Oracle server error.



ORACLE

You can ensure that no DML operations occur on your view by creating it with the `WITH READ ONLY` option. The example in the next slide modifies the `EMPVU10` view to prevent any DML operations on the view.

Denying DML Operations

```
CREATE OR REPLACE VIEW empvu10
    (employee_number, employee_name, job_title)
AS SELECT   employee_id, last_name, job_id
   FROM     employees
   WHERE    department_id = 10
   WITH READ ONLY ;
```

```
view EMPVU10 created.
```

```
    DELETE FROM empvu10
    WHERE  employee_number = 200;
```

```
Error starting at line : 8 in command -
DELETE FROM empvu10
WHERE  employee_number = 200
Error at Command Line : 8 Column : 13
Error report -
SQL Error: ORA-42399: cannot perform a DML operation on a read-only view
42399.0000 - "cannot perform a DML operation on a read-only view"
```

ORACLE

Any attempt to remove a row from a view with a read-only constraint results in an error.

Similarly, any attempt to insert a row or modify a row by using a view with a read-only constraint results in the same error.

Lesson Agenda

- Overview of views
- Creating, modifying, and retrieving data from a view
- Data Manipulation Language (DML) operations on a view
- Dropping a view

Removing a View

- You can remove a view without losing the underlying base tables.
- The Drop View statement removes only the view definition from the database.

```
DROP VIEW view;
```

```
DROP VIEW empvu80;
```

```
view EMPVU80 dropped.
```

- Views or other applications based on the deleted view become invalid.

You use the `DROP VIEW` statement to remove a view. The statement removes the view definition from the database. However, dropping views has no effect on the tables on which the view was based. Alternatively, views or other applications based on the deleted views become invalid. Only the creator or a user with the `DROP ANY VIEW` privilege can remove a view.

In the syntax, *`view`* is the name of the view.

Quiz

Views store the selected data rows from the underlying base tables.

a. True

b. False

Answer: b

Quiz

You cannot add data through a view if the view includes a `GROUP BY` clause.

a. True

b. False

Answer: a

Quiz

In the `CREATE VIEW` statement syntax, which one of the following options enables you to change the definition of an existing view without dropping or re-creating it.

a. `FORCE/NO FORCE`

b. `WITH CHECK OPTION`

c. `CREATE OR REPLACE`

ORACLE

Answer: c

Quiz

You can query the data dictionary view called `USER_VIEWS` to see the names of the views and view definitions.

a. True

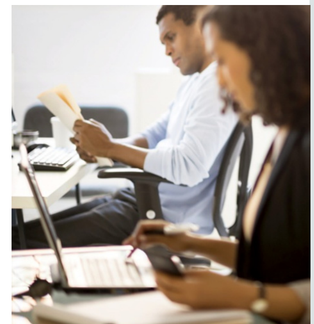
b. False

Answer: a

Summary

In this lesson, you should have learned how to:

- Create, use, and remove views
- Query the data dictionary for view information



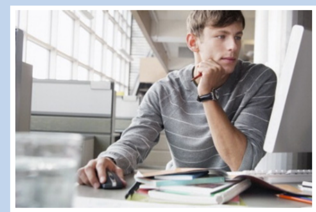
ORACLE

In this lesson, you should have learned about views.

Practice 18: Overview

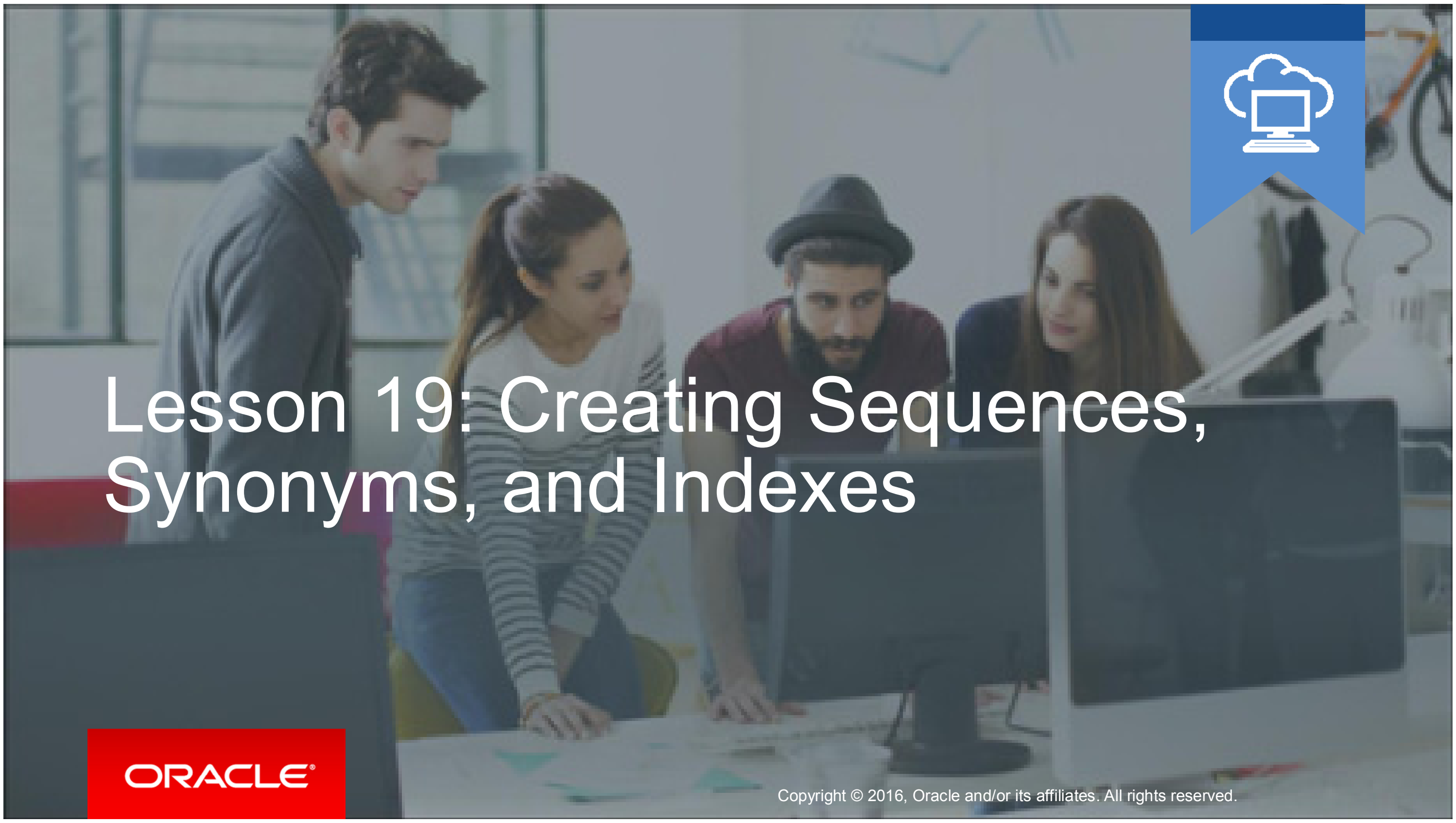
This practice covers the following topics:

- Creating a simple view
- Creating a complex view
- Creating a view with a check constraint
- Attempting to modify data in the view
- Querying the dictionary views for view information
- Removing views



ORACLE

The practice provides you with a variety of exercises in creating, using, querying data dictionary views for view information, and removing views.


Lesson 19: Creating Sequences, Synonyms, and Indexes
ORACLE
Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Course Roadmap

- Lesson 1: Course Overview
- Unit 1: Relational Database and SQL Overview
- Unit 2: Retrieving and Sorting Data
- Unit 3: Joins, Subqueries, and Set Operators
- Unit 4: DML and DDL
- Unit 5: Managing Relational Database
- Unit 6: Advance Queries and Database Management System

You are here

- Lesson 17: Introduction to Data Dictionary Views
- Lesson 18: Creating Views
- Lesson 19: Creating Sequences, Synonyms, and Indexes
- Lesson 20: Managing Constraints, Temporary Tables, and External Tables

In Unit 5, you are introduced to views. You learn to:

- Query data dictionary views
- Create sequences, synonyms, and indexes
- Manage constraints and tables

Objectives

After completing this lesson, you should be able to:

- Create, maintain, and use sequences
- Create private and public synonyms
- Create and maintain indexes
- Query various data dictionary views to find information for sequences, synonyms, and indexes

ORACLE

In this lesson, you are introduced to the sequence, synonyms, and index objects. You learn the basics of creating and using sequences, synonyms, and indexes.

Lesson Agenda

- Overview of sequences:
  - Creating, using, and modifying a sequence
  - Caching sequence values
  - `NEXTVAL` and `CURRVAL` pseudocolumns
  - SQL column defaulting using a sequence
- Overview of synonyms
  - Creating and dropping synonyms
- Overview of indexes
  - Creating indexes
  - Using the `CREATE TABLE` statement
  - Creating function-based indexes
  - Creating multiple indexes on the same set of columns
  - Removing indexes

Database Objects

Object	Description
Table	Is the basic unit of storage; composed of rows
View	Logically represents subsets of data from one or more tables
Sequence	Generates numeric values
Index	Improves the performance of some queries
Synonym	Gives alternative names to objects

ORACLE

There are several other objects in a database in addition to tables.

With views, you can present and hide data from the tables.

Many applications require the use of unique numbers as primary key values. You can either build code into the application to handle this requirement or use a sequence to generate unique numbers.

If you want to improve the performance of data retrieval queries, you should consider creating an index. You can also use indexes to enforce uniqueness on a column or a collection of columns.

You can provide alternative names for objects by using synonyms.

Referencing Another User's Tables

- Tables belonging to other users are not in the user's schema.
- You should use the owner's name as a prefix to those tables.

USERA

```
SELECT *
FROM userB.employees;
```

USERB

```
SELECT *
FROM userA.employees;
```

ORACLE

A schema is a collection of logical structures of data or *schema objects*. A schema is owned by a database user and has the same name as that user. Each user owns a single schema.

Schema objects can be created and manipulated with SQL and include tables, views, synonyms, sequences, stored procedures, indexes, clusters, and database links.

If a table does not belong to the user, the owner's name must be prefixed to the table. For example, if there are schemas named `USERA` and `USERB`, and both have an `EMPLOYEES` table, if `USERA` wants to access the `EMPLOYEES` table that belongs to `USERB`, `USERA` must prefix the table name with the schema name:

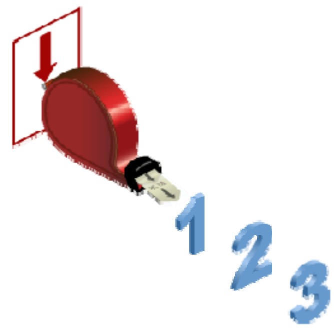
```
SELECT *
FROM   userb.employees;
```

If `USERB` wants to access the `EMPLOYEES` table that is owned by `USERA`, `USERB` must prefix the table name with the schema name:

```
SELECT *
FROM   usera.employees;
```

Sequence

A sequence:
- Can automatically generate unique numbers
- Is a shareable object
- Can be used to create a primary key value
- Replaces application code
- Speeds up the efficiency of accessing sequence values when cached in memory



A sequence is a user-created database object that can be shared by multiple users to generate integers. You can define a sequence to generate unique values or to recycle and use the same numbers again.

A typical usage for sequences is to create a primary key value, which must be unique for each row. A sequence is generated and incremented (or decremented) by an internal Oracle routine. This can be a time-saving object, because it can reduce the amount of application code needed to write a sequence-generating routine.

Sequence numbers are stored and generated independent of tables. Therefore, the same sequence can be used for multiple tables.

CREATE SEQUENCE Statement: Syntax

Define a sequence to generate sequential numbers automatically:

```
CREATE SEQUENCE [ schema. ] sequence
   [ { START WITH|INCREMENT BY } integer
   | { MAXVALUE integer | NOMAXVALUE }
   | { MINVALUE integer | NOMINVALUE }
   | { CYCLE | NOCYCLE }
   | { CACHE integer | NOCACHE }
   | { ORDER | NOORDER }
];
```

First sequence number to be generated

Specifies the interval between sequence numbers

Specifies whether the sequence continues to generate values after reaching its maximum or minimum value

Specifies how many values the Oracle Server pre-allocates and keeps in memory

Specify ORDER to guarantee that sequence numbers are generated in order of request.

ORACLE

Automatically generate sequential numbers by using the `CREATE SEQUENCE` statement.

In the syntax:

`Sequence`	Is the name of the sequence generator
`START WITH` *n*	Specifies the first sequence number to be generated (If this clause is omitted, the sequence starts with 1.)
`INCREMENT BY` *n*	Specifies the interval between sequence numbers, where *n* is an integer (If this clause is omitted, the sequence increments by 1.)
`MAXVALUE` *n*	Specifies the maximum value the sequence can generate
`NOMAXVALUE`	Specifies a maximum value of 10^27 for an ascending sequence and –1 for a descending sequence (This is the default option.)
`MINVALUE` *n*	Specifies the minimum sequence value
`NOMINVALUE`	Specifies a minimum value of 1 for an ascending sequence and –(10^26) for a descending sequence (This is the default option.)

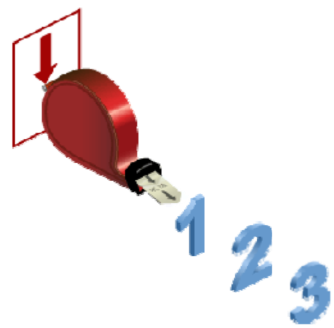
`ORDER`	Specify `ORDER` to guarantee that sequence numbers are generated in order of request. This clause is useful if you are using the sequence numbers as time stamps.
`NOORDER`	Specify `NOORDER` if you do not want to guarantee that sequence numbers are generated in order of request. This is the default.
`CYCLE | NOCYCLE`	Specifies whether the sequence continues to generate values after reaching its maximum or minimum value (`NOCYCLE` is the default option.)
`CACHE` *n* `| NOCACHE`	Specifies how many values the Oracle Server pre-allocates and keeps in memory (By default, the Oracle server caches 20 values.)

Creating a Sequence: Example

- Create a sequence named `DEPT_DEPTID_SEQ` to be used for the primary key of the `DEPARTMENTS` table.
- Do not use the `CYCLE` option.

```
CREATE SEQUENCE dept_deptid_seq
START WITH 280
INCREMENT BY 10
MAXVALUE 9999
NOCACHE;
```

```
Sequence DEPT_DEPTID_SEQ created.
```



ORACLE

The example in the slide creates a sequence named `DEPT_DEPTID_SEQ` to be used for the `DEPARTMENT_ID` column of the `DEPARTMENTS` table. The sequence starts at 280 and increments by 10, its maximum value is 9999, and it does not allow caching.

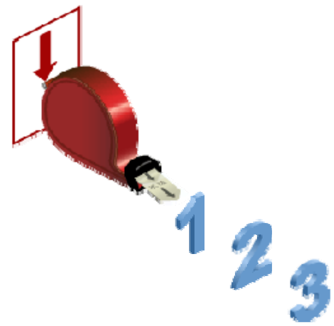
Do not use the `CYCLE` option if the sequence is used to generate primary key values, unless you have a reliable mechanism that purges old rows faster than the sequence cycles.

For more information, see the "`CREATE SEQUENCE`" section in the *Oracle Database SQL Language Reference* for Oracle Database 12*c*.

Note: The sequence is not tied to a table. Generally, you should name the sequence after its intended use. However, the sequence can be used anywhere, regardless of its name.

NEXTVAL and CURRVAL Pseudocolumns

- NEXTVAL returns the next available sequence value. It returns a unique value every time it is referenced, even for different users.
- CURRVAL obtains the current sequence value.
- NEXTVAL must be issued for that sequence before CURRVAL contains a value.



After you create your sequence, sequential numbers are generated for use in your tables. Reference the sequence values by using the NEXTVAL and CURRVAL pseudocolumns.

The NEXTVAL pseudocolumn is used to extract successive sequence numbers from a specified sequence. You must qualify NEXTVAL with the sequence name. When you reference *sequence*.NEXTVAL, a new sequence number is generated and the current sequence number is placed in CURRVAL.

The CURRVAL pseudocolumn is used to refer to a sequence number that the current user has just generated. However, NEXTVAL must be used to generate a sequence number in the current user's session before CURRVAL can be referenced. You must qualify CURRVAL with the sequence name. When you reference *sequence*.CURRVAL, the last value returned to that user's process is displayed.

Rules for Using `NEXTVAL` and `CURRVAL`

You can use `NEXTVAL` and `CURRVAL` in the following contexts:

- The `SELECT` list of a `SELECT` statement that is not part of a subquery
- The `SELECT` list of a subquery in an `INSERT` statement
- The `VALUES` clause of an `INSERT` statement
- The `SET` clause of an `UPDATE` statement

You cannot use `NEXTVAL` and `CURRVAL` in the following contexts:

- The `SELECT` list of a view
- A `SELECT` statement with the `DISTINCT` keyword
- A `SELECT` statement with the `GROUP BY`, `HAVING`, or `ORDER BY` clauses
- A subquery in a `SELECT`, `DELETE`, or `UPDATE` statement

For more information, see the "Pseudocolumns" and "`CREATE SEQUENCE`" sections in *Oracle Database SQL Language Reference* for Oracle Database 12*c.*

Using a Sequence

- Insert a new department named “Support” in location ID 2500:

```
INSERT INTO departments(department_id,
            department_name, location_id)
VALUES      (dept_deptid_seq.NEXTVAL,
            'Support', 2500);
```

```
1 rows inserted
```

- View the current value for the `DEPT_DEPTID_SEQ` sequence:

```
SELECT      dept_deptid_seq.CURRVAL
FROM        dual;
```

	CURRVAL
1	280

ORACLE

The example in the slide inserts a new department in the `DEPARTMENTS` table. It uses the `DEPT_DEPTID_SEQ` sequence to generate a new department number as follows.

You can view the current value of the sequence by using *sequence_name*.`CURRVAL`, as shown in the second example in the slide.

Suppose that you now want to hire employees to staff the new department. The `INSERT` statement to be executed for all new employees can include the following code:

```
INSERT INTO employees (employee_id, department_id, ...)
VALUES (employees_seq.NEXTVAL, dept_deptid_seq .CURRVAL, ...);
```

Note: The preceding example assumes that a sequence called `EMPLOYEE_SEQ` has already been created to generate new employee numbers.

SQL Column Defaulting Using a Sequence

- The SQL syntax for column defaults allows <sequence>.nextval and <sequence>.currval as a SQL column defaulting expression for numeric columns, where <sequence> is an Oracle database sequence.
- The DEFAULT expression can include the sequence pseudocolumns CURRVAL and NEXTVAL, as long as the sequence exists and you have the privileges necessary to access it.

```
CREATE SEQUENCE s1 START WITH 1;
CREATE TABLE emp (a1 NUMBER DEFAULT s1.NEXTVAL NOT
NULL, a2 VARCHAR2(10));
INSERT INTO emp (a2) VALUES ('john');
INSERT INTO emp (a2) VALUES ('mark');
SELECT * FROM emp;
```

```
Sequence S1 created.

Table EMP created.

1 row inserted.

1 row inserted.
```

	A1	A2
1	1	john
2	2	mark

ORACLE

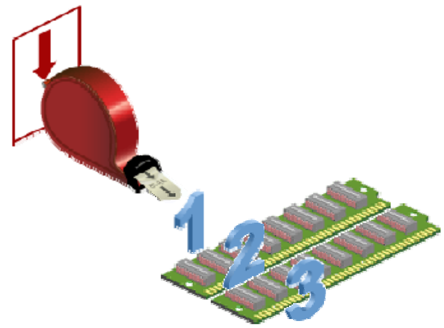
The SQL syntax for column defaults has been enhanced so that it allows <sequence>.nextval and <sequence>.currval as a SQL column defaulting expression for numeric columns, where <sequence> is an Oracle database sequence.

The DEFAULT expression can include the sequence pseudocolumns CURRVAL and NEXTVAL, as long as the sequence exists and you have the privileges necessary to access it. The user that is inserting into a table must have access privileges to the sequence. If the sequence is dropped, subsequent insert DMLs where *expr* is used for defaulting will result in a compilation error.

In the slide example, sequence s1 is created, which starts from 1.

Caching Sequence Values

- Caching sequence values in memory gives faster access to those values.
- Gaps in sequence values can occur when:
  - A rollback occurs
  - The system crashes
  - A sequence is used in another table



ORACLE

You can cache sequences in memory to provide faster access to those sequence values. The cache is populated the first time you refer to the sequence. Each request for the next sequence value is retrieved from the cached sequence. After the last sequence value is used, the next request for the sequence pulls another cache of sequences into memory.

Gaps in the Sequence

Although sequence generators issue sequential numbers without gaps, this action occurs independently of a commit or rollback. Therefore, if you roll back a statement containing a sequence, the number is lost.

Another event that can cause gaps in the sequence is a system crash. If the sequence caches values in memory, those values are lost if the system crashes.

Because sequences are not tied directly to tables, the same sequence can be used for multiple tables. However, if you do so, each table can contain gaps in the sequential numbers.

Modifying a Sequence

You can change the increment value, maximum value, minimum value, cycle option, or cache option:

```
ALTER SEQUENCE dept_deptid_seq
               INCREMENT BY 20
               MAXVALUE 999999
               NOCACHE
               NOCYCLE;
```

```
sequence DEPT_DEPTID_SEQ altered.
```


If you reach the MAXVALUE limit for your sequence, no additional values from the sequence are allocated and you receive an error indicating that the sequence exceeds MAXVALUE. To continue to use the sequence, you can modify it by using the ALTER SEQUENCE statement.

Syntax

```
ALTER  SEQUENCE   sequence
      [INCREMENT BY n]
      [{MAXVALUE n | NOMAXVALUE}]
      [{MINVALUE n | NOMINVALUE}]
      [{CYCLE | NOCYCLE}]
      [{CACHE n | NOCACHE}];
```

In the syntax, *sequence* is the name of the sequence generator.

For more information, see the section on "ALTER SEQUENCE" in *Oracle Database SQL Language Reference* for Oracle Database 12*c*.

Guidelines for Modifying a Sequence

- You must be the owner or have the ALTER privilege for the sequence.
- Only future sequence numbers are affected.
- The sequence must be dropped and re-created to restart the sequence at a different number.
- Some validation is performed.
- To remove a sequence, use the DROP statement:

```
DROP SEQUENCE dept_deptid_seq;
```

```
sequence DEPT_DEPTID_SEQ dropped.
```


- You must be the owner or have the ALTER privilege for the sequence to modify it. You must be the owner or have the DROP ANY SEQUENCE privilege to remove it.
- Only future sequence numbers are affected by the ALTER SEQUENCE statement.
- The START WITH option cannot be changed by using ALTER SEQUENCE. The sequence must be dropped and re-created to restart the sequence at a different number.
- Some validation is performed. For example, a new MAXVALUE that is less than the current sequence number cannot be imposed.

```
ALTER SEQUENCE dept_deptid_seq
            INCREMENT BY 20
            MAXVALUE 90
            NOCACHE
            NOCYCLE;
```

- The error:

```
SQL Error: ORA-04009: MAXVALUE cannot be made to be less than the current value
04009. 00000 - "MAXVALUE cannot be made to be less than the current value"
*Cause:    the current value exceeds the given MAXVALUE
*Action:   make sure that the new MAXVALUE is larger than the current value
```

Sequence Information

- The `USER_SEQUENCES` view describes all the sequences that you own.
- Verify your sequence values in the `USER_SEQUENCES` data dictionary table.

```
SELECT sequence_name, min_value, max_value,
       increment_by, last_number
FROM user_sequences;
```

	SEQUENCE_NAME	MIN_VALUE	MAX_VALUE	INCREMENT_BY	LAST_NUMBER
1	DEPARTMENTS_SEQ	1	9990	10	280
2	DEPT_DEPTID_SEQ	1	9999	10	290
3	EMPLOYEES_SEQ	1	9999999999999999999999999999	1	207
4	LOCATIONS_SEQ	1	9900	100	3300
5	S1	1	9999999999999999999999999999	1	21

ORACLE

The `USER_SEQUENCES` view describes all the sequences that you own. When you create a sequence, you specify criteria that are stored in the `USER_SEQUENCES` view. The columns in this view are:

- `SEQUENCE_NAME`: Name of the sequence
- `MIN_VALUE`: Minimum value of the sequence
- `MAX_VALUE`: Maximum value of the sequence
- `INCREMENT_BY`: Value by which the sequence is incremented
- `CYCLE_FLAG`: Whether sequence wraps around on reaching the limit
- `ORDER_FLAG`: Whether sequence numbers are generated in order
- `CACHE_SIZE`: Number of sequence numbers to cache
- `LAST_NUMBER`: Last sequence number written to disk. If a sequence uses caching, the number written to disk is the last number placed in the sequence cache. This number is likely to be greater than the last sequence number that was used. The `LAST_NUMBER` column displays the next available sequence number if `NOCACHE` is specified.

After creating your sequence, it is documented in the data dictionary. Because a sequence is a database object, you can identify it in the `USER_OBJECTS` data dictionary table.

You can also confirm the settings of the sequence by selecting from the `USER_SEQUENCES` data dictionary view.

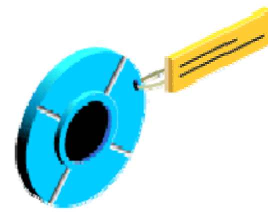
Lesson Agenda

- Overview of sequences:
  - Creating, using, and modifying a sequence
  - Caching sequence values
  - `NEXTVAL` and `CURRVAL` pseudocolumns
  - SQL column defaulting using a sequence
- Overview of synonyms
  - Creating and dropping synonyms
- Overview of indexes
  - Creating indexes
  - Using the `CREATE TABLE` statement
  - Creating function-based indexes
  - Creating multiple indexes on the same set of columns
  - Removing indexes

Synonyms

A synonym:

- Is a database object
- Can be created to give an alternative name to a table or to another database object
- Requires no storage other than its definition in the data dictionary
- Is useful for hiding the identity and location of an underlying schema object



ORACLE

Synonyms are database object that enable you to call a table by another name.

You can create synonyms to give an alternative name to a table or to another database object. For example, you can create a synonym for a table or view, sequence, PL/SQL program unit, user-defined object type, or another synonym.

Because a synonym is simply an alias, it requires no storage other than its definition in the data dictionary.

Synonyms can simplify SQL statements for database users. Synonyms are also useful for hiding the identity and location of an underlying schema object.

Creating a Synonym for an Object

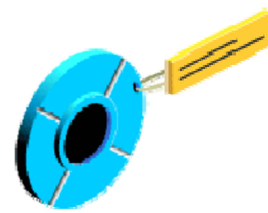
Simplify access to objects by creating a synonym (another name for an object). With synonyms, you can:

- Create an easier reference to a table that is owned by another user
- Shorten lengthy object names

Creates a synonym that is accessible to all users

```
CREATE [PUBLIC] SYNONYM synonym
FOR    object;
```

Identifies the object for which the synonym is created



To refer to a table that is owned by another user, you need to prefix the table name with the name of the user who created it, followed by a period. Creating a synonym eliminates the need to qualify the object name with the schema, and provides you with an alternative name for a table, view, sequence, procedure, or other objects. This method can be especially useful with lengthy object names, such as views.

In the syntax:

`PUBLIC`	Creates a synonym that is accessible to all users
`synonym`	Is the name of the synonym to be created
`object`	Identifies the object for which the synonym is created

Guidelines

- The object cannot be contained in a package.
- A private synonym name must be distinct from all other objects that are owned by the same user.
- To create a `PUBLIC` synonym, you must have the `CREATE PUBLIC SYNONYM` system privilege.

For more information, see the section on "`CREATE SYNONYM`" in *Oracle Database SQL Language Reference* for Oracle Database 12*c*.

Creating and Removing Synonyms

- Create a shortened name for the DEPT_SUM_VU view:

```
CREATE SYNONYM  d_sum
FOR  dept_sum_vu;
```

- Drop a synonym:

```
DROP SYNONYM d_sum;
```


Creating a Synonym

The slide example creates a synonym for the DEPT_SUM_VU view for quicker reference.

The database administrator can create a public synonym that is accessible to all users. The following example creates a public synonym named DEPT for Alice's DEPARTMENTS table:

```
CREATE PUBLIC SYNONYM  dept
FOR    alice.departments;
```

Removing a Synonym

To remove a synonym, use the DROP SYNONYM statement. Only the database administrator can drop a public synonym.

```
DROP PUBLIC SYNONYM  dept;
```

For more information, see the section on "DROP SYNONYM" in *Oracle Database SQL Language Reference* for Oracle Database 12*c*.

Synonym Information

- The USER_SYNONYMS dictionary view describes private synonyms (synonyms that you own).

```
SELECT *
FROM   user_synonyms;
```

	SYNONYM_NAME	TABLE_OWNER	TABLE_NAME	DB_LINK	ORIGIN_CON_ID
1	D_SUM	ORA02	DEPT_SUM_VU	(null)	3

- You can query ALL_SYNONYMS to find out the names of all the synonyms that are available to you and the objects on which these synonyms apply.

ORACLE

The USER_SYNONYMS dictionary view describes private synonyms (synonyms that you own).

You can query this view to find your synonyms. You can query ALL_SYNONYMS to find out the names of all the synonyms that are available to you and the objects on which these synonyms apply.

The columns in this view are:

- SYNONYM_NAME: Name of the synonym
- TABLE_OWNER: Owner of the object that is referenced by the synonym
- TABLE_NAME: Name of the table or view that is referenced by the synonym
- DB_LINK: Name of the database link reference (if any)
- ORIGIN_CON_ID: The ID of the container where the data originates. Refer to the documentation for more information about this.

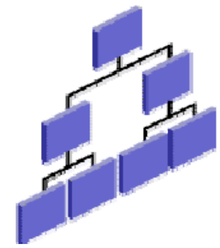
Lesson Agenda

- Overview of sequences:
  - Creating, using, and modifying a sequence
  - Caching sequence values
  - `NEXTVAL` and `CURRVAL` pseudocolumns
  - SQL column defaulting using a sequence
- Overview of synonyms
  - Creating and dropping synonyms
- Overview of indexes
  - Creating indexes
  - Using the `CREATE TABLE` statement
  - Creating function-based indexes
  - Creating multiple indexes on the same set of columns
  - Removing indexes

Indexes

An index:

- Is a schema object
- Can be used by the Oracle Server to speed up the retrieval of rows by using a pointer
- Can reduce disk input/output (I/O) by using a rapid path access method to locate data quickly
- Is dependent on the table that it indexes
- Is used and maintained automatically by the Oracle Server



ORACLE

An Oracle Server index is a schema object that can speed up the retrieval of rows by using a pointer, and improves the performance of some queries. Indexes can be created explicitly or automatically. If you do not have an index on the column, a full table scan occurs.

An index provides direct and fast access to the rows in a table. Its purpose is to reduce disk I/O by using an indexed path to locate data quickly. An index is used and maintained automatically by the Oracle Server. After an index is created, no direct activity is required by the user.

Indexes are logically and physically independent of the data in the objects with which they are associated. This means that they can be created or dropped at any time, and have no effect on the base tables or other indexes.

Note: When you drop a table, the corresponding indexes are also dropped.

For more information, see the section on “Schema Objects: Indexes” in *Oracle Database Concepts 12c Release* 1.

How Are Indexes Created?

- Automatically: A unique index is created automatically when you define a `PRIMARY KEY` or `UNIQUE` constraint in a table definition.

- Manually: You can create unique or nonunique indexes on columns to speed up access to the rows.

You can create two types of indexes.

- **Unique index:** The Oracle Server automatically creates this index when you define a column in a table to have a `PRIMARY KEY` or a `UNIQUE` constraint. The name of the index is the name that is given to the constraint.
- **Nonunique index:** This is an index that a user can create. For example, you can create the `FOREIGN KEY` column index for a join in a query to improve the speed of retrieval.

Note: You can manually create a unique index, but it is recommended that you create a unique constraint, which implicitly creates a unique index.

Creating an Index

- Create an index on one or more columns:

```
CREATE [UNIQUE]INDEX index
ON table (column[, column]...);
```

Name of the column in the table to be indexed

- Improve the speed of query access to the `LAST_NAME` column in the `EMPLOYEES` table:

```
CREATE INDEX emp_last_name_idx
ON employees(last_name);
```

```
index EMP_LAST_NAME_IDX created.
```

ORACLE

Create an index on one or more columns by issuing the `CREATE INDEX` statement.

In the syntax:

- `index` Is the name of the index
- `table` Is the name of the table
- `Column` Is the name of the column in the table to be indexed

Specify `UNIQUE` to indicate that the value of the column (or columns) on which the index is based must be unique. Specify `BITMAP` to indicate that the index is to be created with a bitmap for each distinct key, rather than indexing each row separately. Bitmap indexes store the `rowids` associated with a key value as a bitmap.

For more information, see the section on "`CREATE INDEX`" in *Oracle Database SQL Language Reference* for Oracle Database 12*c*.

CREATE INDEX with the CREATE TABLE Statement

```
CREATE TABLE NEW_EMP
(employee_id NUMBER(6)
             PRIMARY KEY USING INDEX
            (CREATE INDEX emp_id_idx ON
            NEW_EMP(employee_id)),
first_name  VARCHAR2(20),
last_name   VARCHAR2(25));
```

```
table NEW_EMP created.
```

```
SELECT INDEX_NAME, TABLE_NAME
FROM   USER_INDEXES
WHERE  TABLE_NAME = 'NEW_EMP';
```

	INDEX_NAME	TABLE_NAME
1	EMP_ID_IDX	NEW_EMP

In the example in the slide, the CREATE INDEX clause is used with the CREATE TABLE statement to create a PRIMARY KEY index explicitly. You can name your indexes at the time of PRIMARY KEY creation to be different from the name of the PRIMARY KEY constraint.

You can query the USER_INDEXES data dictionary view for information about your indexes.

The following example illustrates the database behavior if the index is not explicitly named:

```
CREATE TABLE EMP_UNNAMED_INDEX
    (employee_id NUMBER(6) PRIMARY KEY ,
     first_name VARCHAR2(20),
     last_name VARCHAR2(25));

SELECT INDEX_NAME, TABLE_NAME
FROM   USER_INDEXES
WHERE  TABLE_NAME = 'EMP_UNNAMED_INDEX';
```

	INDEX_NAME	TABLE_NAME
1	SYS_C0018560	EMP_UNNAMED_INDEX

Observe that the Oracle Server gives a generic name to the index that is created for the PRIMARY KEY column.

You can also use an existing index for your PRIMARY KEY column—for example, when you are expecting a large data load and want to speed up the operation. You may want to disable the constraints while performing the load, and then enable them, in which case having a unique index on the PRIMARY KEY will still cause the data to be verified during the load. Therefore, you can first create a nonunique index on the column that is designated as PRIMARY KEY, and then create the PRIMARY KEY column and specify that it should use the existing index. The following examples illustrate this process:

Step 1: Create the table:

```
CREATE TABLE NEW_EMP2
 (employee_id NUMBER(6),
 first_name  VARCHAR2(20),
 last_name   VARCHAR2(25)
 );
```

Step 2: Create the index:

```
CREATE INDEX emp_id_idx2 ON
  new_emp2(employee_id);
```

Step 3: Create the PRIMARY KEY:

```
ALTER TABLE new_emp2 ADD PRIMARY KEY (employee_id) USING
INDEX emp_id_idx2;
```

Function-Based Indexes

- A function-based index is based on expressions.
- The index expression is built from table columns, constants, SQL functions, and user-defined functions.

```
CREATE INDEX upper_dept_name_idx
ON dept2(UPPER(department_name));
```

```
index UPPER_DEPT_NAME_IDX created.
```

```
SELECT *
FROM   dept2
WHERE  UPPER(department_name) = 'SALES';
```

	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	80	Sales	145	2500

ORACLE

Function-based indexes that are defined with the `UPPER(column_name)` or `LOWER(column_name)` keywords allow non-case-sensitive searches. For example, consider the following index:

```
CREATE INDEX upper_last_name_idx ON emp2 (UPPER(last_name));
```

This facilitates processing queries such as:

```
SELECT * FROM emp2 WHERE UPPER(last_name) = 'KING';
```

The Oracle Server uses the index only when that particular function is used in a query. For example, the following statement may use the index, but without the `WHERE` clause, the Oracle Server may perform a full table scan:

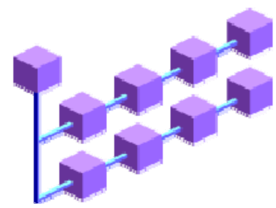
```
SELECT   *
FROM     employees
WHERE    UPPER (last_name) IS NOT NULL
ORDER BY UPPER (last_name);
```

Note: For creating a function-based index, you need the `QUERY REWRITE` system privilege. The `QUERY_REWRITE_ENABLED` initialization parameter must be set to `TRUE` for a function-based index to be used.

The Oracle Server treats indexes with columns marked `DESC` as function-based indexes. The columns marked `DESC` are sorted in descending order.

Creating Multiple Indexes on the Same Set of Columns

- You can create multiple indexes on the same set of columns.
- Multiple indexes can be created on the same set of columns if:
  – The indexes are of different types
  – The indexes use different partitioning
  – The indexes have different uniqueness properties
- Only one of the multiple indexes can be visible at a time.



ORACLE

You can create multiple indexes on the same set of columns if the indexes are of different types, use different partitioning, or have different uniqueness properties. For example, you can create a B-tree index and a bitmap index on the same set of columns.

Similarly, you can create both a unique and a nonunique index on the same set of columns.

When you have multiple indexes on the same set of columns, only one of these indexes can be visible at a time.

Creating Multiple Indexes on the Same Set of Columns: Example

```
CREATE INDEX emp_id_name_ix1
ON employees(employee_id, first_name);
```

```
index EMP_ID_NAME_IX1 created.
```

```
ALTER INDEX emp_id_name_ix1 INVISIBLE;
```

```
index EMP_ID_NAME_IX1 altered.
```

```
CREATE BITMAP INDEX emp_id_name_ix2
ON employees(employee_id, first_name);
```

```
bitmap index EMP_ID_NAME_IX2 created.
```

ORACLE

The code example in the slide shows the creation of a B-tree index, `emp_id_name_ix1`, on the `employee_id` and `first_name` columns of the `employees` table in the `HR` schema. After the creation of the index, it is altered to make it invisible. Then a bitmap index is created on the `employee_id` and `first_name` columns of the `employees` table in the `HR` schema. The bitmap index, `emp_id_name_ix2`, is visible by default.

Index Information

- USER_INDEXES provides information about your indexes.
- USER_IND_COLUMNS describes columns of indexes owned by you and columns of indexes on your tables.

```
SELECT index_name, table_name, uniqueness
FROM   user_indexes
WHERE  table_name = 'EMPLOYEES';
```

	INDEX_NAME	TABLE_NAME	UNIQUENESS
1	EMP_NAME_IX	EMPLOYEES	NONUNIQUE
2	EMP_MANAGER_IX	EMPLOYEES	NONUNIQUE
3	EMP_JOB_IX	EMPLOYEES	NONUNIQUE
4	EMP_DEPARTMENT_IX	EMPLOYEES	NONUNIQUE
5	EMP_EMP_ID_PK	EMPLOYEES	UNIQUE
6	EMP_EMAIL_UK	EMPLOYEES	UNIQUE

ORACLE

You query the USER_INDEXES view to find out the names of your indexes, the table name on which the index is created, and whether the index is unique.

Some of the columns of this view are:

- INDEX_NAME: Name of the index
- INDEX_TYPE: Type of index (NORMAL, BITMAP, FUNCTION-BASED NORMAL, FUNCTION-BASED BITMAP, or DOMAIN)
- TABLE_NAME: Name of the indexed object
- TABLE_OWNER: Owner of the indexed object
- TABLE_TYPE: Type of the indexed object (for example, TABLE, CLUSTER)
- UNIQUENESS: Whether the index is UNIQUE or NONUNIQUE

In the slide example, the USER_INDEXES view is queried to find the name of the index, name of the table on which the index is created, and whether the index is unique.

The USER_IND_COLUMNS dictionary view provides information such as the name of the index, name of the indexed table, name of a column within the index, and the column's position within the index. Use the DESCRIBE command to view the structure of the views.

For example, the emp_test table and LNAME_IDX index are created by using the following code:

```
CREATE TABLE emp_test AS SELECT * FROM employees;
CREATE INDEX lname_idx ON emp_test(last_name);
SELECT index_name, column_name,table_name
 FROM   user_ind_columns
 WHERE  index_name = 'LNAME_IDX';
```

	INDEX_NAME	COLUMN_NAME	TABLE_NAME
1	LNAME_IDX	LAST_NAME	EMP_TEST

Note: For a complete listing and description of the columns in the USER_INDEXES view, see "USER_INDEXES" in the *Oracle Database Reference 12c Release 1*.

Removing an Index

- Remove an index from the data dictionary by using the `DROP INDEX` command:

```
DROP INDEX index;
```

- Remove the `emp_last_name_idx` index from the data dictionary:

```
DROP INDEX emp_last_name_idx;
```

```
index EMP_LAST_NAME_IDX dropped.
```

- To drop an index, you must be the owner of the index or have the `DROP ANY INDEX` privilege.

You cannot modify indexes. To change an index, you must drop it, and then re-create it.

Remove an index definition from the data dictionary by issuing the `DROP INDEX` statement. To drop an index, you must be the owner of the index or have the `DROP ANY INDEX` privilege.

In the syntax, *`index`* is the name of the index.

You can drop an index by using the `ONLINE` keyword.

```
DROP INDEX emp_indx ONLINE;
```

`ONLINE`: Specify `ONLINE` to indicate that DML operations on the table are allowed while dropping the index.

Note: If you drop a table, indexes and constraints are automatically dropped but views remain.

Quiz

Which one of the following clauses of the `CREATE SEQUENCE` statement specifies the interval between the sequence numbers?

a. `START WITH`

b. `INCREMENT BY`

c. `CYCLE`

d. `CACHE`

ORACLE

Answer: b

Quiz

The sequence must be dropped and re-created to restart the sequence at a different number.

a. True

b. False

Answer: a

Quiz

Indexes must be created manually and serve to speed up access to rows in a table.

a. True

b. False

Answer: b

Note: Indexes are designed to speed up query performance. However, not all indexes are created manually. The Oracle Server automatically creates an index when you define a column in a table to have a `PRIMARY KEY` or a `UNIQUE` constraint.

Quiz

Q

You use the following view to find out the names of your indexes, the table name on which the index is created, and whether the index is unique.

a. USER_INDEXES
b. USER_SEQUENCES
c. USER_IND_COLUMNS
d. USER_SYNONYMS

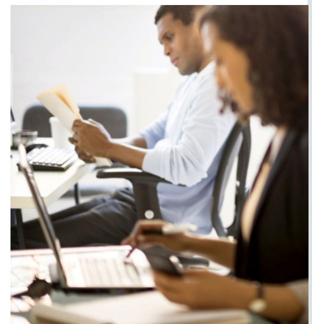
ORACLE

Answer: a

Summary

In this lesson, you should have learned how to:

- Automatically generate sequence numbers by using a sequence generator
- Use synonyms to provide alternative names for objects
- Create indexes to improve the speed of query retrieval
- Find information about your objects through the following dictionary views:
  - USER_SEQUENCES
  - USER_SYNONYMS
  - USER_INDEXES and USER_IND_COLUMNS



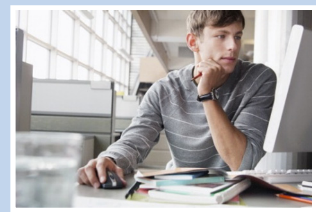
ORACLE

In this lesson, you should have learned about database objects such as sequences, indexes, and synonyms.

Practice 19: Overview

This practice covers the following topics:

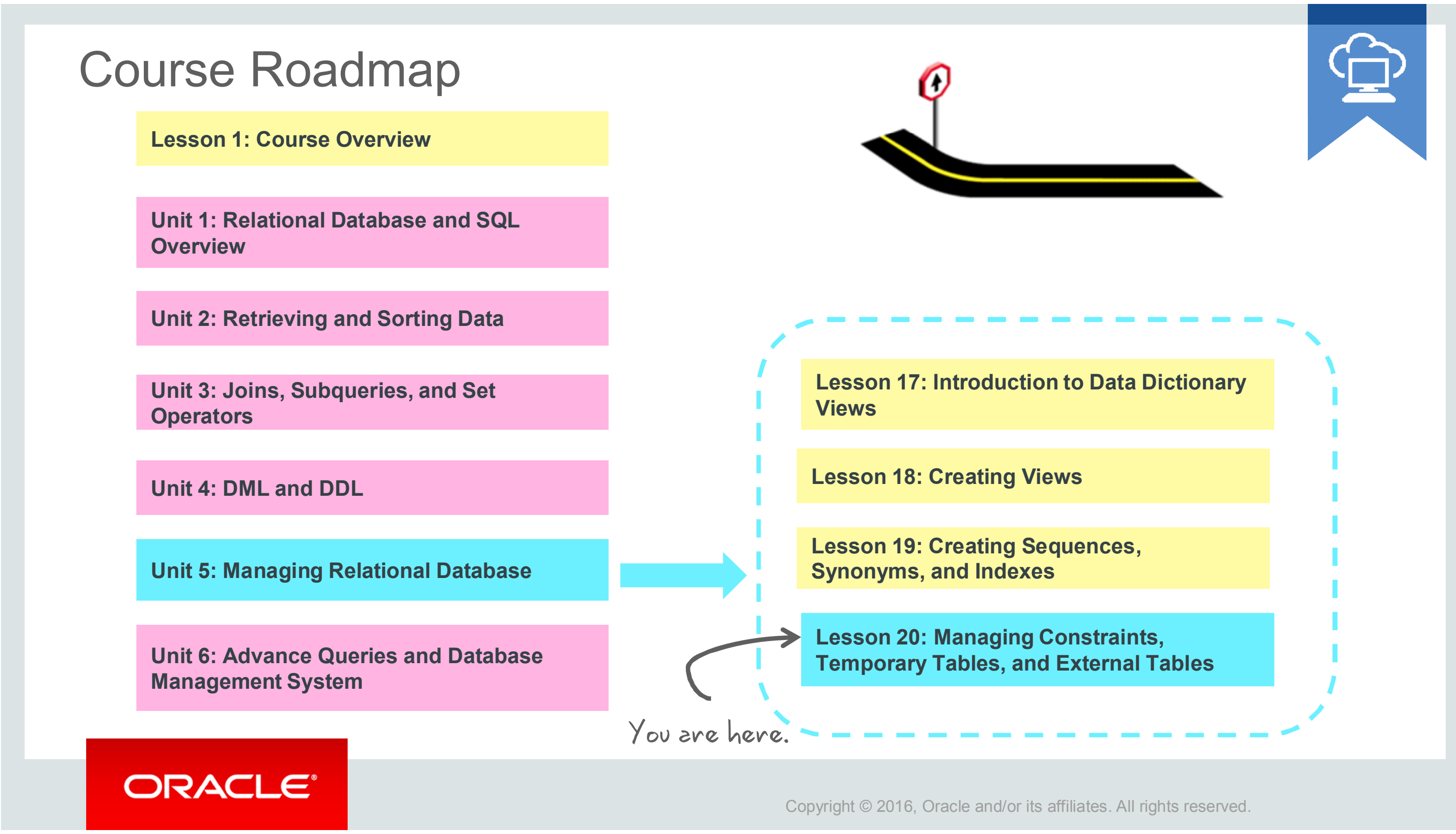
- Creating sequences
- Using sequences
- Querying the dictionary views for sequence information
- Creating synonyms
- Querying the dictionary views for synonyms information
- Creating indexes
- Querying the dictionary views for indexes information



ORACLE

This lesson's practice provides you with a variety of exercises in creating and using a sequence, an index, and a synonym. You also learn how to query the data dictionary views for sequence, synonym, and index information.


Lesson 20: Managing Constraints, Temporary Tables, and External Tables
ORACLE
Copyright © 2016, Oracle and/or its affiliates. All rights reserved.



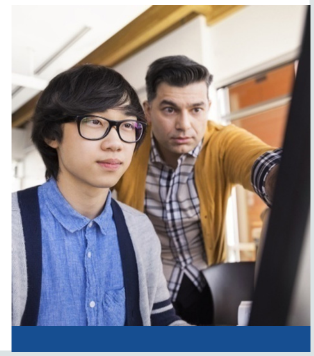
In Unit 5, you are introduced to views. You learn to:

- Query data dictionary views
- Create sequences, synonyms, and indexes
- Manage constraints and tables

Objectives

After completing this lesson, you should be able to:

- Manage constraints
- Create and use temporary tables
- Create external tables



This lesson contains information about managing constraints. You also learn about temporary tables and external tables.

Lesson Agenda

- Managing constraints:
  – Adding and dropping a constraint
  – Enabling and disabling a constraint
  – Deferring constraints
- Creating and using temporary tables
- Creating external tables

Adding a Constraint Syntax

Use the ALTER TABLE statement to:

- Add or drop a constraint
- Enable or disable constraints
- Add a NOT NULL constraint by using the MODIFY clause

```
ALTER TABLE <table_name>
ADD [CONSTRAINT <constraint_name>]
type (<column_name>);
```

You can add a constraint for existing tables by using the ALTER TABLE statement with the ADD clause.

In the syntax:

table	Is the name of the table
constraint	Is the name of the constraint
type	Is the constraint type
column	Is the name of the column affected by the constraint

The constraint name syntax is optional, although recommended. If you do not name your constraints, the system generates constraint names.

Guidelines

- You can add, drop, enable, or disable a constraint, but you cannot modify its structure.
- You can add a NOT NULL constraint to an existing column by using the MODIFY clause of the ALTER TABLE statement.

Note: You can define a NOT NULL column only if the table is empty or if the column has a value for every row.

Adding a Constraint

Add a FOREIGN KEY constraint to the EMP2 table indicating that a manager must already exist as a valid employee in the EMP2 table.

```
ALTER TABLE emp2
MODIFY employee_id PRIMARY KEY;
```

```
ALTER TABLE emp2
ADD CONSTRAINT emp_mgr_fk
  FOREIGN KEY(manager_id)
  REFERENCES emp2(employee_id);
```

ORACLE®

The first example in the slide modifies the EMP2 table to add a PRIMARY KEY constraint on the EMPLOYEE_ID column. Note that because no constraint name is provided, the constraint is automatically named by the Oracle Server. The second example in the slide creates a FOREIGN KEY constraint on the EMP2 table. The constraint ensures that a manager exists as a valid employee in the EMP2 table.

Dropping a Constraint

- The DROP CONSTRAINT clause enables you to drop an integrity constraint from a database.
- Remove the manager constraint from the EMP2 table:

```
ALTER TABLE emp2
DROP CONSTRAINT emp_mgr_fk;
```

- Remove the PRIMARY KEY constraint on the DEPT2 table and drop the associated FOREIGN KEY constraint on the EMP2.DEPARTMENT_ID column:

```
ALTER TABLE emp2
DROP PRIMARY KEY CASCADE;
```

ORACLE

The DROP CONSTRAINT clause enables you to drop an integrity constraint from a database.

To drop a constraint, you can identify the constraint name from the USER_CONSTRAINTS and USER_CONS_COLUMNS data dictionary views. Then use the ALTER TABLE statement with the DROP clause. The CASCADE option of the DROP clause causes any dependent constraints also to be dropped.

Syntax

```
ALTER TABLE    table
 DROP  PRIMARY KEY | UNIQUE (column) |
       CONSTRAINT   constraint   [CASCADE];
```

In the syntax:

table	Is the name of the table
column	Is the name of the column affected by the constraint
constraint	Is the name of the constraint

When you drop an integrity constraint, that constraint is no longer enforced by the Oracle Server and is no longer available in the data dictionary.

Dropping a CONSTRAINT ONLINE

You can specify the ONLINE keyword to indicate that DML operations on the table are allowed while dropping the constraint.

```
ALTER TABLE myemp2
DROP CONSTRAINT emp_name_pk ONLINE;
```

You can also drop a constraint by using an ONLINE keyword. Use the ALTER TABLE statement with the DROP clause. The ONLINE option of the DROP clause indicates that DML operations on the table are allowed while dropping the constraint.

ON DELETE Clause

- Use the ON DELETE CASCADE clause to delete child rows when a parent key is deleted:

```
ALTER TABLE dept2 ADD CONSTRAINT dept_lc_fk
FOREIGN KEY (location_id)
REFERENCES locations(location_id) ON DELETE CASCADE;
```

- Use the ON DELETE SET NULL clause to set the child row value to null when a parent key is deleted:

```
ALTER TABLE emp2 ADD CONSTRAINT emp_dt_fk
FOREIGN KEY (Department_id)
REFERENCES departments(department_id) ON DELETE SET NULL;
```


ON DELETE

By using the ON DELETE clause, you can determine how Oracle Database handles referential integrity if you remove a referenced primary or unique key value.

ON DELETE CASCADE

The ON DELETE CASCADE action allows parent key data that is referenced from the child table to be deleted, but not updated. When data in the parent key is deleted, all the rows in the child table that depend on the deleted parent key values are also deleted. To specify this referential action, include the ON DELETE CASCADE option in the definition of the FOREIGN KEY constraint.

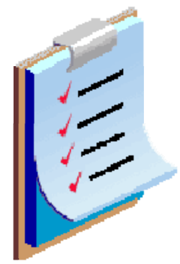
ON DELETE SET NULL

When data in the parent key is deleted, the ON DELETE SET NULL action causes all the rows in the child table that depend on the deleted parent key value to be converted to null.

If you omit this clause, Oracle does not allow you to delete referenced key values in the parent table that have dependent rows in the child table.

Cascading Constraints

- The CASCADE CONSTRAINTS clause:
  - Is used along with the DROP COLUMN clause
  - Drops all referential integrity constraints that refer to the PRIMARY and UNIQUE keys defined on the dropped columns
  - Drops all multicolumn constraints defined on the dropped columns



ORACLE

This statement illustrates the usage of the CASCADE CONSTRAINTS clause. Assume that the TEST1 table is created as follows:

```
CREATE TABLE test1 (
  col1_pk NUMBER PRIMARY KEY,
  col2_fk NUMBER,
  col1 NUMBER,
  col2 NUMBER,
  CONSTRAINT fk_constraint FOREIGN KEY (col2_fk) REFERENCES
    test1,
  CONSTRAINT ck1 CHECK (col1_pk > 0 and col1 > 0),
  CONSTRAINT ck2 CHECK (col2_fk > 0));
```

An error is returned for the following statements:

`ALTER TABLE test1 DROP (col1_pk);` —col1_pk is a parent key.

`ALTER TABLE test1 DROP (col1);` —col1 is referenced by the multicolumn constraint, ck1.

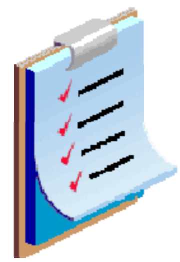
```
Error starting at line : 9 in command -
ALTER TABLE test1 DROP (col1_pk)
Error report -
SQL Error: ORA-12992: cannot drop parent key column
12992. 00000 -  "cannot drop parent key column"
*Cause:    An attempt was made to drop a parent key column.
*Action:   Drop all constraints referencing the parent key column, or
           specify CASCADE CONSTRAINTS in statement.
Error starting at line : 11 in command -
ALTER TABLE test1 DROP (col1)
Error report -
SQL Error: ORA-12991: column is referenced in a multi-column constraint
12991. 00000 -  "column is referenced in a multi-column constraint"
*Cause:    An attempt was made to drop a column referenced by some
           constraints.
*Action:   Drop all constraints referencing the dropped column or
           specify CASCADE CONSTRAINTS in statement.
```

Cascading Constraints

Example:

```
ALTER TABLE emp2
DROP COLUMN employee_id CASCADE CONSTRAINTS;
```

Drops the `employee_id` column, the `PRIMARY KEY` constraint, and any `FOREIGN KEY` constraints referencing the `PRIMARY KEY` constraint for the `EMP2` table



ORACLE

In the slide example, the `ALTER TABLE` statement drops the `EMPLOYEE_ID` column, the `PRIMARY KEY` constraint, and any `FOREIGN KEY` constraints referencing the `PRIMARY KEY` constraint for the `EMP2` table.

If all the columns referenced by the constraints defined on the dropped columns are also dropped, `CASCADE CONSTRAINTS` is not required. For example, assuming that no other referential constraints from other tables refer to the `COL1_PK` column, it is valid to submit the following statement without the `CASCADE CONSTRAINTS` clause for the `TEST1` table created on the previous page:

```
ALTER TABLE test1 DROP (col1_pk, col2_fk, col1);
```

- Enabling a `PRIMARY KEY` constraint that was disabled with the `CASCADE` option does not enable any `FOREIGN KEY`s that are dependent on the `PRIMARY KEY`.
- To enable a `UNIQUE` or `PRIMARY KEY` constraint, you must have the privileges necessary to create an index on the table.

Renaming Table Columns and Constraints

- Use the RENAME TABLE clause of the ALTER TABLE statement to rename tables.

a
```
ALTER TABLE marketing RENAME to new_marketing;
```

- Use the RENAME COLUMN clause of the ALTER TABLE statement to rename table columns.

b
```
ALTER TABLE new_marketing RENAME COLUMN team_id
TO id;
```

- Use the RENAME CONSTRAINT clause of the ALTER TABLE statement to rename any existing constraint for a table.

c
```
ALTER TABLE new_marketing RENAME CONSTRAINT mktg_pk
TO new_mktg_pk;
```

ORACLE

The RENAME TABLE clause allows you to rename an existing table in any schema (except the schema *SYS*).To rename a table, you must either be the database owner or the table owner.

When you rename a table column, the new name must not conflict with the name of any existing column in the table. You cannot use any other clauses in conjunction with the RENAME COLUMN clause.

The slide examples use the marketing table with the PRIMARY KEY mktg_pk defined on the id column.

```
CREATE TABLE marketing (team_id  NUMBER(10),
                        target   VARCHAR2(50),
CONSTRAINT mktg_pk PRIMARY KEY(team_id));
```

Example "**a**" shows that the marketing table is renamed new_marketing. Example "**b**" shows that the id column of the new_marketing table is renamed mktg_id and example "**c**" shows that mktg_pk is renamed new_mktg_pk.

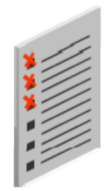
When you rename any existing constraint for a table, the new name must not conflict with any of your existing constraint names. You can use the RENAME CONSTRAINT clause to rename system-generated constraint names.

Disabling Constraints

- Execute the DISABLE clause of the ALTER TABLE statement to deactivate an integrity constraint.
- Apply the CASCADE option to disable the primary key and it will disable all dependent FOREIGN KEY constraints automatically as well.

```
ALTER TABLE  emp2
DISABLE CONSTRAINTS emp_dt_fk;
```

```
ALTER TABLE dept3
DISABLE primary key CASCADE;
```



ORACLE

You can disable a constraint, without dropping it or re-creating it, by using the ALTER TABLE statement with the DISABLE clause. You can also disable the primary key or unique key by using the CASCADE option.

Syntax

```
ALTER  TABLE  table
DISABLE CONSTRAINT constraint  [CASCADE];
```

In the syntax:

table	Is the name of the table
constraint	Is the name of the constraint

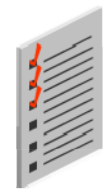
Guidelines

- You can use the DISABLE clause in both the CREATE TABLE statement and the ALTER TABLE statement.
- The CASCADE clause disables dependent integrity constraints.
- Disabling a UNIQUE or PRIMARY KEY constraint removes the unique index.

Enabling Constraints

- Activate an integrity constraint that is currently disabled in the table definition by using the ENABLE clause.

```
ALTER TABLE       emp2
ENABLE CONSTRAINT emp_dt_fk;
```



- A UNIQUE index is automatically created if you enable a UNIQUE key or a PRIMARY KEY constraint.

You can enable a constraint without dropping it or re-creating it by using the ALTER TABLE statement with the ENABLE clause.

Syntax

```
ALTER  TABLE   table
ENABLE CONSTRAINT constraint;
```

In the syntax:

table	Is the name of the table
constraint	Is the name of the constraint

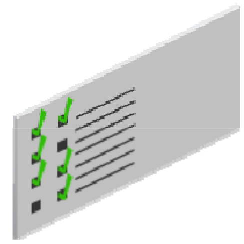
Guidelines

- If you enable a constraint, the constraint applies to all the data in the table. All the data in the table must comply with the constraint.
- If you enable a UNIQUE key or a PRIMARY KEY constraint, a UNIQUE or PRIMARY KEY index is created automatically. If an index already exists, it can be used by these keys.
- You can use the ENABLE clause in both the CREATE TABLE statement and the ALTER TABLE statement.

Constraint States

An integrity constraint defined on a table can be in one of the following states:

- ENABLE VALIDATE
- ENABLE NOVALIDATE
- DISABLE VALIDATE
- DISABLE NOVALIDATE



```
ALTER TABLE dept3
ENABLE NOVALIDATE PRIMARY KEY;
```


You can enable or disable integrity constraints at the table level by using the `CREATE TABLE` or `ALTER TABLE` statement. You can also set constraints to `VALIDATE` or `NOVALIDATE`, in any combination with `ENABLE` or `DISABLE`, where:

- `ENABLE` ensures that all incoming data conforms to the constraint
- `DISABLE` allows incoming data, regardless of whether it conforms to the constraint
- `VALIDATE` ensures that existing data conforms to the constraint
- `NOVALIDATE` means that some existing data may not conform to the constraint

`ENABLE VALIDATE` is the same as `ENABLE`. The constraint is checked and is guaranteed to hold for all rows. `ENABLE NOVALIDATE` means that the constraint is checked, but it does not have to be true for all rows. This allows existing rows to violate the constraint, while ensuring that all new or modified rows are valid. In an `ALTER TABLE` statement, `ENABLE NOVALIDATE` resumes constraint checking on disabled constraints without first validating all the data in the table. `DISABLE NOVALIDATE` is the same as `DISABLE`. The constraint is not checked and is not necessarily true. `DISABLE VALIDATE` disables the constraint, drops the index on the constraint, and disallows any modification of the constrained columns.

Deferring Constraints

Constraints can have the following attributes:

- DEFERRABLE or NOT DEFERRABLE
- INITIALLY DEFERRED or INITIALLY IMMEDIATE

```
ALTER TABLE dept4
ADD CONSTRAINT dept4_id_pk
PRIMARY KEY (department_id)
DEFERRABLE INITIALLY DEFERRED;
```

Deferring constraint on creation

```
SET CONSTRAINTS dept4_id_pk IMMEDIATE;
```

Changing a specific constraint attribute

```
ALTER SESSION
SET CONSTRAINTS= IMMEDIATE;
```

Changing all constraints for a session

ORACLE

You can defer checking constraints for validity until the end of the transaction. A constraint is deferred if the system does not check whether the constraint is satisfied, until a COMMIT statement is submitted. If a deferred constraint is violated, the database returns an error and the transaction is not committed; it is rolled back. If a constraint is immediate (not deferred), it is checked at the end of each statement. If it is violated, the statement is rolled back immediately. If a constraint causes an action (for example, DELETE CASCADE), that action is always taken as part of the statement that caused it, whether the constraint is deferred or immediate. Use the SET CONSTRAINTS statement to specify, for a particular transaction, whether a deferrable constraint is checked following each data manipulation language (DML) statement or when the transaction is committed. To create deferrable constraints, you must create a nonunique index for that constraint.

You can define constraints as either deferrable or NOT DEFERRABLE (default), and either initially deferred or INITIALLY IMMEDIATE (default). These attributes can be different for each constraint.

Usage scenario: Company policy dictates that department number 40 should be changed to 45. Changing the DEPARTMENT_ID column affects the employees assigned to this department. Therefore, you make the PRIMARY KEY and FOREIGN KEYs deferrable and initially deferred. You update both department and employee information, and at the time of commit, all the rows are validated.

Difference Between INITIALLY DEFERRED and INITIALLY IMMEDIATE

INITIALLY DEFERRED	Waits until the transaction ends to check the constraint
INITIALLY IMMEDIATE	Checks the constraint at the end of the statement execution

```
CREATE TABLE emp_new_sal (salary NUMBER
        CONSTRAINT sal_ck
        CHECK (salary > 100)
        DEFERRABLE INITIALLY IMMEDIATE);
```

ORACLE

A constraint that is defined as deferrable can be specified as either INITIALLY DEFERRED or INITIALLY IMMEDIATE. The INITIALLY IMMEDIATE clause is the default.

In the slide example:

- The sal_ck constraint is created as DEFERRABLE INITIALLY IMMEDIATE

After creating the emp_new_sal table, as shown in the slide, you attempt to insert values into the table and observe the results.

Example 1: Insert a row that violates sal_ck. In the CREATE TABLE statement, sal_ck is specified as an initially immediate constraint. This means that the constraint is verified immediately after the INSERT statement and you observe an error.

```
INSERT INTO emp_new_sal VALUES(90);
```

```
Error report -
SQL Error: ORA-02290: check constraint (ORA02.SAL_CK) 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.
```

Example 2: In the following CREATE TABLE statement, bonus_ck is specified as deferrable and also initially deferred constraint. Insert a row that violates bonus_ck. Observe that the constraint is not verified until you COMMIT or set the constraint state back to immediate.

```
CREATE TABLE emp_new_bonus (
bonus NUMBER
CONSTRAINT bonus_ck
CHECK (bonus > 0 )
DEFERRABLE INITIALLY DEFERRED );
INSERT INTO emp_new_bonus VALUES(-1);
```

```
1 row inserted.
```

The row insertion is successful. But you observe an error when you commit the transaction.

```
COMMIT;
```

```
SQL Error: ORA-02091: transaction rolled back
ORA-02290: check constraint (ORA03.BONUS_CK) violated
02091. 00000 -  "transaction rolled back"
```

The commit failed due to constraint violation. Therefore, at this point, the transaction is rolled back by the database.

Set the DEFERRED status to all constraints that can be deferred. Note that you can also set the DEFERRED status to a single constraint if required.

```
SET CONSTRAINTS ALL DEFERRED;
```

Now, if you attempt to insert a row that violates the sal_ck or the bonus_ck constraint, the statement is executed successfully. However, you observe an error when you commit the transaction. The transaction fails and is rolled back. This is because the constraints are checked upon COMMIT.

You can set the IMMEDIATE status for the constraints that were set as DEFERRED .

```
SET CONSTRAINTS ALL IMMEDIATE;
```

You observe an error if you attempt to insert a row that violates either sal_ck or bonus_ck.

Note: If you create a table without specifying constraint deferability, the constraint is checked immediately at the end of each statement. For example, with the CREATE TABLE statement of the newemp_details table, if you do not specify the newemp_det_pk constraint deferability, the constraint is checked immediately.

```
CREATE TABLE newemp_details(emp_id NUMBER, emp_name
VARCHAR2(20),CONSTRAINT newemp_det_pk PRIMARY KEY(emp_id));
```

When you attempt to defer the newemp_det_pk constraint that is not deferrable, you observe the following error:

```
SET CONSTRAINT newemp_det_pk DEFERRED;
```

```
Error report -
SQL Error: ORA-02447: cannot defer a constraint that is not deferrable
02447. 00000 -  "cannot defer a constraint that is not deferrable"
*Cause:    An attempt was made to defer a nondeferrable constraint
*Action:   Drop the constraint and create a new one that is deferrable
```

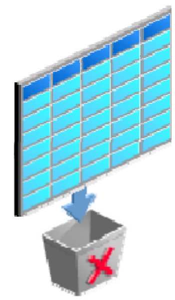
DROP TABLE … PURGE

When you drop a table:

- It is renamed and placed in the recycle bin
- Space is not released immediately
- Can be recovered by using the FLASHBACK TABLE statement

With the PURGE clause, the table is not placed in the recycle bin. It is dropped and the space associated with it is released in a single step.

```
DROP TABLE emp_new_sal PURGE;
```



Oracle Database provides a feature for dropping tables. When you drop a table, the database does not immediately release the space associated with the table. Rather, the database renames the table and places it in a recycle bin, where it can later be recovered with the FLASHBACK TABLE statement if you find that you dropped the table in error. If you want to immediately release the space associated with the table at the time you issue the DROP TABLE statement, include the PURGE clause as shown in the statement in the slide.

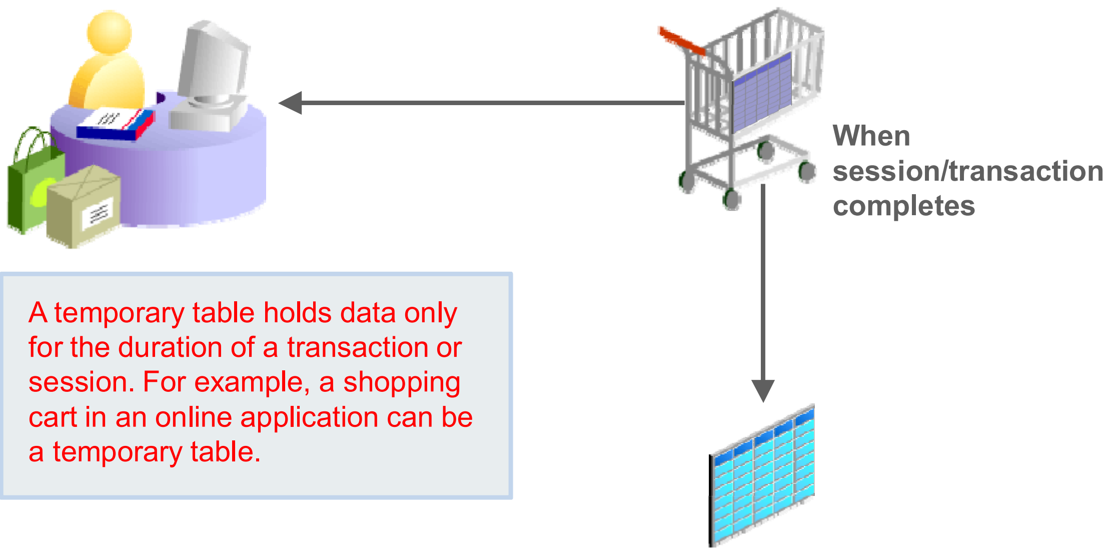
Specify PURGE only if you want to drop the table and release the space associated with it in a single step. If you specify PURGE, the database does not place the table and its dependent objects into the recycle bin.

Using this clause is equivalent to first dropping the table, and then purging it from the recycle bin. This clause saves you one step in the process. It also provides enhanced security if you want to prevent sensitive material from appearing in the recycle bin.

Lesson Agenda

- Managing constraints:
  - Adding and dropping a constraint
  - Enabling and disabling a constraint
  - Deferring constraints
- Creating and using temporary tables
- Creating external tables

Temporary Tables



A temporary table holds data only for the duration of a transaction or session. For example, a shopping cart in an online application can be a temporary table.

ORACLE

A temporary table is a table that holds data that exists only for the duration of a transaction or session. Data in a temporary table is private to the session, which means that each session can see and modify only its own data.

Temporary tables are useful in applications where a result set must be buffered. For example, a shopping cart in an online application can be a temporary table. Each item is represented by a row in the temporary table. While you are shopping in an online store, you can keep on adding or removing items from your cart. During the session, this cart data is private. After you finalize your shopping and make the payments, the application moves the row for the chosen cart to a permanent table. At the end of the session, the data in the temporary table is automatically dropped.

Because temporary tables are statically defined, you can create indexes for them. The indexes that are created on temporary tables are also temporary. The data in the index has the same session or transaction scope as the data in the temporary table. You can also create a view or trigger on a temporary table.

Creating a Temporary Table

```
CREATE GLOBAL TEMPORARY TABLE cart(n NUMBER,d DATE)
ON COMMIT DELETE ROWS;
```

1

```
CREATE GLOBAL TEMPORARY TABLE emp_details
ON COMMIT PRESERVE ROWS AS
      SELECT * FROM employees
                WHERE hire_date = SYSDATE;
```

2

ORACLE

To create a temporary table, you can use the following command:

```
CREATE GLOBAL TEMPORARY TABLE tablename
ON COMMIT [PRESERVE | DELETE] ROWS
```

By associating one of the following settings with the `ON COMMIT` clause, you can decide whether the data in the temporary table is transaction-specific (default) or session-specific.

1. `DELETE ROWS`: As shown in example 1 in the slide, the `DELETE ROWS` setting creates a temporary table that is transaction-specific. A session becomes bound to the temporary table with a transaction's first insert into the table. The binding goes away at the end of the transaction. The database truncates the table (deletes all rows) after each commit.
2. `PRESERVE ROWS`: As shown in example 2 in the slide, the `PRESERVE ROWS` setting creates a temporary table that is session-specific. Each HR representative session can store its own employees data for the day in the table. When an HR person performs the first insert on the `emp_details` table, his or her session gets bound to the `emp_details` table. This binding goes away at the end of the session or by issuing a `TRUNCATE` of the table in the session. The database truncates the table when you terminate the session.

When you create a temporary table in an Oracle database, you create a static table definition. Like permanent tables, temporary tables are defined in the data dictionary. However, temporary tables and their indexes do not automatically allocate a segment when created. Instead, temporary segments are allocated when data is first inserted. Until data is loaded in a session, the table appears empty.

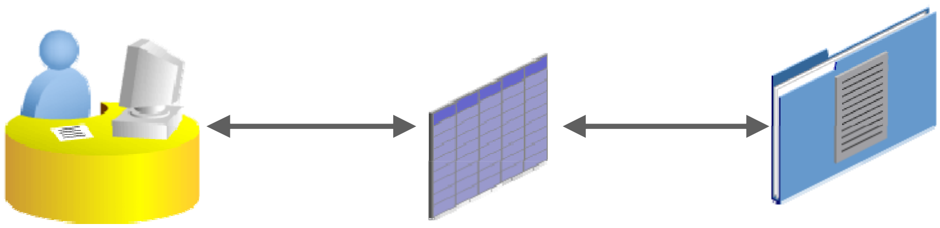
Lesson Agenda

- Managing constraints:
  - Adding and dropping a constraint
  - Enabling and disabling a constraint
  - Deferring constraints
- Creating and using temporary tables
- Creating external tables

External Tables

- Are read-only tables whose metadata is stored in the database but the data is stored externally in flat files
- Can be queried and joined directly and in parallel without the need for loading the data in the database

No DML operations are possible on external tables.



An external table is a read-only table whose metadata is stored in the database but whose data is stored outside the database. This external table definition can be thought of as a view that is used for running any SQL query against external data without requiring that the external data first be loaded into the database. The external table data can be queried and joined directly and in parallel without requiring that the external data first be loaded in the database. You can use SQL, PL/SQL, and Java to query the data in an external table. External tables are useful for querying flat files.

The main difference between external tables and regular tables is that externally organized tables are read-only. No DML operations are possible, and no indexes can be created on them. However, you can create an external table, and thus unload data, by using the `CREATE TABLE AS SELECT` command.

The Oracle Server provides two major access drivers for external tables. One, the loader access driver (or `ORACLE_LOADER`), is used for reading data from external files whose format can be interpreted by the SQL*Loader utility. Note that not all SQL*Loader functionality is supported with external tables. The `ORACLE_DATAPUMP` access driver can be used to both import and export data by using a platform-independent format. The `ORACLE_DATAPUMP` access driver writes rows from a `SELECT` statement to be loaded into an external table as part of a `CREATE TABLE ...ORGANIZATION EXTERNAL...AS SELECT` statement. You can then use `SELECT` to read data out of that data file. You can also create an external table definition on another system and use that data file. This allows data to be moved between Oracle databases.

Creating a Directory for the External Table

Create a DIRECTORY object that corresponds to the directory on the file system where the external data source resides.

```
CREATE OR REPLACE DIRECTORY emp_dir
AS '/…/emp_dir';

GRANT READ ON DIRECTORY emp_dir TO <username>;
```

Use the CREATE DIRECTORY statement to create a directory object. A directory object specifies an alias for a directory on the server's file system where an external data source resides. You can use directory names when referring to an external data source, rather than hard code the operating system path name, for greater file management flexibility.

You must have the CREATE ANY DIRECTORY system privileges to create directories. When you create a directory, you are automatically granted the READ and WRITE object privileges and can grant READ and WRITE privileges to other users and roles. The DBA can also grant these privileges to other users and roles.

A user needs READ privileges on the directory used by the external table in order to access the flat file and WRITE privileges for writing to the log, bad, and discard files.

In addition, a WRITE privilege is necessary when the external table framework is being used to unload data.

Oracle also provides the ORACLE_DATAPUMP type, with which you can unload data (that is, read data from a table in the database and insert it into an external table), and then reload it into an Oracle database. This is a one-time operation that can be performed when the table is created. After the creation and initial population, you cannot update, insert, or delete any rows.

Note: The emp.dat file is saved at /home/oracle/emp_dir folder location on your database file system. A directory object emp_dir is already created and you have been granted READ and WRITE privileges on the same.

Syntax

```
CREATE [OR REPLACE] DIRECTORY AS 'path_name';
```

In the syntax:

`OR REPLACE`	Specify `OR REPLACE` to re-create the directory database object if it already exists. You can use this clause to change the definition of an existing directory without dropping, re-creating, and regranting the database object privileges that were previously granted on the directory. Users who were previously granted privileges on a redefined directory can continue to access the directory without requiring that the privileges be regranted.
`directory`	Specify the name of the directory object to be created. The maximum length of the directory name is 30 bytes. You cannot qualify a directory object with a schema name.
`'path_name'`	Specify the full path name of the operating system directory to be accessed. The path name is case-sensitive.

Creating an External Table

Use the ORGANIZATION EXTERNAL clause to create external tables.

```
CREATE TABLE <table_name>
    ( <col_name> <datatype>, … )
ORGANIZATION EXTERNAL
    (TYPE <access_driver_type>
     DEFAULT DIRECTORY <directory_name>
     ACCESS PARAMETERS
       (… ) )
      LOCATION ('<location_specifier>')
REJECT LIMIT [0 | <number> | UNLIMITED];
```

Indicates the access driver, ORACLE_LOADER(default) or ORACLE_DATAPUMP

Enables you to assign values for specific parameters of the access driver used

Specify the directory object that references the external data source.

ORACLE

You create external tables by using the ORGANIZATION EXTERNAL clause of the CREATE TABLE statement. You are not, in fact, creating a table. Rather, you are creating metadata in the data dictionary that you can use to access external data. You use the ORGANIZATION clause to specify the order in which the data rows of the table are stored. By specifying EXTERNAL in the ORGANIZATION clause, you indicate that the table is a read-only table located outside the database. Note that the external files must already exist outside the database.

TYPE *<access_driver_type>* indicates the access driver of the external table. The access driver is the application programming interface (API) that interprets the external data for the database. If you do not specify TYPE, Oracle uses the default access driver, ORACLE_LOADER. The other option is ORACLE_DATAPUMP.

You use the DEFAULT DIRECTORY clause to specify one or more Oracle database directory objects that correspond to directories on the file system where the external data sources may reside.

The optional ACCESS PARAMETERS clause enables you to assign values to the parameters of the specific access driver for this external table.

Use the `LOCATION` clause to specify one external locator for each external data source. Usually, *<location_specifier>* is a file, but it need not be.

The `REJECT LIMIT` clause enables you to specify how many conversion errors can occur during a query of the external data before an Oracle error is returned and the query is aborted. The default value is 0.

The syntax for using the `ORACLE_DATAPUMP` access driver is as follows:

```
CREATE TABLE <ext_table_name>
ORGANIZATION EXTERNAL (TYPE ORACLE_DATAPUMP
                       DEFAULT DIRECTORY …
                       ACCESS PARAMETERS (… )
                       LOCATION (…)
                       PARALLEL 4
                       REJECT LIMIT UNLIMITED
AS
SELECT * FROM <table_name>;
```

An external table does not describe any data that is stored in the database. It does not describe how data is stored in the external source. Instead, it describes how the external table layer must present the data to the server. It is the responsibility of the access driver and the external table layer to do the necessary transformations required on the data in the data file so that it matches the external table definition.

When the database server accesses data in an external source, it calls the appropriate access driver to get the data from the external source in a form that the database server expects.

It is important to remember that the description of data in the data source is separate from the definition of the external table. The source file can contain more or fewer fields than there are columns in the table. Also, the data types for fields in the data source can be different from the columns in the table. The access driver takes care of ensuring that the data from the data source is processed so that it matches the definition of the external table.

Creating an External Table by Using ORACLE_LOADER

```
CREATE TABLE oldemp (fname char(25), lname CHAR(25))
ORGANIZATION EXTERNAL
(TYPE ORACLE_LOADER
 DEFAULT DIRECTORY emp_dir
 ACCESS PARAMETERS
 (RECORDS DELIMITED BY NEWLINE
  FIELDS(fname POSITION ( 1:20) CHAR,
  lname POSITION (22:41) CHAR))
 LOCATION ('emp.dat'));
```

emp_dir is the directory object that is created by using the CREATE DIRECTORY statement.

ORACLE

Assume that there is a flat file that has records in the following format:

```
10,jones,11-Dec-1934
20,smith,12-Jun-1972
```

Records are delimited by new lines. The file is `/home/oracle/emp_dir/emp.dat`.

To convert this file as the data source for an external table whose metadata will reside in the database, you must perform the following steps:

1. Create a directory object, `emp_dir`, as follows:
   `CREATE DIRECTORY emp_dir AS '/home/oracle/emp_dir' ;`
2. Run the `CREATE TABLE` command shown in the slide.

The example in the slide illustrates the table specification to create an external table for the file, `emp.dat`, that is located at `/home/oracle/emp_dir` folder and is referenced by the `emp_dir` directory object.

After the `CREATE TABLE` command executes successfully, the `OLDEMP` external table can be described and queried in the same way as a relational table.

```
SELECT * FROM oldemp;
```

	FNAME	LNAME
1	onstantin	elles
2	Harry	Pacino
3	Manisha	Taylor
4	Harrison	Sutherland
5	Matthias	MacGraw

Note: A directory object with the name `emp_dir` already exists in this course setup. To run the `CREATE DIRECTORY` code shown above, use a different name.

In the example, the TYPE specification is given only to illustrate its use. ORACLE_LOADER is the default access driver if not specified. The ACCESS PARAMETERS option provides values to the parameters of the specific access driver, which are interpreted by the access driver, not by the Oracle Server.

Note: For this course, the emp_dir directory object has already been created. However, if you want you can create a new directory object pointing to the same location, /home/oracle/emp_dir and run the CREATE TABLE command shown in the slide. You do not need to grant yourself the READ privileges because you are the owner of the directory object.

You can also perform the unload and reload operations with external tables by using the ORACLE_DATAPUMP access driver.

The following example illustrates the table specification to create an external table by using the ORACLE_DATAPUMP access driver. Data is then populated into the two files: emp1.exp and emp2.exp.

To populate data read from the EMPLOYEES table into an external table, you must run the following CREATE TABLE command:

```
CREATE TABLE emp_ext
  (employee_id, first_name, last_name)
   ORGANIZATION EXTERNAL
    (
     TYPE ORACLE_DATAPUMP
     DEFAULT DIRECTORY emp_dir
     LOCATION
      ('emp1.exp','emp2.exp')
    )
   PARALLEL
AS
SELECT employee_id, first_name, last_name
FROM   employees;
```

You can query the external table by executing the following code:

```
SELECT * FROM emp_ext;
```

Note: In the context of external tables, loading data refers to the act of data being read from an external table and loaded into a table in the database. Unloading data refers to the act of reading data from a table and inserting it into an external table.

Quiz

Which one of the following clauses would you use with the `DROP COLUMN` statement to drop all referential integrity constraints that refer to the `PRIMARY` and `UNIQUE` keys defined on the dropped columns?

a. `ON DELETE CASCADE`

b. `CASCADE CONSTRAINTS`

Answer: b

Quiz

`ENABLE NOVALIDATE` allows existing rows to violate the constraint, while ensuring that all new or modified rows are valid.

a. True
b. False

Answer: a

Quiz

You can decide whether the data in a temporary table is transaction-specific (default) or session-specific by associating the following settings with the `ON COMMIT` clause

(select all that apply):

a. `PRESERVE ROWS`

b. `DELETE ROWS`

c. `DEFER ROWS`

d. `PURGE`

Answer: a, b

Quiz

You create external tables by using the `ORGANIZATION EXTERNAL` clause of the `CREATE TABLE` statement.

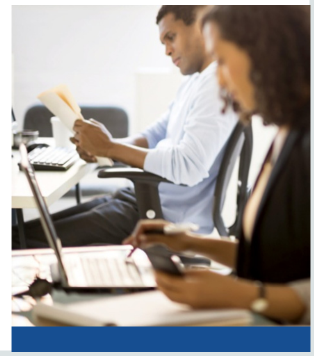
a. True
b. False

Answer: a

Summary

In this lesson, you should have learned how to:

- Manage constraints
- Create and use temporary tables
- Create and use external tables



ORACLE

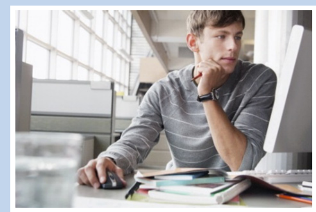
In this lesson, you learned how to perform the following tasks for schema object management:

- Alter tables to add or modify columns or constraints.
- Create and use temporary tables.
- Use the `ORGANIZATION EXTERNAL` clause of the `CREATE TABLE` statement to create an external table. An external table is a read-only table whose metadata is stored in the database but whose data is stored outside the database.
- Use external tables to query data without first loading it into the database.

Practice 20: Overview

This practice covers the following topics:

- Adding and dropping constraints
- Deferring constraints
- Creating external tables



In this practice, you use the `ALTER TABLE` command to add, drop, and defer constraints. You also create external tables.


Lesson 21: Using Advanced Subqueries
ORACLE
Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Course Roadmap

- Lesson 1: Course Overview
- Unit 1: Relational Database and SQL Overview
- Unit 2: Retrieving and Sorting Data
- Unit 3: Joins, Subqueries, and Set Operators
- Unit 4: DML and DDL
- Unit 5: Managing Relational Database
- Unit 6: Advance Queries and Database Management System
  - Lesson 21: Using Advanced Subqueries
  - Lesson 22: Manipulating Data by Using Advanced Subqueries
  - Lesson 23: Controlling User Access
  - Lesson 24: Advanced Data Manipulation
  - Lesson 25: Managing Multiple Timezones

You are here

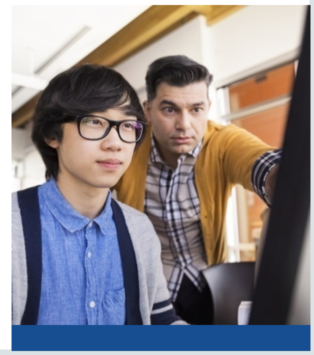
ORACLE

In Unit 6, you will be introduced to some advanced features of SQL. You will learn to write advanced subqueries. You will learn to create users and manage users. You will also learn about managing multiple timezones.

Objectives

After completing this lesson, you should be able to:

- Write a multiple-column subquery
- Use scalar subqueries in SQL
- Solve problems with correlated subqueries
- Use the `EXISTS` and `NOT EXISTS` operators
- Use the `WITH` clause



ORACLE

In this lesson, you learn how to write multiple-column subqueries and subqueries in the `FROM` clause of a `SELECT` statement. You also learn how to solve problems by using scalar, correlated subqueries and by using the `WITH` clause.

Lesson Agenda

- Retrieving data by using a subquery as a source
- Writing a multiple-column subquery
- Using scalar subqueries in SQL
- Solving problems with correlated subqueries
- Using the `EXISTS` and `NOT EXISTS` operators
- Using the `WITH` clause

Retrieving Data by Using a Subquery as a Source

```
SELECT department_name, city
FROM   departments
NATURAL JOIN (SELECT l.location_id, l.city, l.country_id
              FROM   locations l
              JOIN   countries c
              ON(l.country_id = c.country_id)
              JOIN regions
              USING(region_id)
              WHERE region_name = 'Europe');
```

	DEPARTMENT_NAME	CITY
1	Sales	Oxford
2	Human Resources	Oxford

ORACLE

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 example in the slide 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.

You can display the same output as in the example in the slide 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   locations l
JOIN   countries c
ON(l.country_id = c.country_id)
JOIN regions USING(region_id)
WHERE region_name = 'Europe';
```

2. Join the `EUROPEAN_CITIES` view with the `DEPARTMENTS` table:

```
SELECT department_name, city
FROM   departments
NATURAL JOIN european_cities;
```

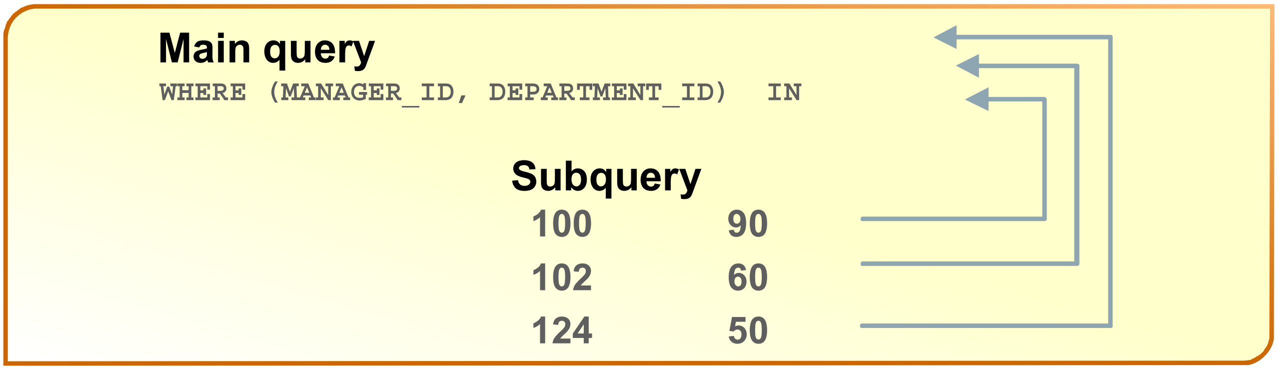
Note: You learned how to create database views in the lesson titled “Creating Views.”

Lesson Agenda

- Retrieving data by using a subquery as a source
- Writing a multiple-column subquery
- Using scalar subqueries in SQL
- Solving problems with correlated subqueries
- Using the `EXISTS` and `NOT EXISTS` operators
- Using the `WITH` clause

ORACLE

Multiple-Column Subqueries



Each row of the main query is compared to values from a multiple-row and multiple-column subquery.

ORACLE

So far, you have written single-row subqueries and multiple-row subqueries where only one column is returned by the inner `SELECT` statement and this is used to evaluate the expression in the parent `SELECT` statement. If you want to compare two or more columns, you must write a compound `WHERE` clause by using logical operators. Using multiple-column subqueries, you can combine duplicate `WHERE` conditions into a single `WHERE` clause.

Syntax

```
SELECT     column, column, ...
FROM  table
WHERE(column, column, ...) IN
             (SELECT column, column, ...
                              FROM   table
                              WHERE  condition);
```

The graphic in the slide illustrates that the values of `MANAGER_ID` and `DEPARTMENT_ID` from the main query are being compared with the `MANAGER_ID` and `DEPARTMENT_ID` values retrieved by the subquery. Because the number of columns that are being compared is more than one, the example qualifies as a multiple-column subquery.

Note: Before you run the examples in the next few slides, you need to create the `empl_demo` table and populate data into it by using the `lab_06_insert_empdata.sql` file.

Column Comparisons

Multiple-column comparisons involving subqueries can be:

- Pairwise comparisons
- Nonpairwise comparisons

Pairwise Versus Nonpairwise Comparisons

Multiple-column comparisons involving subqueries can be nonpairwise comparisons or pairwise comparisons. If you consider the example "Display the details of employees who work in the same department, and have the same manager, as 'Daniel'?," you get the correct result with the following statement:

```
SELECT first_name, last_name, manager_id, department_id
FROM empl_demo
WHERE manager_id IN (SELECT manager_id
                     FROM empl_demo
                     WHERE first_name = 'Daniel')
AND department_id IN (SELECT department_id
                      FROM empl_demo
                      WHERE first_name = 'Daniel');
```

There is only one "Daniel" in the `EMPL_DEMO` table (Daniel Faviet, who is managed by employee 108 and works in department 100). However, if the subqueries return more than one row, the result might not be correct. For example, if you run the same query but substitute "John" for "Daniel," you get an incorrect result. This is because the combination of `department_id` and `manager_id` is important. To get the correct result for this query, you need a pairwise comparison.

Pairwise Comparison Subquery

Display the details of employees who are managed by the same manager and work in the same department as employees with EMPLOYEE_ID 199 or 174.

```
SELECT employee_id, manager_id, department_id
FROM employees
WHERE (manager_id, department_id) IN
                            (SELECT manager_id, department_id
                             FROM employees
                             WHERE employee_id IN (174, 199))
AND employee_id NOT IN (174,199);
```

	EMPLOYEE_ID	MANAGER_ID	DEPARTMENT_ID
1	176	149	80

ORACLE

The example in the slide shows a pairwise comparison of columns. It compares the values in the MANAGER_ID column and the DEPARTMENT_ID column of each row in the EMPLOYEES table with the values in the MANAGER_ID column and the DEPARTMENT_ID column for employees with EMPLOYEE_ID 199 or 174.

First, the subquery to retrieve the MANAGER_ID and DEPARTMENT_ID values for employees with EMPLOYEE_ID 199 or 174 is executed. These values are compared with the MANAGER_ID column and the DEPARTMENT_ID column of each row in the EMPLOYEES table. If the values match, the row is displayed. In the output, the records of employees with EMPLOYEE_ID 199 or 174 will not be displayed. The output of the query is shown in the slide.

Nonpairwise Comparison Subquery

Display the details of employees who are managed by the same manager as employees with EMPLOYEE_ID 174 or 141 and work in the same department as employees with EMPLOYEE_ID 174 or 141.

```
SELECT employee_id, manager_id, department_id
FROM employees
WHERE manager_id IN
                  (SELECT manager_id
                   FROM employees
                   WHERE employee_id IN (174,141))
AND department_id IN
                  (SELECT department_id
                   FROM employees
                   WHERE employee_id IN (174,141))
AND employee_id NOT IN(174,141);
```

	EMPLOYEE_ID	MANAGER_ID	DEPARTMENT_ID
1	144	124	50
2	143	124	50

...

ORACLE

The example in the slide shows a nonpairwise comparison of columns. It displays the EMPLOYEE_ID, MANAGER_ID, and DEPARTMENT_ID of any employee whose manager ID matches any of the manager IDs of employees whose employee IDs are either 174 or 141 and DEPARTMENT_ID matches any of the department IDs of employees whose employee IDs are either 174 or 141.

First, the subquery to retrieve the MANAGER_ID values for employees with EMPLOYEE_ID 174 or 141 is executed. Similarly, the second subquery to retrieve the DEPARTMENT_ID values for employees with EMPLOYEE_ID 174 or 141 is executed. The retrieved values of the MANAGER_ID and DEPARTMENT_ID columns are compared with the MANAGER_ID and DEPARTMENT_ID columns for each row in the EMPLOYEES table. If the MANAGER_ID column of the row in the EMPLOYEES table matches with any of the values of the MANAGER_ID retrieved by the inner subquery and if the DEPARTMENT_ID column of the row in the EMPLOYEES table matches with any of the values of the DEPARTMENT_ID retrieved by the second subquery, the record is displayed.

Lesson Agenda

- Retrieving data by using a subquery as a source
- Writing a multiple-column subquery
- Using scalar subqueries in SQL
- Solving problems with correlated subqueries
- Using the `EXISTS` and `NOT EXISTS` operators
- Using the `WITH` clause

ORACLE

Scalar Subquery Expressions

- A scalar subquery expression is a subquery that returns exactly one column value from one row.
- Scalar subqueries can be used in:
  - The condition and expression part of `DECODE` and `CASE`
  - All clauses of `SELECT` except `GROUP BY`
  - The `SET` clause and `WHERE` clause of an `UPDATE` statement

ORACLE

A subquery that returns exactly one column value from one row is also referred to as a scalar subquery. Multiple-column subqueries that are written to compare two or more columns, using a compound `WHERE` clause and logical operators, do not qualify as scalar subqueries.

The value of a scalar subquery expression is the value of the select list item of the subquery. If the subquery returns 0 rows, the value of the scalar subquery expression is `NULL`. If the subquery returns more than one row, the Oracle Server returns an error. The Oracle Server has always supported the usage of a scalar subquery in a `SELECT` statement. You can use scalar subqueries in:

- The condition and expression part of `DECODE` and `CASE`
- All clauses of `SELECT` except `GROUP BY`
- The `SET` clause and `WHERE` clause of an `UPDATE` statement

However, scalar subqueries are not valid expressions in the following places:

- In the `RETURNING` clause of data manipulation language (DML) statements
- As the basis of a function-based index
- In `GROUP BY` clauses and `CHECK` constraints
- In `CONNECT BY` clauses
- In statements that are unrelated to queries, such as `CREATE PROFILE`

Scalar Subqueries: Examples

- Scalar subqueries in CASE expressions:

```
SELECT employee_id, last_name,
       (CASE
        WHEN department_id =                    20
                 (SELECT department_id
                  FROM departments
                  WHERE location_id = 1800)
         THEN 'Canada' ELSE 'USA' END) location
FROM   employees;
```

- Scalar subqueries in a SELECT statement:

```
select department_id, department_name,
       (select count(*)
        from   employees e
        where  e.department_id = d.department_id) as emp_count
from   departments d;
```

ORACLE

The first example in the slide demonstrates that scalar subqueries can be used in CASE expressions. The inner query returns the value 20, which is the department ID of the department whose location ID is 1800. The CASE expression in the outer query uses the result of the inner query to display the employee ID, last names, and a value of Canada or USA, depending on whether the department ID of the record retrieved by the outer query is 20.

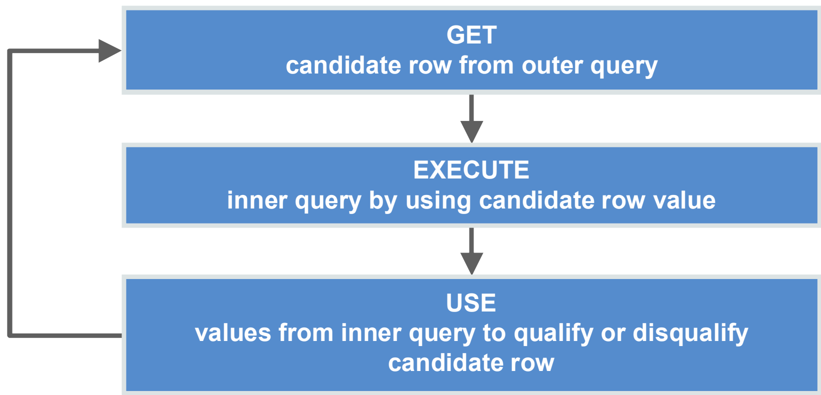
The second example in the slide demonstrates that scalar subqueries can be used in SELECT statements.

Lesson Agenda

- Retrieving data by using a subquery as a source
- Writing a multiple-column subquery
- Using scalar subqueries in SQL
- Solving problems with correlated subqueries
- Using the `EXISTS` and `NOT EXISTS` operators
- Using the `WITH` clause

Correlated Subqueries

Correlated subqueries are used for row-by-row processing. Each subquery is executed once for every row of the outer query.



ORACLE

The Oracle Server performs a correlated subquery when the subquery references a column from a table referred to in the parent statement. A correlated subquery is evaluated once for each row processed by the parent statement. The parent statement can be a `SELECT`, `UPDATE`, or `DELETE` statement.

Nested Subqueries Versus Correlated Subqueries

With a normal nested subquery, the inner `SELECT` query runs first and executes once, returning values to be used by the main query. A correlated subquery, however, executes once for each candidate row considered by the outer query. That is, the inner query is driven by the outer query.

Nested Subquery Execution

- The inner query executes first and finds a value.
- The outer query executes once, using the value from the inner query.

Correlated Subquery Execution

- Get a candidate row (fetched by the outer query).
- Execute the inner query by using the value of the candidate row.
- Use the values resulting from the inner query to qualify or disqualify the candidate.
- Repeat until no candidate row remains.

Correlated Subqueries

The subquery references a column from a table in the parent query.

```
SELECT column1, column2, ...
FROM   table1          Outer_table
WHERE  column1 operator
                      (SELECT  column1, column2
                 FROM    table2
                 WHERE   expr1 =  Outer_table
                                      .expr2);
```


A correlated subquery is one way of reading every row in a table and comparing values in each row against related data. It is used whenever a subquery must return a different result or set of results for each candidate row considered by the main query. That is, you use a correlated subquery to answer a multipart question whose answer depends on the value in each row processed by the parent statement.

The Oracle Server performs a correlated subquery when the subquery references a column from a table in the parent query.

Note: You can use the ANY and ALL operators in a correlated subquery.

Using Correlated Subqueries: Example 1

Find all employees who earn more than the average salary in their department.

```
SELECT last_name, salary, department_id
FROM   employees outer_table
WHERE  salary >
       (SELECT AVG(salary)
           FROM   employees inner_table
           WHERE  inner_table.department_id =
           outer_table.department_id);
```

	LAST_NAME	SALARY	DEPARTMENT_ID
1	Hartstein	13000	20
2	King	24000	90
3	Hunold	12008	60
4	Mourgos	5800	50
5	Zlotkey	10500	80
6	Abel	11000	80

Each time a row from the outer query is processed, the inner query is evaluated.

ORACLE®

The example in the slide finds employees who earn more than the average salary in their department. In this case, the correlated subquery specifically computes the average salary for each department.

Because both the outer query and inner query use the `EMPLOYEES` table in the `FROM` clause, an alias is given to `EMPLOYEES` in the outer `SELECT` statement for clarity. The alias makes the entire `SELECT` statement more readable. Without the alias, the query would not work properly because the inner statement would not be able to distinguish the inner table column from the outer table column.

The correlated subquery performs the following steps for each row of the `EMPLOYEES` table:

1. The `department_id` of the row is determined.
2. The `department_id` is then used to evaluate the parent query.
3. If the salary in that row is greater than the average salary of the departments of that row, the row is returned.

The subquery is evaluated once for each row of the `EMPLOYEES` table.

Using Correlated Subqueries: Example 2

Display details of the highest earning employee in each department.

```
SELECT department_id, employee_id, salary
FROM EMPLOYEES e
WHERE 1 =
    (SELECT  COUNT(DISTINCT salary)
     FROM EMPLOYEES
     WHERE e.department_id = department_id
     AND e.salary <= salary)
```

	DEPARTMENT_ID	EMPLOYEE_ID	SALARY
1	90	100	24000
2	60	103	9000
3	100	108	12008
4	30	114	11000

....

ORACLE

The example in the slide displays the details of the highest earning employees in each department. The Oracle Server evaluates a correlated subquery as follows:

1. Select a row from the table specified in the outer query. This will be the current candidate row.
2. Store the value of the column referenced in the subquery from this candidate row. (In the example in the slide, the column referenced in the subquery is `e.salary`.)
3. Perform the subquery with its condition referencing the value from the outer query's candidate row. (In the example in the slide, the `COUNT(DISTINCT salary)` group function is evaluated based on the value of the `E.SALARY` column obtained in step 2.)
4. Evaluate the `WHERE` clause of the outer query on the basis of the results of the subquery performed in step 3. This determines whether the candidate row is selected for output. (In the example, the number of times an employee has changed jobs, evaluated by the subquery, is compared with 2 in the `WHERE` clause of the outer query. If the condition is satisfied, that employee record is displayed.)
5. Repeat the procedure for the next candidate row of the table, and so on, until all the rows in the table have been processed.

The correlation is established by using an element from the outer query in the subquery. In this example, you compare `EMPLOYEE_ID` from the table in the subquery with `EMPLOYEE_ID` from the table in the outer query.

Lesson Agenda

- Retrieving data by using a subquery as a source
- Writing a multiple-column subquery
- Using scalar subqueries in SQL
- Solving problems with correlated subqueries
- Using the `EXISTS` and `NOT EXISTS` operators
- Using the `WITH` clause

ORACLE

Using the EXISTS Operator

- The `EXISTS` operator tests for existence of rows in the results set of the subquery.
- If a subquery row value is found:
  - The search does not continue in the inner query
  - The condition is flagged `TRUE`
- If a subquery row value is not found:
  - The condition is flagged `FALSE`
  - The search continues in the inner query

ORACLE

With nesting `SELECT` statements, all logical operators are valid. In addition, you can use the `EXISTS` operator. This operator is frequently used with correlated subqueries to test whether a value retrieved by the outer query exists in the results set of the values retrieved by the inner query. If the subquery returns at least one row, the operator returns `TRUE`. If the value does not exist, it returns `FALSE`. Accordingly, `NOT EXISTS` tests whether a value retrieved by the outer query is not a part of the results set of the values retrieved by the inner query.

Using the EXISTS Operator

```
SELECT employee_id, last_name, job_id, department_id
FROM   employees outer
WHERE  EXISTS ( SELECT NULL
                FROM   employees
                WHERE  manager_id =
                       outer.employee_id);
```

	EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
1	100	King	AD_PRES	90
2	101	Kochhar	AD_VP	90
3	102	De Haan	AD_VP	90
4	103	Hunold	IT_PROG	60
5	124	Mourgos	ST_MAN	50
6	149	Zlotkey	SA_MAN	80
7	201	Hartstein	MK_MAN	20
8	205	Higgins	AC_MGR	110

ORACLE

The EXISTS operator ensures that the search in the inner query does not continue when at least one match is found for the manager and employee number by the condition:

```
WHERE manager_id = outer.employee_id
```

Note that the inner SELECT query does not need to return a specific value; so a constant can be selected.

Finding All Departments That Do Not Have Any Employees

```
SELECT department_id, department_name
FROM departments d
WHERE NOT EXISTS (SELECT NULL
                  FROM   employees
                  WHERE  department_id = d.department_id);
```

	DEPARTMENT_ID	DEPARTMENT_NAME
1	280	Support
2	190	Contracting

All Rows Fetched: 2

ORACLE

Using the `NOT EXISTS` Operator

Alternative Solution

A `NOT IN` construct can be used as an alternative for a `NOT EXISTS` operator, as shown in the following example:

```
SELECT department_id, department_name
FROM   departments
WHERE  department_id NOT IN (SELECT department_id
                             FROM   employees);
```

However, `NOT IN` evaluates to `FALSE` if any member of the set is a `NULL` value. Therefore, your query will not return any rows even if there are rows in the `departments` table that satisfy the `WHERE` condition.

Lesson Agenda

- Retrieving data by using a subquery as a source
- Writing a multiple-column subquery
- Using scalar subqueries in SQL
- Solving problems with correlated subqueries
- Using the `EXISTS` and `NOT EXISTS` operators
- Using the `WITH` clause

WITH Clause

- Using the WITH clause, you can use the same query block in a SELECT statement when it occurs more than once within a complex query.
- The WITH clause retrieves the results of a query block and stores it in the user's temporary tablespace.
- The WITH clause may improve performance.

Using the WITH clause, you can define a query block before using it in a query. 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. This is particularly useful when a query has many references to the same query block and there are joins and aggregations.

Using the WITH clause, you can reuse the same query when it is costly to evaluate the query block and it occurs more than once within a complex query. Using the WITH clause, the Oracle Server retrieves the results of a query block and stores it in the user's temporary tablespace. This can improve performance.

WITH Clause Benefits

- Makes the query easy to read
- Evaluates a clause only once, even if it appears multiple times in the query
- In most cases, may improve performance for large queries

WITH Clause: Example

```
WITH CNT_DEPT AS
(
SELECT department_id,
 COUNT(1) NUM_EMP
FROM EMPLOYEES
GROUP BY department_id
)
SELECT employee_id,
 SALARY/NUM_EMP
FROM EMPLOYEES E
JOIN CNT_DEPT C
ON (e.department_id = c.department_id);
```

	EMPLOYEE_ID	SALARY/NUM_EMP
1	101	5666.666666666666666666666666666666666667
2	176	2866.666666666666666666666666666666666667
3	107	1400
4	149	3500
5	200	4400
6	205	6004
7	206	4150

...

The SQL code in the slide is an example of a situation in which you can improve performance and write SQL more simply by using the WITH clause. The query creates the query name as CNT_DEPT, and then uses it in the body of the main query. Here, you perform a math operation by dividing the salary of an employee with the total number of employees in each department. Internally, the WITH clause is resolved either as an inline view or a temporary table. The optimizer chooses an appropriate resolution depending on the cost or benefit of temporarily storing the results of the WITH clause.

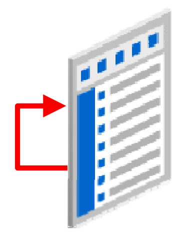
WITH Clause Usage Notes

- It is used only with SELECT statements.
- A query name is visible to all WITH element query blocks (including their subquery blocks) that are defined after it and the main query block itself (including its subquery blocks).
- When the query name is the same as an existing table name, the parser searches from the inside out, and the query block name takes precedence over the table name.
- The WITH clause can hold more than one query. Each query is then separated by a comma.

Recursive `WITH` Clause

The Recursive `WITH` clause:

- Enables formulation of recursive queries
- Creates a query with a name, called the Recursive `WITH` element name
- Contains two types of query block members: an anchor and a recursive
- Is ANSI-compatible



The `WITH` clause has been extended to enable formulation of recursive queries.

Recursive `WITH` defines a recursive query with a name, the Recursive `WITH` element name. The Recursive `WITH` element definition must contain at least two query blocks: an anchor member and a recursive member. There can be multiple anchor members, but there can be only a single recursive member. The anchor member must appear before the recursive member, and it cannot reference *query_name*. The anchor member can be composed of one or more query blocks combined by the set operators, for example, `UNION ALL`, `UNION`, `INTERSECT`, or `MINUS`. The recursive member must follow the anchor member and must reference *query_name* exactly once. You must combine the recursive member with the anchor member by using the `UNION ALL` set operator.

The Recursive `WITH` clause complies with the American National Standards Institute (ANSI) standard.

Recursive `WITH` can be used to query hierarchical data such as organization charts.

Recursive `WITH` Clause: Example

FLIGHTS Table

	SOURCE	DESTIN	FLIGHT_TIME
1	San Jose	Los Angeles	1.3
2	New York	Boston	1.1
3	Los Angeles	New York	5.8

1

```
WITH Reachable_From (Source, Destin, TotalFlightTime) AS
(
    SELECT Source, Destin, Flight_time
    FROM Flights
  UNION ALL
     SELECT incoming.Source, outgoing.Destin,
            incoming.TotalFlightTime+outgoing.Flight_time
     FROM Reachable_From incoming, Flights outgoing
     WHERE incoming.Destin = outgoing.Source
)
SELECT Source, Destin, TotalFlightTime
FROM Reachable_From;
```

2

3

	SOURCE	DESTIN	TOTALFLIGHTTIME
1	San Jose	Los Angeles	1.3
2	New York	Boston	1.1
3	Los Angeles	New York	5.8
4	Los Angeles	Boston	6.9
5	San Jose	New York	7.1
6	San Jose	Boston	8.2

ORACLE

Example 1 in the slide displays records from a `FLIGHTS` table that describes flights between two cities.

Using the query in example 2, you query the `FLIGHTS` table to display the total flight time between any source and destination. The `WITH` clause in the query, which is named `Reachable From`, has a `UNION ALL` query with two branches. The first branch is the *anchor* branch, which selects all the rows from the `Flights` table. The second branch is the recursive branch. It joins the contents of `Reachable From` to the `Flights` table to find other cities that can be reached, and adds these to the content of `Reachable From`. The operation finishes when no more rows are found by the recursive branch.

Example 3 displays the result of the query that selects everything from the `WITH` clause element `Reachable From`.

For details, see:

- *Oracle Database SQL Language Reference 12c Release 1.0*
- *Oracle Database Data Warehousing Guide 12c Release 1.0*

Quiz

With a correlated subquery, the inner `SELECT` statement drives the outer `SELECT` statement.

a. True

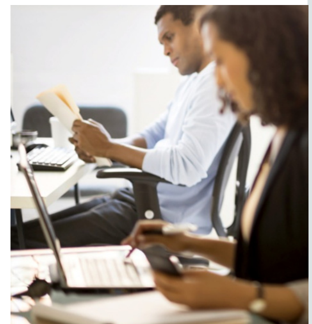
b. False

Answer: b

Summary

In this lesson, you should have learned how to:

- Write a multiple-column subquery
- Use scalar subqueries in SQL
- Solve problems with correlated subqueries
- Use the EXISTS and NOT EXISTS operators
- Use the WITH clause



ORACLE

You can use multiple-column subqueries to combine multiple WHERE conditions in a single WHERE clause. Column comparisons in a multiple-column subquery can be pairwise comparisons or nonpairwise comparisons.

You can use a subquery to define a table to be operated on by a containing query.

Scalar subqueries can be used in:

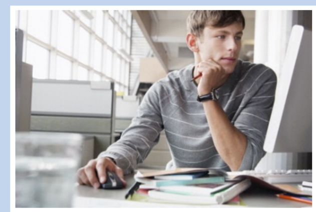
- The condition and expression part of DECODE and CASE
- All clauses of SELECT except GROUP BY
- The SET clause and WHERE clause of the UPDATE statement

The Oracle Server performs a correlated subquery when the subquery references a column from a table referred to in the parent statement. A correlated subquery is evaluated once for each row processed by the parent statement. The parent statement can be a SELECT statement. Using the WITH clause, you can reuse the same query when it is costly to re-evaluate the query block and it occurs more than once within a complex query.

Practice 21: Overview

This practice covers the following topics:

- Creating multiple-column subqueries
- Writing correlated subqueries
- Using the EXISTS operator
- Using scalar subqueries
- Using the WITH clause



In this practice, you write multiple-column subqueries, and correlated and scalar subqueries. You also solve problems by writing the WITH clause.


Lesson 22: Manipulating Data by Using Advanced Subqueries
ORACLE
Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Course Roadmap

Lesson 1: Course Overview

Unit 1: Relational Database and SQL Overview

Unit 2: Retrieving and Sorting Data

Unit 3: Joins, Subqueries, and Set Operators

Unit 4: DML and DDL

Unit 5: Managing Relational Database

Unit 6: Advance Queries and Database Management System

You are here

Lesson 21: Using Advanced Subqueries

Lesson 22: Manipulating Data by Using Advanced Subqueries

Lesson 23: Controlling User Access

Lesson 24: Advanced Data Manipulation

Lesson 25: Managing Multiple Timezones

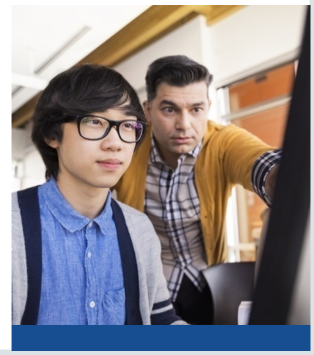
ORACLE

In Unit 6, you will be introduced to some advanced features of SQL. You will learn to write advanced subqueries. You will learn to create users and manage users. You will also learn about managing multiple timezomes.

Objectives

After completing this lesson, you should be able to:

- Use advanced subqueries to manipulate data
- Insert values by using a subquery as a target
- Use the `WITH CHECK OPTION` keyword on DML statements
- Use correlated subqueries to update and delete rows



In this lesson, you learn how to manipulate data in the Oracle database by using subqueries. You also learn how to solve problems by using correlated subqueries.

Lesson Agenda

- Using subqueries to manipulate data
- Inserting values by using a subquery as a target
- Using the `WITH CHECK OPTION` keyword on DML statements
- Using correlated subqueries to update and delete rows

ORACLE

Using Subqueries to Manipulate Data

You can use subqueries in data manipulation language (DML) statements to:

- Retrieve data by 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

Subqueries can be used to retrieve data from a table that you can use as input to an `INSERT` into a different table. Thus, you can easily copy large volumes of data from one table to another with a single `SELECT` statement. Similarly, you can use subqueries to perform 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 lesson titled “Managing Tables Using DML Statements.”

Lesson Agenda

- Using subqueries to manipulate data
- Inserting values by using a subquery as a target
- Using the `WITH CHECK OPTION` keyword on DML statements
- Using correlated subqueries to update and delete rows

Inserting by 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.
```

ORACLE

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 use of subqueries helps you to avoid having to create a view only for performing an `INSERT`.

The example in the slide uses a subquery in 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');
```

For the example in the slide, the `loc` table is created by running the following statement:

```
CREATE TABLE loc AS SELECT * FROM locations;
```

Inserting by Using a Subquery as a Target

Verify the results.

```
SELECT location_id, city, country_id
FROM   loc;
```

20	2900	Geneva	CH
21	3000	Bern	CH
22	3100	Utrecht	NL
23	3200	Mexico City	MX
24	3300	Cardiff	UK

ORACLE

The example in the slide shows that the insert via the inline view created a new record in the base table `LOC`.

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';
```

Lesson Agenda

- Using subqueries to manipulate data
- Inserting values by using a subquery as a target
- Using the `WITH CHECK OPTION` keyword on DML statements
- Using correlated subqueries to update and delete rows

ORACLE

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 report:
SQL Error: ORA-01402: view `WITH CHECK OPTION` where-clause violation
01402. 00000 - "view `WITH CHECK OPTION` where-clause violation"
***Cause:**
***Action:**

Specify the `WITH CHECK OPTION` keyword to indicate that if a subquery is used in place of a table in an `INSERT`, `UPDATE`, or `DELETE` statement, changes that will produce rows that are not included in the subquery will not be permitted to that table.

The example in the slide shows how to use an inline view with `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');
```

The use 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 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.

Lesson Agenda

- Using subqueries to manipulate data
- Inserting values by using a subquery as a target
- Using the `WITH CHECK OPTION` keyword on DML statements
- Using correlated subqueries to update and delete rows

Correlated UPDATE

Use a correlated subquery to update rows in one table based on rows from another table.

```
UPDATE table1 alias1
SET    column = (SELECT expression
                 FROM   table2 alias2
                 WHERE  alias1.column =
                        alias2.column);
```


In the case of an UPDATE statement, you can use a correlated subquery to update rows in one table based on rows from another table.

Using Correlated UPDATE

- Denormalize the EMPL6 table by adding a column to store the department name.
- Populate the table by using a correlated update.

```
ALTER TABLE empl6
ADD(department_name VARCHAR2(25));
```

```
UPDATE empl6 e
SET    department_name =
              (SELECT department_name
               FROM   departments d
               WHERE  e.department_id = d.department_id);
```

ORACLE

The example in the slide denormalizes the EMPL6 table by adding a column to store the department name, and then populates the table by using a correlated update.

Another example for a correlated update is as follows.

Problem Statement

The REWARDS table has a list of employees who have exceeded expectations in their performance. Use a correlated subquery to update the rows in the EMPL6 table based on the rows from the REWARDS table:

```
UPDATE empl6
SET    salary = (SELECT empl6.salary + rewards.pay_raise
                 FROM   rewards
                 WHERE  employee_id  =
                        empl6.employee_id
                 AND  payraise_date =
                      (SELECT MAX(payraise_date)
                       FROM   rewards
                       WHERE  employee_id = empl6.employee_id))
WHERE  empl6.employee_id
IN     (SELECT employee_id FROM rewards);
```

This example uses the REWARDS table. The REWARDS table has the following columns: EMPLOYEE_ID, PAY_RAISE, and PAYRAISE_DATE. Every time an employee gets a pay raise, a record with details such as the employee ID, the amount of the pay raise, and the date of receipt of the pay raise is inserted into the REWARDS table. The REWARDS table can contain more than one record for an employee. The PAYRAISE _DATE column is used to identify the most recent pay raise received by an employee.

In the example, the SALARY column in the EMPL6 table is updated to reflect the latest pay raise received by an employee. This is done by adding the current salary of the employee with the corresponding pay raise from the REWARDS table.

Correlated DELETE

Use a correlated subquery to delete rows in one table based on rows from another table.

```
DELETE FROM table1 alias1
WHERE  column operator
               (SELECT expression
                FROM   table2 alias2
                WHERE  alias1.column = alias2.column);
```

ORACLE

In the case of a DELETE statement, you can use a correlated subquery to delete only those rows that also exist in another table. If you decide that you will maintain only the last four job history records in the JOB_HISTORY table, when an employee transfers to a fifth job, you delete the oldest JOB_HISTORY row by looking up the JOB_HISTORY table for MIN(START_DATE) for the employee. The following code illustrates how the preceding operation can be performed by using a correlated DELETE:

```
DELETE FROM job_history JH
WHERE employee_id =
      (SELECT employee_id
       FROM employees E
       WHERE JH.employee_id = E.employee_id
       AND START_DATE =
             (SELECT MIN(start_date)
              FROM job_history JH
              WHERE JH.employee_id = E.employee_id)
              AND 5 >  (SELECT COUNT(*)
                        FROM job_history JH
                        WHERE JH.employee_id = E.employee_id
                        GROUP BY EMPLOYEE_ID
                        HAVING COUNT(*) >= 4));
```

Using Correlated DELETE

Use a correlated subquery to delete only those rows from the EMPL6 table that also exist in the EMP_HISTORY table.

```
DELETE FROM empl6 E
WHERE employee_id =
           (SELECT employee_id
            FROM   emp_history
            WHERE  employee_id = E.employee_id);
```

ORACLE

Example

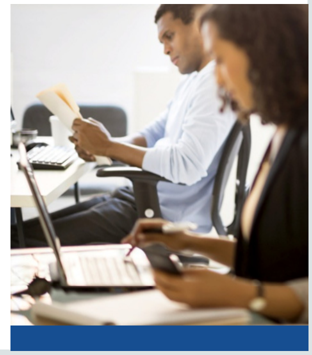
Two tables are used in this example. They are:

- The EMPL6 table, which provides details of all current employees
- The EMP_HISTORY table, which provides details of previous employees

EMP_HISTORY contains data about previous employees, so it would be erroneous if the same employee's record existed in both the EMPL6 and EMP_HISTORY tables. You can delete such erroneous records by using the correlated subquery shown in the slide.

Summary

In this lesson, you should have learned how to:
- Manipulate data by using subqueries
- Insert values by using a subquery as a target
- Use the `WITH CHECK OPTION` keyword on DML statements
- Use correlated subqueries with `UPDATE` and `DELETE` statements



In this lesson, you should have learned how to manipulate data in the Oracle database by using subqueries. You learn how to use the `WITH CHECK OPTION` keyword on DML statements and use correlated subqueries with `UPDATE` and `DELETE` statements.

Practice 22: Overview

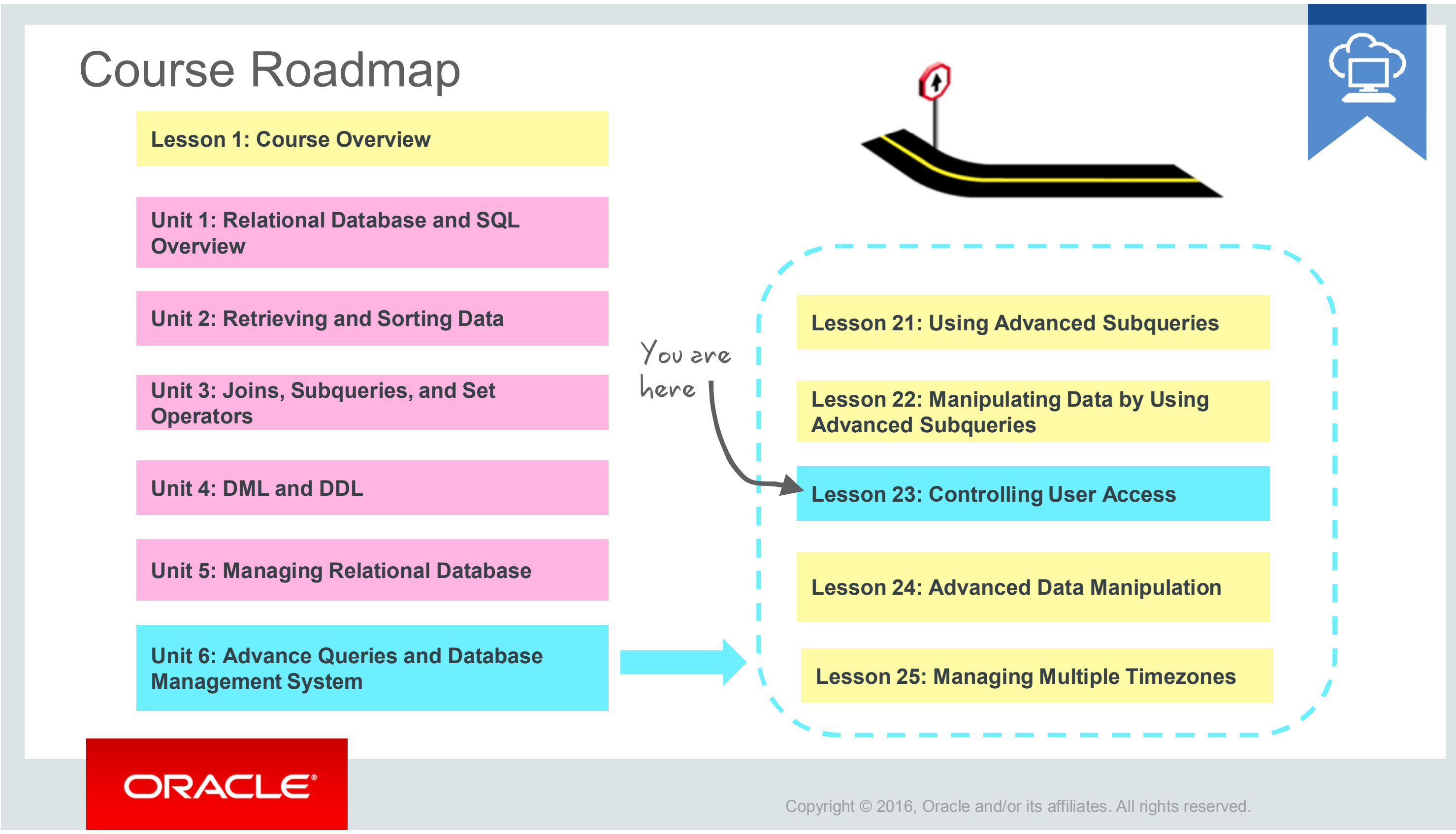
This practice covers the following topics:

- Using subqueries to manipulate data
- Inserting values by using a subquery as a target
- Using the `WITH CHECK OPTION` keyword on DML statements
- Using correlated subqueries to update and delete rows

ORACLE

In this practice, you learn the concepts of manipulating data by using subqueries, `WITH CHECK OPTION`, and correlated subqueries to `UPDATE` and `DELETE` rows.


Lesson 23: Controlling User Access
ORACLE
Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

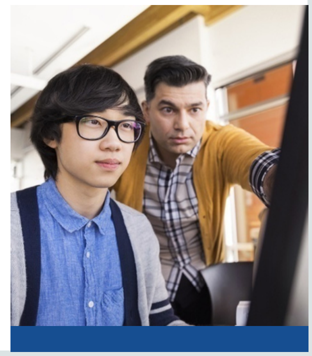


In Unit 6, you will be introduced to some advanced features of SQL. You will learn to write advanced subqueries. You will learn to create users and manage users. You will also learn about managing multiple timezomes.

Objectives

After completing this lesson, you should be able to:

- Differentiate system privileges from object privileges
- Grant privileges on tables
- Grant roles
- Distinguish between privileges and roles
- Describe Oracle Cloud service administration roles



ORACLE

In this lesson, you learn how to control database access to specific objects and add new users with different levels of access privileges.

Lesson Agenda

- System privileges
- Creating a role
- Object privileges
- Revoking object privileges
- Oracle Cloud Service administration roles

Database Security

- Database security can be classified as:

System Security	Access and use of the database at the system level such as username/password security, disk space allocation, and system operations
Data Security	Access and use of the database objects and the allowed actions

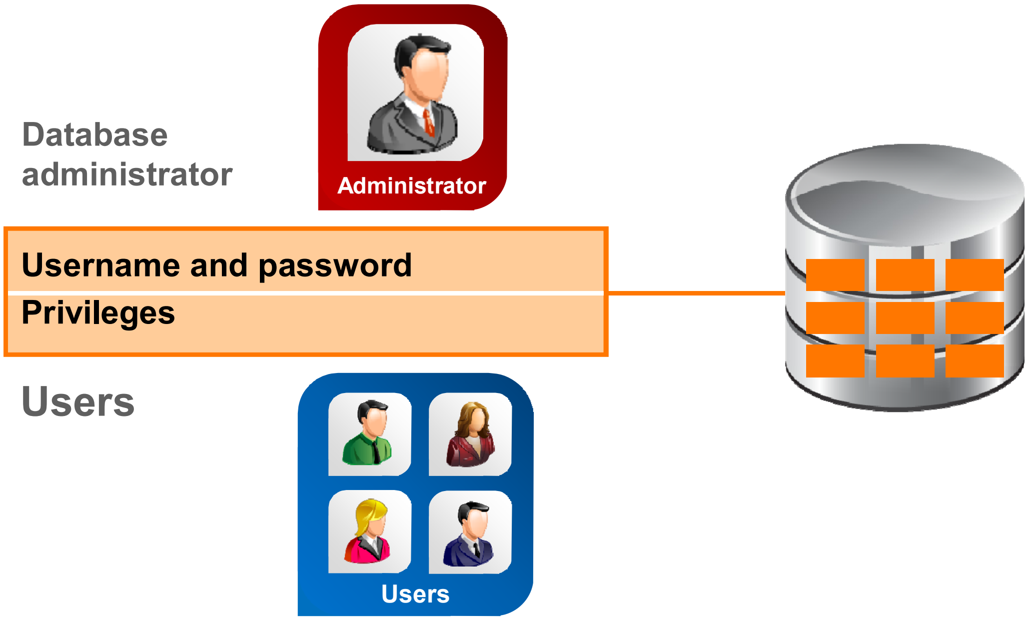
ORACLE

You must provide a basic level of database security. There must be some rules to control user access to data and to limit the kinds of SQL statements that users can execute. When creating a user, you grant those abilities (in the form of privileges) to enable the user to connect to the database, to run queries and make updates, to create schema objects, and more.

Database security can be classified into two categories: system security and data security. System security covers access and use of the database at the system level, such as the username and password, the disk space allocated to users, and the system operations that users can perform. Data security covers access and use of the database objects and the actions that specific users can perform on the objects.

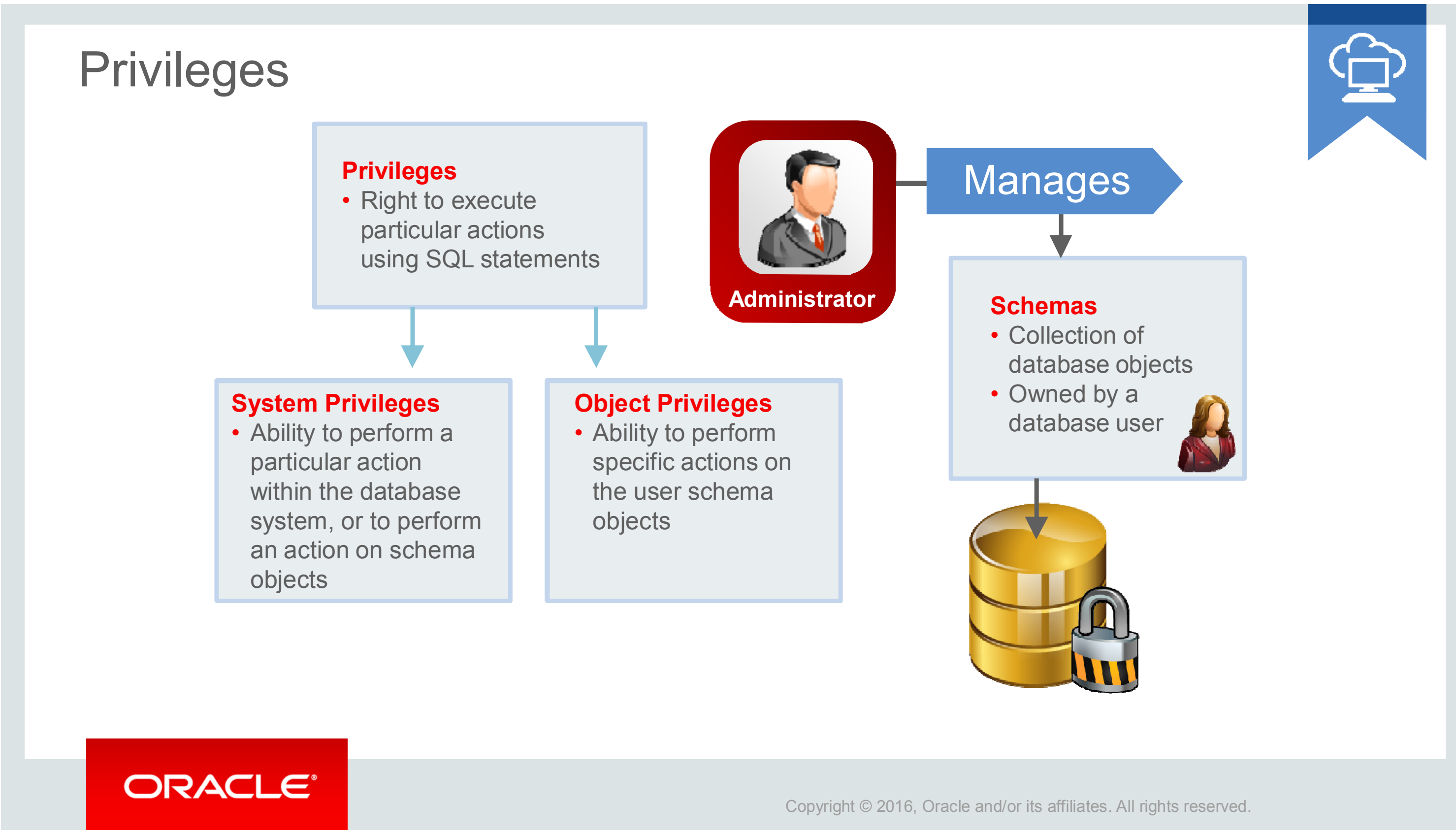
For more information, see the *Oracle Database 2 Day DBA* reference manual for Oracle Database12*c*.

Controlling User Access



In a multiple-user environment, you want to maintain security of database access and use. With Oracle Server database security, you can do the following:

- Control database access.
- Give access to specific objects in the database.
- Confirm the given and received privileges with the Oracle data dictionary.



A privilege is the right to execute particular SQL statements. The database administrator (DBA) is a high-level user with the ability to create users and grant users access to the database and its objects. Users require *system privileges* to gain access to the database and *object privileges* to manipulate the content of the objects in the database. Users can also be given the privilege to grant additional privileges to other users or to *roles*, which are named groups of related privileges.

Schemas

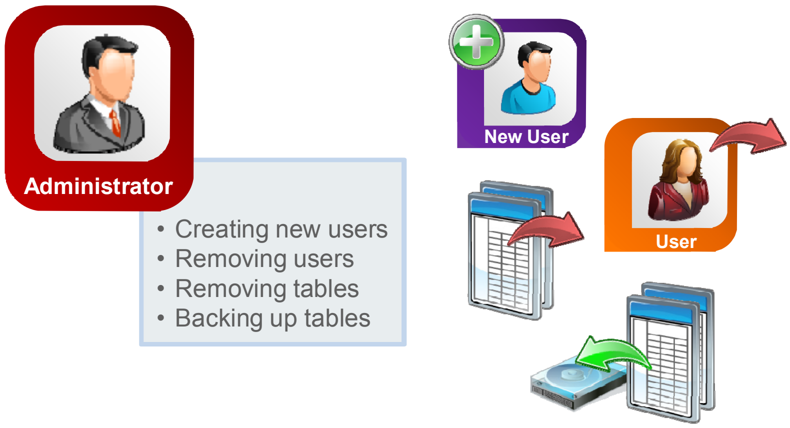
A *schema* is a collection of objects such as tables, views, and sequences. The schema is owned by a database user and has the same name as that user.

A system privilege is the right to perform a particular action within the database system, or to perform an action on any schema objects of a particular type. An object privilege provides the user the ability to perform a particular action on a specific schema object.

For more information, see the *Oracle Database 2 Day DBA* reference manual for Oracle Database12*c*.

System Privileges

- More than 200 privileges are available.
- The DBA has high-level system privileges.



More than 200 distinct system privileges are available for users and roles. Typically, system privileges are provided by the DBA.

The table `SYSTEM_PRIVILEGE_MAP` contains all the system privileges available, based on the version release. This table is also used to map privilege type numbers to type names.

Typical DBA Privileges

System Privilege	Operations Authorized
`CREATE USER`	Grantee can create other Oracle users.
`DROP USER`	Grantee can drop another user.
`DROP ANY TABLE`	Grantee can drop a table in any schema.
`BACKUP ANY TABLE`	Grantee can back up any table in any schema with the export utility.
`SELECT ANY TABLE`	Grantee can query tables, views, or materialized views in any schema.
`CREATE ANY TABLE`	Grantee can create tables in any schema.

ORACLE

The DBA (Database Administrator) is a predefined role in Oracle Database. Any user who is assigned the DBA role has almost all the rights, system and object, on the database system.

The DBA is a user who performs most administrative functions, including creating users and granting privileges; creating and granting roles; and creating, modifying, and deleting schema objects. This role grants all system privileges.

Creating Users

The DBA creates users with the `CREATE USER` statement.

```
CREATE USER user
IDENTIFIED BY   password;
```

Example:

```
CREATE USER  demo
IDENTIFIED BY demo;
```


The DBA creates a user by executing the `CREATE USER` statement. The user does not have any privileges at this point. The DBA can then grant privileges to that user. These privileges determine what the user can do at the database level.

The slide gives the abridged syntax for creating a user.

In the syntax:

user	Is the name of the user to be created
Password	Specifies that the user must log in with this password

For more information, see the *Oracle Database SQL Language Reference* for Oracle Database12*c*.

Note: Starting with Oracle Database 11*g*, passwords are case-sensitive.

User System Privileges

- After a user is created, the DBA can grant specific system privileges to that user.

```
GRANT privilege [, privilege...]
TO user [, user| role, PUBLIC...];
```

Is the name of the user, the name of the role, or PUBLIC (which designates that every user is granted the privilege)

- An application developer, for example, requires the following system privileges:

CREATE SESSION	Connect to the database.
CREATE TABLE	Create tables in the user's schema.
CREATE SEQUENCE	Create a sequence in the user's schema.
CREATE VIEW	Create a view in the user's schema.
CREATE PROCEDURE	Create a stored procedure, function, or package in the user's schema.

ORACLE

Typical User Privileges

After the DBA creates a user, the DBA can assign privileges to that user. Without these basic system privileges, the new user can barely perform any database tasks.

In the syntax:

privilege	Is the system privilege to be granted
user|role|PUBLIC	Is the name of the user, the name of the role, or PUBLIC (which designates that every user is granted the privilege)

Note: Current system privileges can be found in the SESSION_PRIVS dictionary view. Data dictionary is a collection of tables and views created and maintained by the Oracle Server. These data dictionary views contain information about the database.

Granting System Privileges

The DBA can grant specific system privileges to a user.

```
GRANT   create session, create table,
        create sequence, create view
TO      demo;
```

```
GRANT succeeded.
```

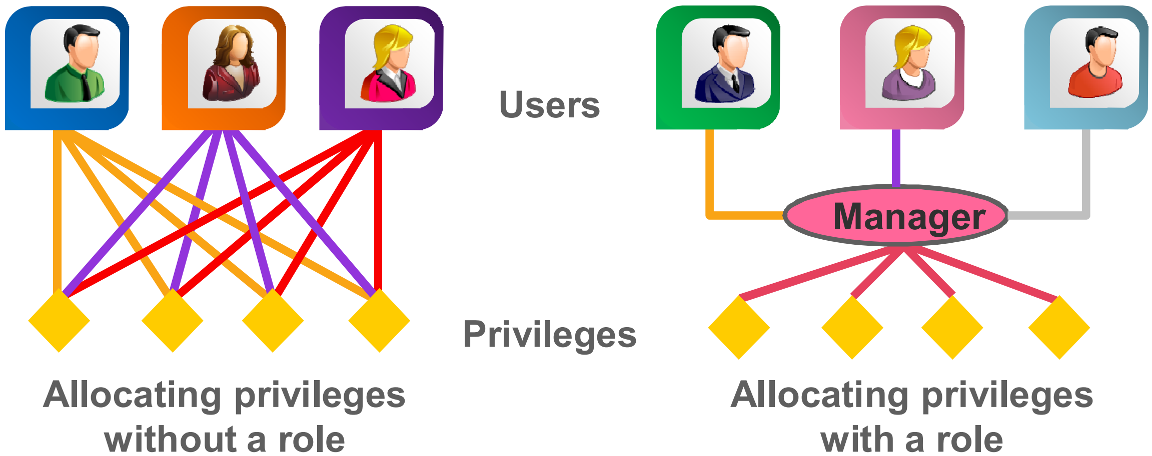

The DBA uses the GRANT statement to allocate system privileges to the user. After the user has been granted the privileges, the user can immediately use those privileges.

In the example in the slide, the demo user has been assigned the privileges to create sessions, tables, sequences, and views.

Lesson Agenda

- System privileges
- Creating a role
- Object privileges
- Revoking object privileges
- Oracle Cloud Service administration roles

What Is a Role?



A role is a named group of related privileges that can be granted to a user. This method makes it easier to revoke and maintain privileges.

A user can have access to several roles, and several users can be assigned the same role. Roles are typically created for a database application.

Creating and Assigning a Role

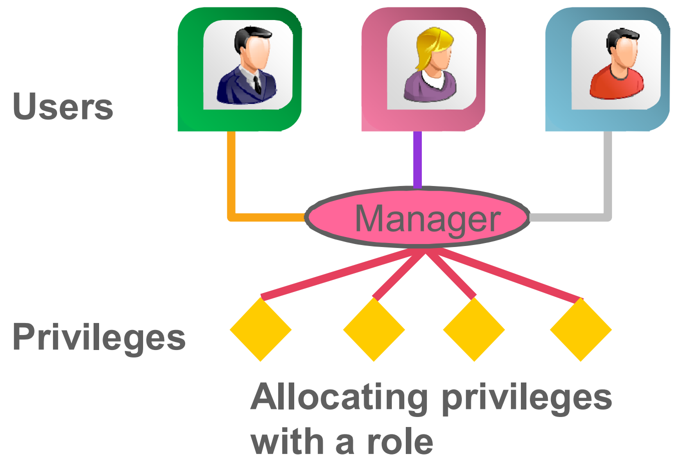
First, the DBA must create the role. Then the DBA can assign privileges to the role and assign the role to users.

After the role is created, the DBA can use the `GRANT` statement to assign the role to users as well as assign privileges to the role. A role is not a schema object; therefore, any user can add privileges to a role.

Role: Syntax

`CONNECT`, `RESOURCE`, and `DBA` are predefined Oracle Database Roles.

```
CREATE ROLE role;
```



Syntax

```
CREATE  ROLE role;
```

In the syntax:

role Is the name of the role to be created

Oracle provides the following predefined roles:

- **CONNECT**: Required to connect to the database. You should grant this role to any user that needs to access the database.
- **RESOURCE**: Required to create, modify, and delete schema objects in the user's schema. You should grant this role to users who create schema objects. This role grants a subset of the create object system privileges.
- **DBA**: Required to perform most administrative functions, including creating users and roles; granting privileges and roles; and creating, modifying, and deleting schema objects in any schema. This role grants all system privileges.

Note: Users SYS and SYSTEM have the privileges to start or shut down the database instance. These privileges are not included in the DBA role.

Creating and Granting Privileges to a Role

- Create a role:

```
CREATE ROLE manager;
```

- Grant privileges to a role:

```
GRANT create table, create view
TO manager;
```

- Grant a role to users:

```
GRANT manager TO alice;
```

ORACLE

Creating a Role

The example in the slide creates a `manager` role and then enables the manager to create tables and views. It then grants user `alice` the role of a manager. Now `alice` can create tables and views.

If users have multiple roles granted to them, they receive all the privileges associated with all the roles.

Changing Your Password

- The DBA creates your user account and initializes your password.
- You can change your password by using the `ALTER USER` statement.

```
ALTER USER user
IDENTIFIED BY password;
```

- Example:

```
ALTER USER demo
IDENTIFIED BY employ;
```


The DBA creates an account and initializes a password for every user. You can change your password by using the `ALTER USER` statement.

The slide example shows how the `demo` user changes the password by using the `ALTER USER` statement.

For more information, see the *Oracle Database SQL Language Reference* for Oracle Database 12*c*.

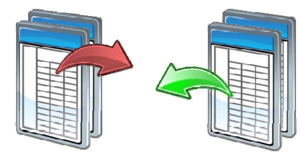
Lesson Agenda

- System privileges
- Creating a role
- Object privileges
- Revoking object privileges
- Oracle Cloud Service administration roles

ORACLE

Object Privileges

- Right to perform a particular action on a specific table, view, sequence, or procedure
- Vary from object to object



ORACLE

An *object privilege* is a privilege or right to perform a particular action on a specific table, view, sequence, or procedure. Each object has a particular set of grantable privileges. Different object privileges are available for different types of schema objects.

Object Privileges

Object Privilege	Table	View	Sequence
ALTER	✓		✓
DELETE	✓	✓	
INDEX	✓		
INSERT	✓	✓	
REFERENCES	✓		
SELECT	✓	✓	✓
UPDATE	✓	✓	

ORACLE

Each object has a particular set of grantable privileges. The table in the slide lists the privileges for various objects. Note that the only privileges that apply to a sequence are `SELECT` and `ALTER`.

A `SELECT` privilege can be restricted by creating a view with a subset of columns and granting the `SELECT` privilege only on the view. A privilege granted on a synonym is converted to a privilege on the base table referenced by the synonym.

Note: With the `REFERENCES` privilege, you can ensure that other users can create `FOREIGN KEY` constraints that reference your table.

Object Privileges

- An owner has all the privileges on the object.
- An owner can grant specific privileges on the objects to other users.

Object privilege to be granted

Specifies ALL object privileges

```
GRANT object_priv|ALL [(columns)]
ON               object
TO               {user|role|PUBLIC}
[WITH GRANT OPTION];
```

Column from table or view on which privileges are granted

Table, view, sequence, or procedure on which privileges are granted

Enables the grantee to grant the object privileges to other users and roles

Identifies to whom the privilege is given; PUBLIC gives to all users

ORACLE

Granting Object Privileges

A user automatically has all object privileges for schema objects contained in the user's schema. A user can grant any object privilege on any schema object that the user owns to any other user or role. If the grant includes `WITH GRANT OPTION`, the grantee can further grant the object privilege to other users; otherwise, the grantee can use the privilege but cannot grant it to other users.

Granting Object Privileges

- Grant query privileges on the EMPLOYEES table:

```
GRANT  select
ON     employees
TO     demo;
```

- Grant privileges to update specific columns to users and roles:

```
GRANT  update (department_name, location_id)
ON     departments
TO     demo, manager;
```

ORACLE

Guidelines

- To grant privileges on an object, the object must be in your own schema, or you must have been granted the object privileges WITH GRANT OPTION.
- An object owner can grant any object privilege on the object to any other user or role of the database.
- The owner of an object automatically acquires all object privileges on that object.

The first example in the slide grants the demo user the privilege to query your EMPLOYEES table. The second example grants UPDATE privileges on specific columns in the DEPARTMENTS table to demo and to the manager role.

For example, if your schema is oraxx, and the demo user now wants to use a SELECT statement to obtain data from your EMPLOYEES table, the syntax he or she must use is:

```
SELECT  * FROM oraxx.employees;
```

Alternatively, the demo user can create a synonym for the table and issue a SELECT statement from the synonym:

```
CREATE SYNONYM emp FOR oraxx.employees;
SELECT * FROM emp;
```

Note: DBAs generally allocate system privileges; any user who owns an object can grant object privileges.

Passing On Your Privileges

- Give a user authority to pass along privileges:

```
GRANT   select, insert
ON      departments
TO      demo
WITH    GRANT OPTION;
```

- Allow all users on the system to query data from the DEPARTMENTS table:

```
GRANT   select
ON      departments
TO      PUBLIC;
```

ORACLE

WITH GRANT OPTION

A privilege that is granted with the WITH GRANT OPTION clause can be passed on to other users and roles by the grantee. Object privileges granted with the WITH GRANT OPTION clause are revoked when the grantor's privilege is revoked. You can specify WITH GRANT OPTION only when granting to a user or to PUBLIC, not when granting to a role.

The grantor must meet one or more of the following criteria. The grantor:

- Must be the object owner or must have object access with GRANT OPTION from the user
- Must have the GRANT ANY OBJECT PRIVILEGE system privilege and an object privilege on the object

The example in the slide gives the demo user access to your DEPARTMENTS table with the privileges to query the table and add rows to the table. The example also shows that demo can give others these privileges.

PUBLIC Keyword

An owner of a table can grant access to all users by using the PUBLIC keyword. The second example allows all users on the system to query data from the DEPARTMENTS table.

Confirming Granted Privileges

Data Dictionary View	Description
ROLE_SYS_PRIVS	System privileges granted to roles
ROLE_TAB_PRIVS	Table privileges granted to roles
USER_ROLE_PRIVS	Roles accessible by the user
USER_SYS_PRIVS	System privileges granted to the user
USER_TAB_PRIVS_MADE	Object privileges granted on the user's objects
USER_TAB_PRIVS_RECD	Object privileges granted to the user
USER_COL_PRIVS_MADE	Object privileges granted on the columns of the user's objects
USER_COL_PRIVS_RECD	Object privileges granted to the user on specific columns

ORACLE

If you attempt to perform an unauthorized operation, such as deleting a row from a table for which you do not have the DELETE privilege, the Oracle server does not permit the operation to take place.

If you receive the Oracle server error message "Table or view does not exist," you have done either of the following:

- Named a table or view that does not exist
- Attempted to perform an operation on a table or view for which you do not have the appropriate privilege

The data dictionary is organized in tables and views and contains information about the database. You can access the data dictionary to view the privileges that you have. The table in the slide describes various data dictionary views.

You learn about data dictionary views in the lesson titled "Introduction to Data Dictionary Views."

Note: The ALL_TAB_PRIVS_MADE dictionary view describes all the object grants made by the user or made on the objects owned by the user.

Lesson Agenda

- System privileges
- Creating a role
- Object privileges
- Revoking object privileges
- Oracle Cloud Service administration roles

ORACLE

Revoking Object Privileges

- You use the `REVOKE` statement to revoke privileges granted to other users.
- Privileges granted to others through the `WITH GRANT OPTION` clause are also revoked.

```
REVOKE {privilege [, privilege...]|ALL}
ON       object
FROM   {user[, user...]|role|PUBLIC};
```


You can remove privileges granted to other users by using the `REVOKE` statement. When you use the `REVOKE` statement, the privileges that you specify are revoked from the users you name and from any other users to whom those privileges were granted by the revoked user.

For more information, see the *Oracle Database SQL Language Reference* for Oracle Database12*c*.

Note: If a user leaves the company and you revoke his or her privileges, you must regrant any privileges that this user granted to other users. If you drop the user account without revoking privileges from it, the system privileges granted by this user to other users are not affected by this action.

Revoking Object Privileges

Revoke the SELECT and INSERT privileges given to the demo user on the DEPARTMENTS table.

```
REVOKE  select, insert
ON      departments
FROM    demo;
```

```
REVOKE succeeded.
```


The example in the slide revokes the SELECT and INSERT privileges given to the demo user on the DEPARTMENTS table.

Note: If a user is granted a privilege with the WITH GRANT OPTION clause, that user can also grant the privilege with the WITH GRANT OPTION clause, so that a long chain of grantees is possible, but no circular grants (granting to a grant ancestor) are permitted. If the owner revokes a privilege from a user who granted the privilege to other users, the revoking cascades to all the privileges granted.

For example, if user A grants a SELECT privilege on a table to user B including the WITH GRANT OPTION clause, user B can grant to user C the SELECT privilege with the WITH GRANT OPTION clause as well, and user C can then grant to user D the SELECT privilege. If user A revokes privileges from user B, the privileges granted to users C and D are also revoked.

Lesson Agenda

- System privileges
- Creating a role
- Object privileges
- Revoking object privileges
- Oracle Cloud Service administration roles

ORACLE

Oracle Cloud Service Administration Roles

Roles	Tasks
Buyer	• Controls the buying process • Designates the initial account administrator for the Oracle Cloud service
Account Administrator	• Activates and creates identity domains • Monitors status and usage of services
Identity Administrator	• Creates and manages users who access the Oracle Cloud services • Assigns and manages user roles
Service Administrator	• Administers an Oracle Cloud service • Monitors the service status and usage

ORACLE

With the Oracle Database as a Service cloud technology, various Oracle Cloud user roles have originated. The slide describes the user roles and the privileges associated with each role.

A user can be assigned more than one role. A role may include privileges that let the user purchase an Oracle Cloud service, manage one or more Oracle Cloud services, or manage the accounts of the users who can access a service.

These roles are not predefined.

When Oracle Cloud services are provisioned in an identity domain, Oracle Cloud automatically populates the My Services application with several roles and several user accounts. These roles:

- Are based on the type of Oracle Cloud service being provisioned
- Include both administrative roles and non-administrative roles
- Grant certain privileges to the users based on the role assigned to them. Users can be assigned more than one role.

There are many concepts and details associated with this topic that are out of the scope of this course. For more information, refer to the Oracle Cloud Help Center at: https://docs.oracle.com/cloud/latest/

Quiz

Which one of the following is a collection of objects such as tables, views, and sequences, that is owned by a database user and has the same name?

a. Administrator
b. Schema
c. Privilege
d. Role

Answer: b

Quiz

A database role can only be created by a user with DBA privileges.

a. True
b. False

Answer: a

Quiz

Which command can you use to change your password?

a. `ALTER USER`

b. `REVOKE`

c. `GRANT`

ORACLE

Answer: a

Quiz

Which of the following statements are true?

a. After a user creates an object, the user can pass along any of the available object privileges to other users by using the `GRANT` statement.

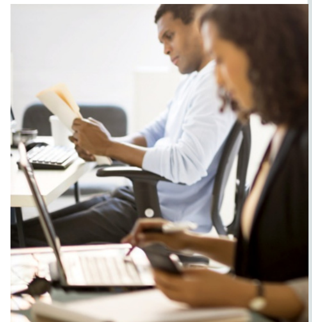
b. Users cannot view the privileges granted to them and those that are granted on their objects.

Answer: a

Summary

In this lesson, you should have learned how to:

- Differentiate system privileges from object privileges
- Grant privileges on tables
- Grant roles
- Distinguish between privileges and roles



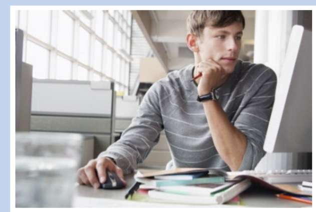
DBAs establish initial database security for users by assigning privileges to the users.

- The DBA creates users who must have a password. The DBA is also responsible for establishing the initial system privileges for a user.
- After the user has created an object, the user can pass along any of the available object privileges to other users or to all users by using the `GRANT` statement.
- A DBA can create roles by using the `CREATE ROLE` statement to pass along a collection of system or object privileges to multiple users. Roles make granting and revoking privileges easier to maintain.
- Users can change their passwords by using the `ALTER USER` statement.
- You can remove privileges from users by using the `REVOKE` statement.
- With data dictionary views, users can view the privileges granted to them and those that are granted on their objects.

Practice 23: Overview

This practice covers the following topics:

- Creating a new user
- Granting the user system privileges through a pre-defined role
- Granting the user privileges to your table
- Accessing data in the new users SQL Developer session



In this practice, you learn how to grant other users privileges to your table and how to modify another user's table through the privileges granted to you.


Lesson 24: Advanced Data Manipulation
ORACLE
Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Course Roadmap

- Lesson 1: Course Overview
- Unit 1: Relational Database and SQL Overview
- Unit 2: Retrieving and Sorting Data
- Unit 3: Joins, Subqueries, and Set Operators
- Unit 4: DML and DDL
- Unit 5: Managing Relational Database
- Unit 6: Advance Queries and Database Management System

You are here

- Lesson 21: Using Advanced Subqueries
- Lesson 22: Manipulating Data by Using Advanced Subqueries
- Lesson 23: Controlling User Access
- Lesson 24: Advanced Data Manipulation
- Lesson 25: Managing Multiple Timezones

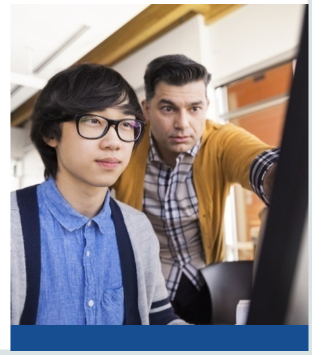
ORACLE

In Unit 6, you will be introduced to some advanced features of SQL. You will learn to write advanced subqueries. You will learn to create users and manage users. You will also learn about managing multiple timezomes.

Objectives

After completing this lesson, you should be able to:

- Specify explicit default values in the `INSERT` and `UPDATE` statements
- Describe the features of multitable `INSERT`s
- Use the following types of multitable `INSERT`s:
  - Unconditional `INSERT`
  - Conditional `INSERT ALL`
  - Conditional `INSERT FIRST`
  - Pivoting `INSERT`
- Merge rows in a table
- Perform flashback operations
- Track changes made to data over a period of time



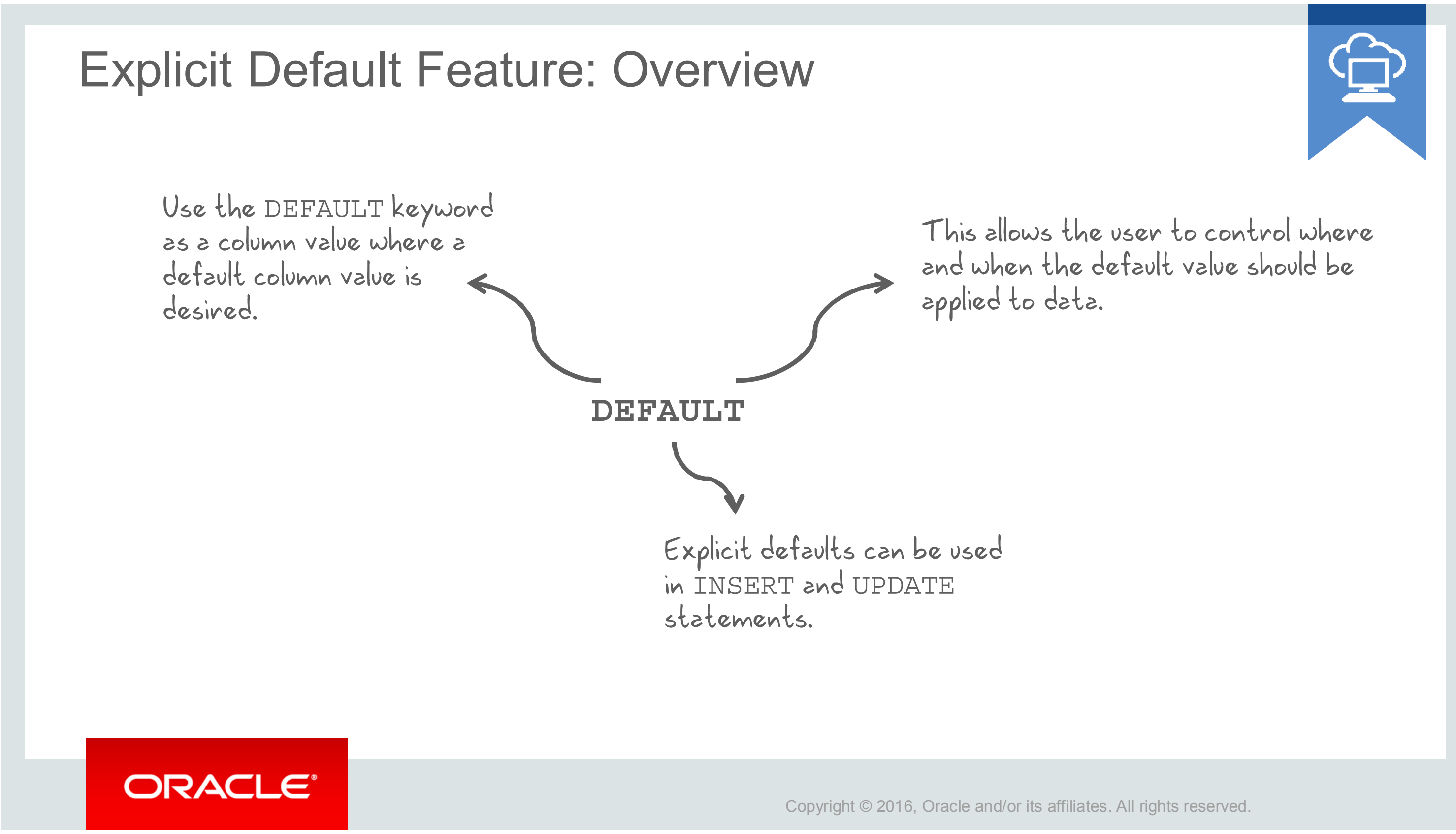
ORACLE

In this lesson, 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, performing flashback operations, and tracking changes in the database.

Lesson Agenda

- Specifying explicit default values in `INSERT` and `UPDATE` statements
- Using the following types of multitable `INSERT`s:
  - Unconditional `INSERT`
  - Conditional `INSERT ALL`
  - Conditional `INSERT FIRST`
  - Pivoting `INSERT`
- Merging rows in a table
- Performing flashback operations
- Tracking changes to data over a period of time

ORACLE



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 having to hard code the default value in your programs or query 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; therefore, this is a very important feature.

Using Explicit Default Values

- 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;
```


Specify DEFAULT to set the column to the value that was 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 no default value is 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.

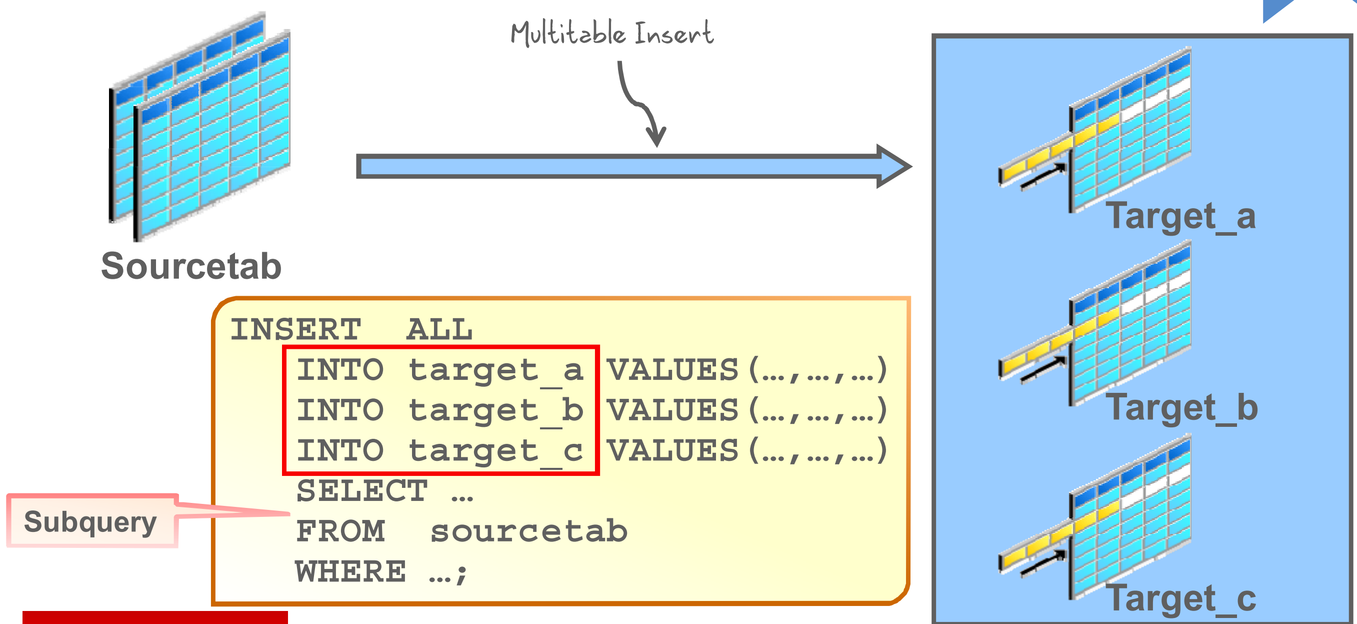
Note: When creating a table, you can specify a default value for a column. This is discussed in the lesson titled “Introduction to Data Definition Language.”

Lesson Agenda

- Specifying explicit default values in INSERT and UPDATE statements
- Using the following types of multitable INSERTs:
  - Unconditional INSERT
  - Conditional INSERT ALL
  - Conditional INSERT FIRST
  - Pivoting INSERT
- Merging rows in a table
- Performing flashback operations
- Tracking changes to data over a period of time

ORACLE

Multitable INSERT Statements: Overview



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.

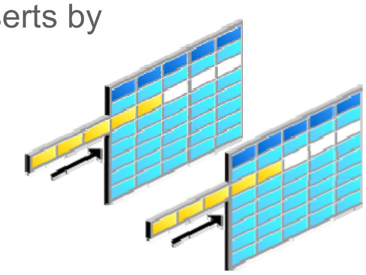
Multitable INSERT statements are useful in a data warehouse scenario. You need to load your data warehouse regularly so that it can serve its purpose of facilitating business analysis. To do this, data from one or more operational systems must be extracted and copied into the warehouse. The process of extracting data from the source system and bringing it into the data warehouse is commonly called ETL, which stands for extraction, transformation, and loading.

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 by using SQL operations. A multitable INSERT statement is one of the techniques for implementing SQL data transformations.

Multitable `INSERT` Statements: Overview

- 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:
  - Single DML versus multiple `INSERT…SELECT` statements
  - Single DML versus a procedure to perform multiple inserts by using the `IF...THEN` syntax



ORACLE

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.

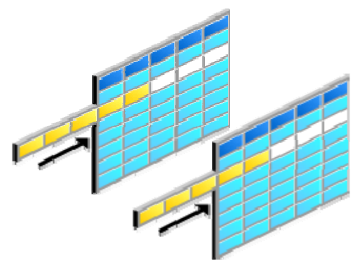
As with the existing `INSERT ... SELECT` statement, the new statement can be parallelized and used with the direct-load mechanism for faster performance.

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
- Conditional INSERT FIRST
- Pivoting INSERT



You 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.
- **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.
- **Pivoting `INSERT`:** This is a special case of the unconditional INSERT ALL.

Multitable INSERT Statements

- Syntax for multitable INSERT:

```
{ ALL
{ insert_into_clause [ values_clause ]}...
| conditional_insert_clause
 } subquery
```

- conditional_insert_clause:

```
[ ALL | FIRST ]
 WHEN condition THEN insert_into_clause
                     [ values_clause ]
                     [ insert_into_clause [ values_clause ]]…
  [ ELSE insert_into_clause
                     [ values_clause ]
                     [ insert_into_clause [ values_clause ]]…
 ]
```

ORACLE

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;
```

```
8 rows inserted.
```

ORACLE

The example in the slide inserts rows into both the SAL_HISTORY and the MGR_HISTORY tables.

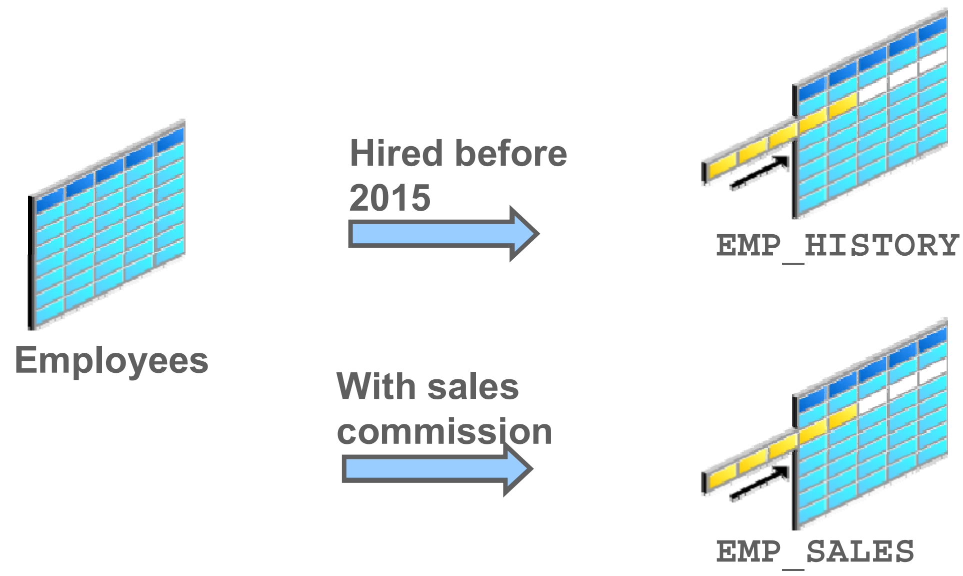
The SELECT statement retrieves the details of employee ID, hire date, salary, and manager ID of those employees whose employee ID is greater than 200 from the EMPLOYEES table. The details of the employee ID, hire date, and salary are inserted into the SAL_HISTORY table. The details of employee ID, manager 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 12 rows were inserted:

```
SELECT COUNT(*) total_in_sal FROM sal_history;
SELECT COUNT(*) total_in_mgr FROM mgr_history;
```

Conditional `INSERT ALL`: Example



For all employees in the `EMPLOYEES` table, if an employee was hired before 2015, insert that employee record into employee history. If the employee earns a sales commission, insert the record information into the `EMP_SALES` table. The SQL statement is shown on the next page.

Conditional `INSERT ALL`

```
INSERT  ALL
 WHEN HIREDATE < '01-JAN-15' 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;
```

```
19 rows inserted.
```


The example in the slide is similar to the example in the previous slide because it inserts rows into both the `EMP_HISTORY` and `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 is before 2015 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;
```

Result: 15 rows fetched.

```
SELECT count(*) FROM emp_sales;
```

Result: 4 rows fetched.

You can also optionally use the ELSE clause with the INSERT ALL statement.

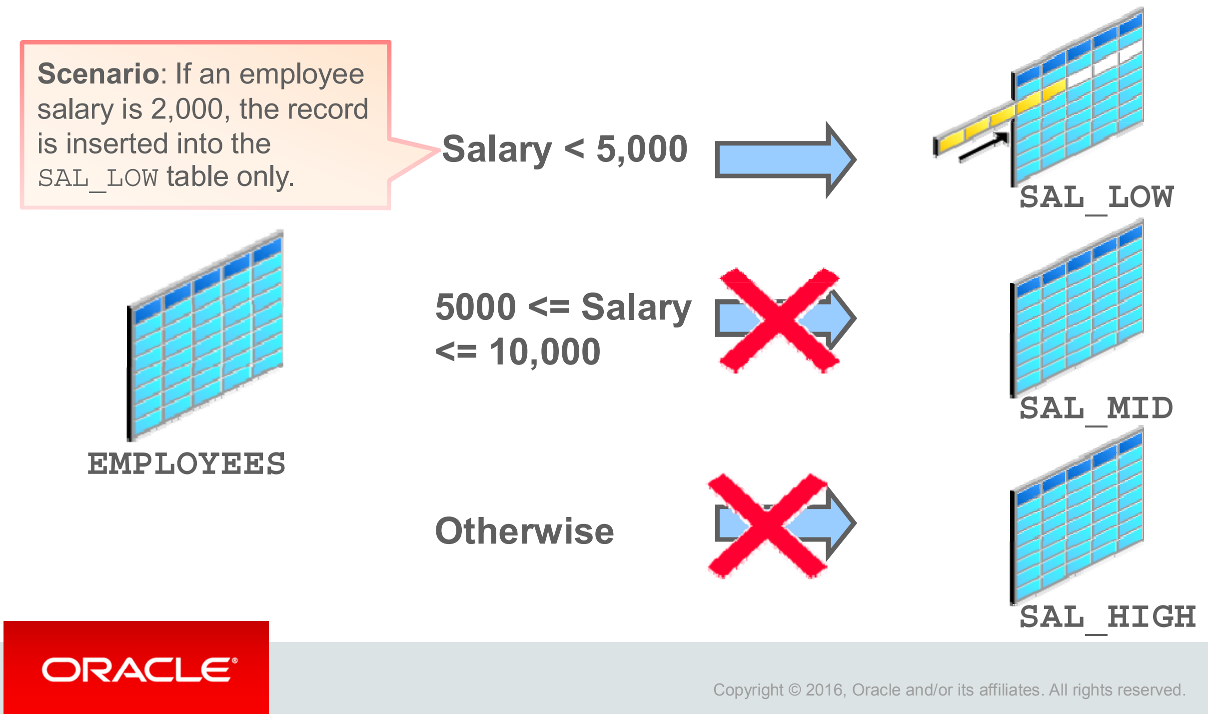
Example:

```
INSERT ALL
WHEN job_id IN
(select job_id FROM jobs WHERE job_title LIKE '%Manager%') THEN
INTO managers2(last_name,job_id,SALARY)
VALUES (last_name,job_id,SALARY)
WHEN SALARY>10000 THEN
INTO richpeople(last_name,job_id,SALARY)
VALUES (last_name,job_id,SALARY)
ELSE
INTO poorpeople VALUES (last_name,job_id,SALARY)
SELECT * FROM employees;
```

Result:

```
24 rows inserted
```

Conditional INSERT FIRST: Example



For all employees in the EMPLOYEES table, insert 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 is shown on the next page.

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;
```

```
20 rows inserted.
```

ORACLE

The SELECT statement retrieves details such as employee ID, last name, and salary for every employee in the EMPLOYEES table. For each employee record, the information 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 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) evaluates to true, the Oracle Server executes the corresponding INTO clause for the ELSE clause.

A total of 20 rows are inserted:

```
SELECT count(*) low FROM sal_low;

6 rows fetched.



SELECT count(*) mid FROM sal_mid;

6 rows fetched.



SELECT count(*) high FROM sal_high;

8 rows fetched.
```

Pivoting INSERT

Convert the set of sales records from the nonrelational database table to the 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

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, must be 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 to this problem is shown on the next page.

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

In the example in the slide, the sales data is received from the nonrelational database table, SALES_SOURCE_DATA, which includes details of the sales performed by a sales representative on each day of a week, for a week with a particular week ID.

DESC SALES_SOURCE_DATA

```
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
```

```
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

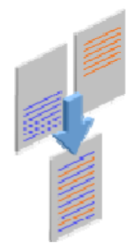
Observe in the preceding example that by using a pivoting `INSERT`, one row from the `SALES_SOURCE_DATA` table is converted into five records for the relational table, `SALES_INFO`.

Lesson Agenda

- Specifying explicit default values in `INSERT` and `UPDATE` statements
- Using the following types of multitable `INSERT`s:
  - Unconditional `INSERT`
  - Conditional `INSERT ALL`
  - Conditional `INSERT FIRST`
  - Pivoting `INSERT`
- Merging rows in a table
- Performing flashback operations
- Tracking changes to data over a period of time

`MERGE` Statement

- Provides the ability to conditionally update, insert, or delete data in 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



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 in a table, thus avoiding multiple DML statements. The decision whether to perform update, insert, or delete in 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);
```

ORACLE

Merging Rows

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 | WHEN NOT MATCHED	Instructs the server on how it should respond to the results of the join condition

Note: For more information, see *Oracle Database SQL Language Reference* for Oracle Database 12*c*.

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);
```

```
20 rows merged.
```

ORACLE


```
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;
```

Observe that in the output, there are some employees with SALARY < 10000 and there are two 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. The records of the two employees with values in the COMMISSION_PCT column are deleted. If a match is not found, rows are inserted into the COPY_EMP3 table.

Merging Rows: Example

```
TRUNCATE TABLE copy_emp3;
SELECT * FROM copy_emp3;
no rows selected
```

```
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, ...
```

```
SELECT * FROM copy_emp3;
20 rows selected.
```

ORACLE

The examples in the slide show that the `COPY_EMP3` table is empty. The `c.employee_id = e.employee_id` condition is evaluated. The condition returns false—there are no matches. The logic falls into the `WHEN NOT MATCHED` clause, and the `MERGE` command inserts the rows of the `EMPLOYEES` table into the `COPY_EMP3` table. This means that the `COPY_EMP3` table now has exactly the same data as in the `EMPLOYEES` table.

```
SELECT employee_id, salary, commission_pct from copy_emp3;
```

Lesson Agenda

- Specifying explicit default values in `INSERT` and `UPDATE` statements
- Using the following types of multitable `INSERTs`:
  - Unconditional `INSERT`
  - Conditional `INSERT ALL`
  - Conditional `INSERT FIRST`
  - Pivoting `INSERT`
- Merging rows in a table
- Performing flashback operations
- Tracking changes to data over a period of time

`FLASHBACK TABLE` Statement

- Enables you to recover tables to a specified point in time with a single statement
- Restores table data along with associated indexes and constraints
- Enables you to revert the table and its contents to a certain point in time or System Change Number (SCN)

Oracle Flashback Table enables you to recover tables to a specified point in time with a single statement. You can restore table data along with the associated indexes and constraints while the database is online, undoing changes to only the specified tables.

The Flashback Table feature is similar to a self-service repair tool. For example, if a user accidentally deletes important rows from a table, and wants to recover the deleted rows, you can use the `FLASHBACK TABLE` statement to restore the table to the time before the deletion and see the missing rows in the table.

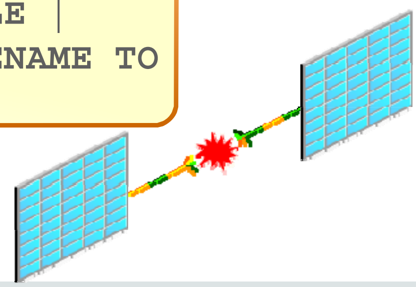
When using the `FLASHBACK TABLE` statement, you can revert the table and its contents to a certain time or to an SCN.

Note: The SCN is an integer value associated with each change to the database. It is a unique incremental number in the database. Every time you commit a transaction, a new SCN is recorded.

FLASHBACK TABLE Statement

- Repair tool for accidental table modifications
  - Restores a table to an earlier point in time
  - Offers ease of use, availability, and fast execution
  - Is performed in place
- Syntax:

```
FLASHBACK TABLE [ schema. ] table [, [ schema.
] table ]... TO { { { SCN | TIMESTAMP } expr |
RESTORE POINT restore_point } [ { ENABLE |
DISABLE } TRIGGERS ] | BEFORE DROP [ RENAME TO
table ] } ;
```



ORACLE

Self-Service Repair Facility

Oracle Database provides a SQL data definition language (DDL) command, `FLASHBACK TABLE`, to restore the state of a table to an earlier point in time in case it is inadvertently deleted or modified. The `FLASHBACK TABLE` command is a self-service repair tool to restore data in a table along with the associated attributes such as indexes or views. This is done, while the database is online, by rolling back only subsequent changes to the given table. Compared to traditional recovery mechanisms, this feature offers significant benefits such as ease of use, availability, and faster restoration. It also takes the burden off the DBA to find and restore application-specific properties. The flashback table feature does not address physical corruption caused because of a bad disk.

Syntax

You can invoke a `FLASHBACK TABLE` operation on one or more tables, even on tables in different schemas. You specify the point in time to which you want to revert by providing a valid time stamp. By default, database triggers are disabled during the flashback operation for all the tables that are involved. You can override this default behavior by specifying the `ENABLE TRIGGERS` clause.

Note: For more information about recycle bin and flashback semantics, refer to *Oracle Database Administrator's Guide* for Oracle Database 12*c*.

Using the FLASHBACK TABLE Statement

```
DROP TABLE emp3;
```

```
table EMP3 dropped.
```

```
SELECT original_name, operation, droptime FROM
recyclebin;
```

	ORIGINAL_NAME	OPERATION	DROPTIME
1	EMP3	DROP	2016-03-22:09:04:13

...

```
FLASHBACK TABLE emp3 TO BEFORE DROP;
```

```
table EMP3 succeeded.
```

ORACLE

Syntax and Examples

The example restores the `EMP3` table to a state before a `DROP` statement.

The recycle bin is actually a data dictionary table containing information about dropped objects. Dropped tables and any associated objects—such as, indexes, constraints, nested tables, and so on—are not removed and still occupy space. They continue to count against user space quotas until specifically purged from the recycle bin, or until they must be purged by the database because of tablespace space constraints.

Each user can be thought of as an owner of a recycle bin because, unless a user has the `SYSDBA` privilege, the only objects that the user has access to in the recycle bin are those that the user owns. A user can view his or her objects in the recycle bin by using the following statement:

```
SELECT * FROM RECYCLEBIN;
```

When you drop a user, any objects belonging to that user are not placed in the recycle bin and any objects in the recycle bin are purged.

You can purge the recycle bin with the following statement:

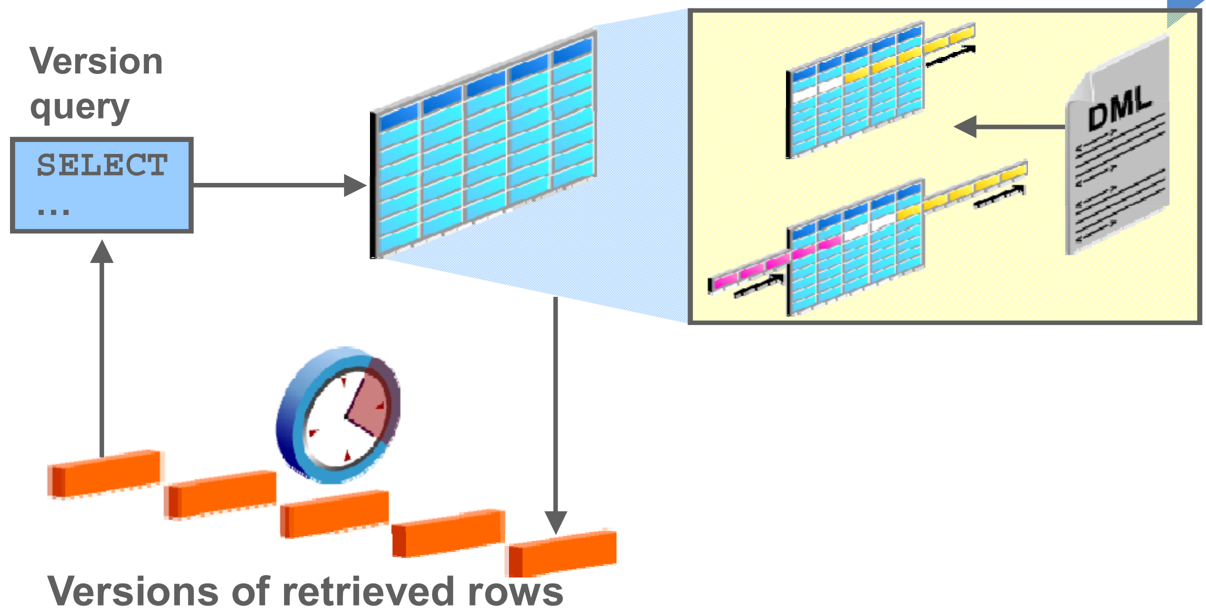
```
PURGE RECYCLEBIN;
```

Lesson Agenda

- Specifying explicit default values in `INSERT` and `UPDATE` statements
- Using the following types of multitable `INSERTs`:
  - Unconditional `INSERT`
  - Conditional `INSERT ALL`
  - Conditional `INSERT FIRST`
  - Pivoting `INSERT`
- Merging rows in a table
- Performing flashback operations
- Tracking changes to data over a period of time

ORACLE

Tracking Changes in Data



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. You can use Oracle Flashback Query to retrieve data as it existed at an earlier 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 result of a version query behaves as though the `WHERE` clause were applied to the versions of the rows. The version query returns versions of the rows only across transactions.

System change number (SCN): The Oracle server assigns an SCN to identify the redo records for each committed transaction.

Flashback Query: Example

```
SELECT salary FROM employees3
WHERE last_name = 'Matos';
```

```
UPDATE employees3 SET salary = 4000
WHERE last_name = 'Matos';

SELECT salary FROM employees3
WHERE last_name = 'Matos';
```

```
SELECT salary FROM employees3
AS OF TIMESTAMP (SYSTIMESTAMP - INTERVAL '1' MINUTE)
WHERE last_name = 'Matos';
```

1

	SALARY
1	2600

2

	SALARY
1	4000

3

	SALARY
1	2600

To use Oracle Flashback Query, use a `SELECT` statement with an `AS OF` clause. Oracle Flashback Query retrieves data as it existed at some time in the past. The query explicitly references a past time through a time stamp or System Change Number (SCN). It returns committed data that was current at that point in time.

In the example in the slide, the salary for employee “Matos” is retrieved (1). The salary for employee “Matos” is increased to 4000 (2). To learn what the value was before the update, you can use the Flashback Query(3).

Oracle Flashback Query can be used in the following scenarios:

- Recovering lost data or undoing incorrect, committed changes. For example, if you mistakenly delete or update rows, and then commit them, you can immediately undo the mistake.
- Comparing current data with the corresponding data at some time in the past. For example, you can run a daily report that shows the change in data from yesterday. You can compare individual rows of table data or find intersections or unions of sets of rows.
- Checking the state of transactional data at a particular time

Flashback Version Query: Example

```
SELECT salary FROM employees3
WHERE  employee_id = 107;
```
1

```
UPDATE employees3 SET salary = salary * 1.30
WHERE  employee_id = 107;

COMMIT;
```
2

```
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®

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 the 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 the 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.

VERSIONS BETWEEN Clause

```
SELECT versions_starttime "START_DATE",
       versions_endtime   "END_DATE",
       salary
FROM   employees3
   VERSIONS BETWEEN SCN MINVALUE
   AND MAXVALUE
WHERE  last_name = 'Lorentz';
```

	START_DATE	END_DATE	SALARY
1	22-MAR-16 09.14.09.000000000 AM	(null)	5460
2	(null)	22-MAR-16 09.14.09.000000000 AM	4200

You can use the VERSIONS BETWEEN clause to retrieve all 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.

Quiz

When you use the `INSERT` or `UPDATE` command, the `DEFAULT` keyword saves you from hard-coding the default value in your programs or querying the dictionary to find it.

a. True

b. False

Answer: a

Quiz

In all the cases, when you execute a DROP TABLE command, the database renames the table and places it in the recycle bin, from where it can later be recovered by using the FLASHBACK TABLE statement.

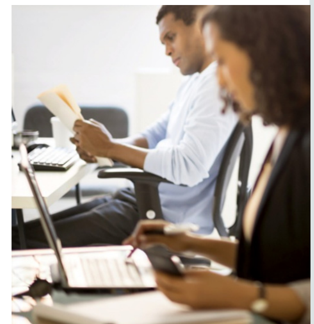
a. True
b. False

Answer: b

Summary

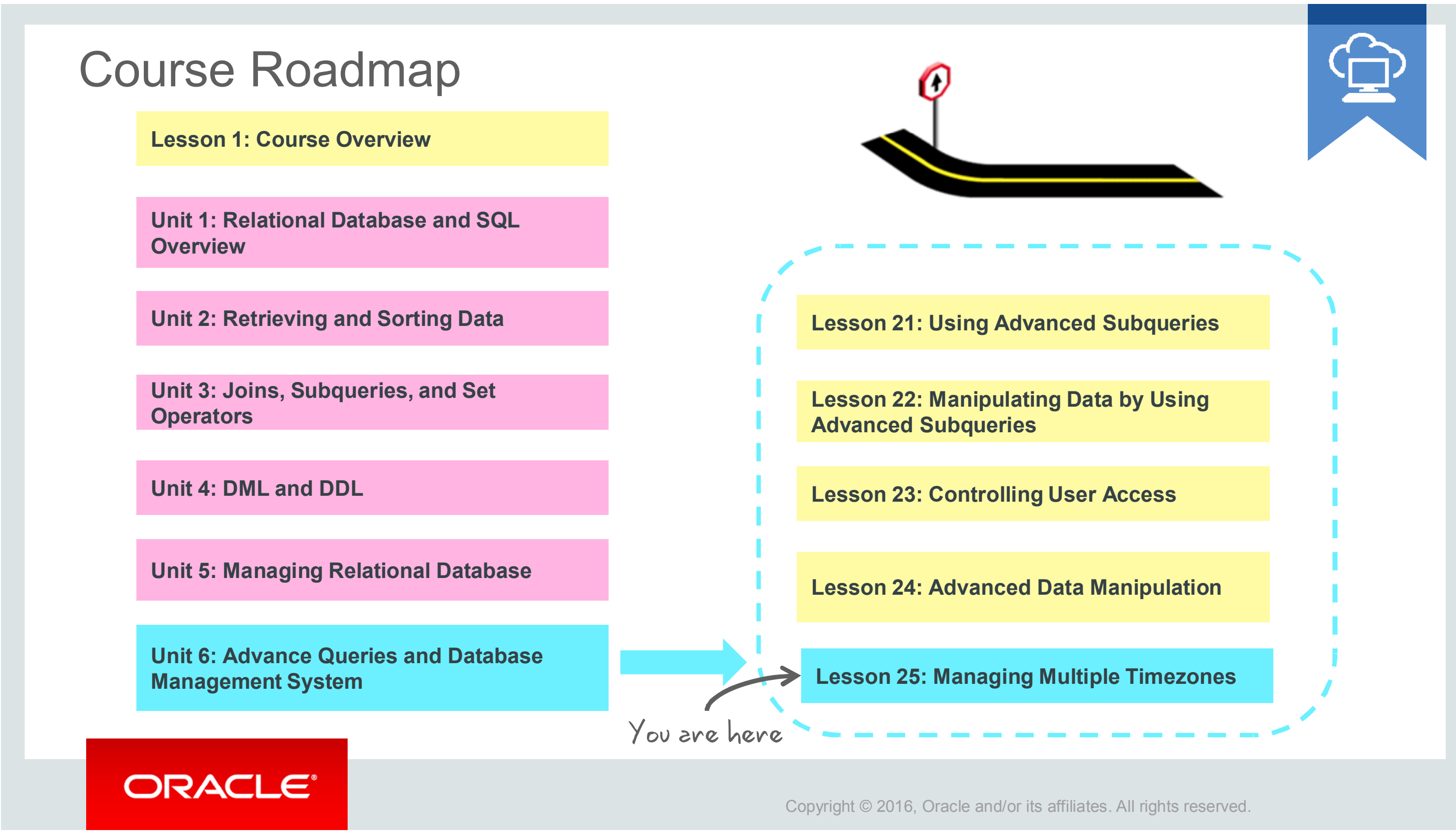
In this lesson, you should have learned how to:

- Specify explicit default values in `INSERT` and `UPDATE` statements
- Describe the features of multitable `INSERT`s
- Use the following types of multitable `INSERT`s:
  - Unconditional `INSERT`
  - Conditional `INSERT ALL`
  - Conditional `INSERT FIRST`
  - Pivoting `INSERT`
- Merge rows in a table
- Perform flashback operations
- Track changes to data over a period of time



In this lesson, you also should have learned about multitable `INSERT` statements, the `MERGE` statement, and tracking changes in the database.


Lesson 25: Managing Multiple Time Zones
ORACLE
Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

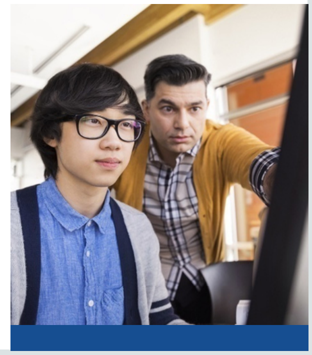


In Unit 6, you will be introduced to some advanced features of SQL. You will learn to write advanced subqueries. You will learn to create users and manage users. You will also learn about managing multiple timezones.

Objectives

After completing this lesson, you should be able to:

- Use data types similar to DATE that store fractional seconds and track time zones
- Use data types that store the difference between two datetime values
- Use the following datetime functions:
  - CURRENT_DATE
  - CURRENT_TIMESTAMP
  - LOCALTIMESTAMP
  - DBTIMEZONE
  - SESSIONTIMEZONE
  - EXTRACT
  - TZ_OFFSET
  - FROM_TZ
  - TO_TIMESTAMP
  - TO_YMINTERVAL
  - TO_DSINTERVAL



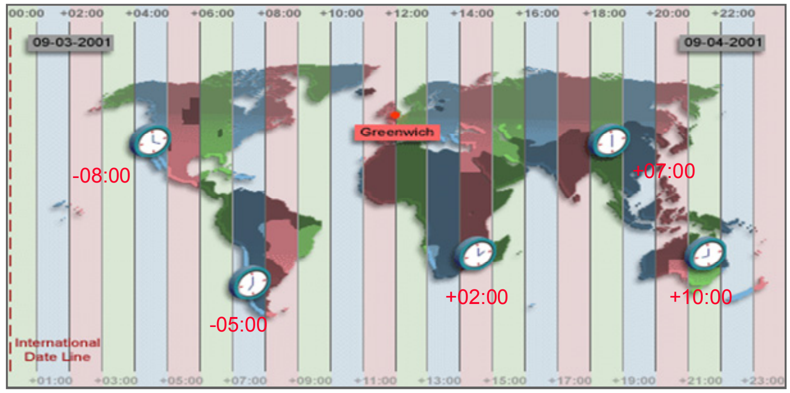
In this lesson, you learn how to use data types similar to DATE that store fractional seconds and track time zones. This lesson also addresses some of the datetime functions available in the Oracle database.

Lesson Agenda

- CURRENT_DATE, CURRENT_TIMESTAMP, and LOCALTIMESTAMP
- INTERVAL data types
- Using the following functions:
  - EXTRACT
  - TZ_OFFSET
  - FROM_TZ
  - TO_TIMESTAMP
  - TO_YMINTERVAL
  - TO_DSINTERVAL

ORACLE

Time Zones



The image represents the time for each time zone when Greenwich time is 12:00.

The hours of the day are measured by earth's rotation. The time of day at any particular moment depends on where you are. When it is noon in Greenwich, England, it is midnight along the International Date Line. The earth is divided into 24 time zones, one for each hour of the day. The time along the prime meridian in Greenwich, England, is known as Greenwich Mean Time (GMT). GMT is now known as Coordinated Universal Time (UTC). UTC is the time standard against which all other time zones in the world are referenced. It is the same all year round and is not affected by summer time or daylight saving time. The meridian line is an imaginary line that runs from the North Pole to the South Pole. It is known as zero longitude and it is the line from which all other lines of longitude are measured. All time is measured relative to UTC and all places have a latitude (their distance north or south of the equator) and a longitude (their distance east or west of the Greenwich meridian).

TIME_ZONE Session Parameter

TIME_ZONE may be set to:

- An absolute offset
- Database time zone
- OS local time zone
- A named region

```
ALTER SESSION SET TIME_ZONE = '-05:00';
ALTER SESSION SET TIME_ZONE = dbtimezone;
ALTER SESSION SET TIME_ZONE = local;
ALTER SESSION SET TIME_ZONE = 'America/New_York';
```


The Oracle database supports storing the time zone in your date and time data, as well as fractional seconds. The ALTER SESSION command can be used to change the time zone values in a user's session. The time zone values can be set to an absolute offset, a named time zone, a database time zone, or the local time zone.

CURRENT_DATE, CURRENT_TIMESTAMP, and LOCALTIMESTAMP

- CURRENT_DATE:
  – Returns the current date from the user session
  – Has a data type of DATE
- CURRENT_TIMESTAMP:
  – Returns the current date and time from the user session
  – Has a data type of TIMESTAMP WITH TIME ZONE
- LOCALTIMESTAMP:
  – Returns the current date and time from the user session
  – Has a data type of TIMESTAMP

ORACLE

The CURRENT_DATE and CURRENT_TIMESTAMP functions return the current date and current time stamp, respectively. The data type of CURRENT_DATE is DATE. The data type of CURRENT_TIMESTAMP is TIMESTAMP WITH TIME ZONE. The values returned display the time zone displacement of the SQL session that is executing the functions. Time zone displacement is the difference (in hours and minutes) between local time and UTC. The TIMESTAMP WITH TIME ZONE data type has the format:

```
TIMESTAMP [ (fractional_seconds_precision) ] WITH TIME ZONE
```

where fractional_seconds_precision optionally specifies the number of digits in the fractional part of the SECOND datetime field and can be a number in the range 0 through 9. The default is 6.

The LOCALTIMESTAMP function returns the current date and time in the session time zone. The difference between LOCALTIMESTAMP and CURRENT_TIMESTAMP is that LOCALTIMESTAMP returns a TIMESTAMP value, whereas CURRENT_TIMESTAMP returns a TIMESTAMP WITH TIME ZONE value.

These functions are national language support (NLS)–sensitive—that is, the results will be in the current NLS calendar and datetime formats.

Note: The SYSDATE function returns the current date and time as a DATE data type. You learned how to use the SYSDATE function in the lesson titled “Using Single-Row Functions to Customize Output.”

Comparing Date and Time in a Session's Time Zone

The `TIME_ZONE` parameter is set to `-5:00`, and then `SELECT` statements for each date and time are executed to compare differences.

```
ALTER SESSION
SET NLS_DATE_FORMAT = 'DD-MON-YYYY HH24:MI:SS';
ALTER SESSION SET TIME_ZONE = '-5:00';

SELECT SESSIONTIMEZONE, CURRENT_DATE FROM DUAL;        1

SELECT SESSIONTIMEZONE, CURRENT_TIMESTAMP FROM DUAL;   2

SELECT SESSIONTIMEZONE, LOCALTIMESTAMP FROM DUAL;      3
```

ORACLE

The `ALTER SESSION` command sets the date format of the session to `'DD-MON-YYYY HH24:MI:SS'`—that is, day of month (1–31)-abbreviated name of month-4-digit year hour of day (0–23):minute (0–59):second (0–59).

The example in the slide illustrates that the session is altered to set the `TIME_ZONE` parameter to –5:00. Then the `SELECT` statement for `CURRENT_DATE`, `CURRENT_TIMESTAMP`, and `LOCALTIMESTAMP` is executed to observe the differences in format.

Note: The `TIME_ZONE` parameter specifies the default local time zone displacement for the current SQL session. `TIME_ZONE` is a session parameter only; it is not an initialization parameter. The `TIME_ZONE` parameter is set as follows:

```
TIME_ZONE = '[+ | -] hh:mm'
```

The format mask (`[+ | -] hh:mm`) indicates the hours and minutes before or after UTC.

Comparing Date and Time in a Session's Time Zone

Results of queries:

```
session SET altered.
```

	SESSIONTIMEZONE	CURRENT_DATE
1	-05:00	22-MAR-2016 05:42:43

1

	SESSIONTIMEZONE	CURRENT_TIMESTAMP
1	-05:00	22-MAR-16 05.43.44.646116000 AM -05:00

2

	SESSIONTIMEZONE	LOCALTIMESTAMP
1	-05:00	22-MAR-16 05.44.16.544371000 AM

3

ORACLE

In this case, the `CURRENT_DATE` function returns the current date in the session's time zone, the `CURRENT_TIMESTAMP` function returns the current date and time in the session's time zone as a value of the data type `TIMESTAMP WITH TIME ZONE`, and the `LOCALTIMESTAMP` function returns the current date and time in the session's time zone.

Note: The code example output may vary depending on when the command is run.

DBTIMEZONE and SESSIONTIMEZONE

- Display the value of the database time zone:

```
SELECT DBTIMEZONE FROM DUAL;
```

	DBTIMEZONE
1	-07:00

- Display the value of the session's time zone:

```
SELECT SESSIONTIMEZONE FROM DUAL;
```

	SESSIONTIMEZONE
1	-05:00

The DBA sets the database's default time zone by specifying the `SET TIME_ZONE` clause of the `CREATE DATABASE` statement. If omitted, the default database time zone is the operating system time zone. The database time zone cannot be changed for a session with an `ALTER SESSION` statement.

The `DBTIMEZONE` function returns the value of the database time zone. The return type is a time zone offset (a character type in the format: `'[+|-]TZH:TZM'`) or a time zone region name, depending on how the user specified the database time zone value in the most recent `CREATE DATABASE` or `ALTER DATABASE` statement. The example in the slide shows that the database time zone is set to "–05:00," because the `TIME_ZONE` parameter is in the format:

```
TIME_ZONE = '[+ | -] hh:mm'
```

The `SESSIONTIMEZONE` function returns the value of the current session's time zone. The return type is a time zone offset (a character type in the format `'[+|-]TZH:TZM'`) or a time zone region name, depending on how the user specified the session time zone value in the most recent `ALTER SESSION` statement. The example in the slide shows that the session time zone is offset to UTC by –5 hours. Observe that the database time zone is different from the current session's time zone.

TIMESTAMP Data Types

Data Type	Fields
TIMESTAMP	Year, Month, Day, Hour, Minute, Second with fractional seconds
TIMESTAMP WITH TIME ZONE	Same as the TIMESTAMP data type; also includes: TIMEZONE_HOUR, and TIMEZONE_MINUTE or TIMEZONE_REGION
TIMESTAMP WITH LOCAL TIME ZONE	Same as the TIMESTAMP data type; also includes a time zone offset in its value

ORACLE

The TIMESTAMP data type is an extension of the DATE data type.

```
TIMESTAMP (fractional_seconds_ precision)
```

This data type contains the year, month, and day values of date, as well as the hour, minute, and second values of time, where significant fractional seconds precision is the number of digits in the fractional part of the SECOND datetime field. The accepted values of significant fractional_seconds_precision are 0 through 9. The default is 6.

```
TIMESTAMP (fractional_seconds_precision) WITH TIME ZONE
```

This data type contains all values of TIMESTAMP as well as the time zone displacement value.

```
TIMESTAMP (fractional_seconds_precision) WITH LOCAL TIME ZONE
```

This data type contains all values of TIMESTAMP, with the following exceptions:

- Data is normalized to the database time zone when it is stored in the database.
- When the data is retrieved, users see the data in the session time zone.

TIMESTAMP Fields

Datetime Field	Valid Values
YEAR	–4712 to 9999 (excluding year 0)
MONTH	01 to 12
DAY	01 to 31
HOUR	00 to 23
MINUTE	00 to 59
SECOND	00 to 59.9(N) where 9(N) is precision
TIMEZONE_HOUR	–12 to 14
TIMEZONE_MINUTE	00 to 59

Each datetime data type is composed of several of the fields listed in the slide. Datetimes are mutually comparable and assignable only if they have the same datetime fields.

Difference Between DATE and TIMESTAMP

A

```
-- when hire_date is
of type DATE

SELECT hire_date
FROM emp4;
```

	HIRE_DATE
1	17-SEP-11
2	17-FEB-12
3	17-AUG-13
4	07-JUN-10
5	07-JUN-10
6	17-JUN-11
7	21-SEP-13
8	13-JAN-09
9	03-JAN-14
10	21-MAY-15

…

B

```
ALTER TABLE emp4
MODIFY hire_date
TIMESTAMP(7);
SELECT hire_date
FROM emp4;
```

	HIRE_DATE
1	17-JUN-11 12.00.00.000000000 AM
2	21-SEP-13 12.00.00.000000000 AM
3	13-JAN-09 12.00.00.000000000 AM
4	03-JAN-14 12.00.00.000000000 AM
5	21-MAY-15 12.00.00.000000000 AM
6	07-FEB-15 12.00.00.000000000 AM
7	16-NOV-15 12.00.00.000000000 AM
8	17-OCT-11 12.00.00.000000000 AM
9	29-JAN-13 12.00.00.000000000 AM
10	15-MAR-14 12.00.00.000000000 AM

…

TIMESTAMP Data Type: Example

In the slide, example A shows data from the hire_date column of the EMP4 table when the data type of the column is DATE. In example B, the table is altered and the data type of the hire_date column is changed to TIMESTAMP. The output shows the differences in display. You can convert from DATE to TIMESTAMP when the column has data, but you cannot convert from DATE or TIMESTAMP to TIMESTAMP WITH TIME ZONE unless the column is empty.

You can specify fractional seconds precision for a time stamp. If none are specified, as in this example, it defaults to 6.

For example, the following statement sets the fractional seconds precision as 7:

```
ALTER TABLE emp4
MODIFY hire_date TIMESTAMP(7);
```

Note: The Oracle DATE data type, by default, looks like what is shown in the example in the slide. However, the date data type also contains additional information such as hours, minutes, seconds, AM, and PM. To obtain the date in this format, you can apply a format mask or a function to the date value.

Comparing TIMESTAMP Data Types

```
CREATE TABLE web_orders
(order_date TIMESTAMP WITH TIME ZONE,
 delivery_time TIMESTAMP WITH LOCAL TIME ZONE);
```

```
INSERT INTO web_orders values
(current_date, current_timestamp + 2);
```

```
SELECT * FROM web_orders;
```

	ORDER_DATE	DELIVERY_TIME
1	22-MAR-16 05.55.10.000000000 AM -05:00	24-MAR-16 05.55.10.000000000 AM

ORACLE

In the example in the slide, a new table `web_orders` is created with a column of data type `TIMESTAMP WITH TIME ZONE` and a column of data type `TIMESTAMP WITH LOCAL TIME ZONE`. This table is populated whenever a `web_order` is placed. The time stamp and time zone for the user placing the order are inserted based on the `CURRENT_DATE` value. The local time stamp and time zone are populated by inserting two days from the `CURRENT_TIMESTAMP` value into it every time an order is placed. When a web-based company guarantees shipping, it can estimate its delivery time based on the time zone of the person placing the order.

Note: The code example output may vary as per the time of run of the command.

Lesson Agenda

- CURRENT_DATE, CURRENT_TIMESTAMP, and LOCALTIMESTAMP
- INTERVAL data types
- Using the following functions:
  - EXTRACT
  - TZ_OFFSET
  - FROM_TZ
  - TO_TIMESTAMP
  - TO_YMINTERVAL
  - TO_DSINTERVAL

INTERVAL Data Types

- INTERVAL data types are used to store the difference between two datetime values.
- There are two classes of intervals:
  - Year-month
  - Day-time
- The precision of the interval is:
  - The actual subset of fields that constitutes an interval
  - Specified in the interval qualifier

Data Type	Fields
INTERVAL YEAR TO MONTH	Year, Month
INTERVAL DAY TO SECOND	Days, Hour, Minute, Second with fractional seconds

ORACLE

INTERVAL data types are used to store the difference between two datetime values. There are two classes of intervals: year-month intervals and day-time intervals. A year-month interval is made up of a contiguous subset of fields of YEAR and MONTH, whereas a day-time interval is made up of a contiguous subset of fields consisting of DAY, HOUR, MINUTE, and SECOND. The actual subset of fields that constitute an interval is called the precision of the interval and is specified in the interval qualifier. Because the number of days in a year is calendar-dependent, the year-month interval is NLS-dependent, whereas day-time interval is NLS-independent.

The interval qualifier may also specify the leading field precision, which is the number of digits in the leading or only field, and in case the trailing field is SECOND, it may also specify the fractional seconds precision, which is the number of digits in the fractional part of the SECOND value. If not specified, the default value for leading field precision is 2 digits, and the default value for fractional seconds precision is 6 digits.

INTERVAL YEAR (year_precision) TO MONTH

This data type stores a period of time in years and months, where year_precision is the number of digits in the YEAR datetime field. The accepted values are 0 through 9. The default is 6.

INTERVAL DAY (day_precision) TO SECOND (fractional_seconds_precision)

This data type stores a period of time in days, hours, minutes, and seconds, where day_precision is the maximum number of digits in the DAY datetime field (accepted values are 0 through 9; the default is 2), and fractional_seconds_precision is the number of digits in the fractional part of the SECOND field. The accepted values are 0 through 9. The default is 6.

INTERVAL Fields

INTERVAL Field	Valid Values for Interval
YEAR	Any positive or negative integer
MONTH	00 to 11
DAY	Any positive or negative integer
HOUR	00 to 23
MINUTE	00 to 59
SECOND	00 to 59.9(N) where 9(N) is precision

ORACLE

INTERVAL YEAR TO MONTH can have fields of YEAR and MONTH.

INTERVAL DAY TO SECOND can have fields of DAY, HOUR, MINUTE, and SECOND.

The actual subset of fields that constitute an item of either type of interval is defined by an interval qualifier, and this subset is known as the precision of the item.

Year-month intervals are mutually comparable and assignable only with other year-month intervals, and day-time intervals are mutually comparable and assignable only with other day-time intervals.

INTERVAL YEAR TO MONTH: Example

```
CREATE TABLE warranty
(prod_id number,  warranty_time INTERVAL YEAR(3) TO
MONTH);
INSERT INTO warranty VALUES (123, INTERVAL '8' MONTH);
INSERT INTO warranty VALUES (155, INTERVAL '200'
YEAR(3));
INSERT INTO warranty VALUES (678, '200-11');
SELECT * FROM warranty;
```

	PROD_ID	WARRANTY_TIME
1	123	0-8
2	155	200-0
3	678	200-11

ORACLE

`INTERVAL YEAR TO MONTH` stores a period of time by using the `YEAR` and `MONTH` datetime fields. Specify `INTERVAL YEAR TO MONTH` as follows:

```
INTERVAL YEAR [(year_precision)] TO MONTH
```

where `year_precision` is the number of digits in the `YEAR` datetime field. The default value of `year_precision` is 2.

Restriction: The leading field must be more significant than the trailing field. For example, `INTERVAL '0-1' MONTH TO YEAR` is not valid.

Examples

- `INTERVAL '123-2' YEAR(3) TO MONTH`

Indicates an interval of 123 years, 2 months

- `INTERVAL '123' YEAR(3)`

Indicates an interval of 123 years, 0 months

- `INTERVAL '300' MONTH(3)`

Indicates an interval of 300 months

- `INTERVAL '123' YEAR`

Returns an error because the default precision is 2, and '123' has 3

The Oracle database supports two interval data types: INTERVAL YEAR TO MONTH and INTERVAL DAY TO SECOND; the column type, PL/SQL argument, variable, and return type must be one of the two. However, for interval literals, the system recognizes other American National Standards Institute (ANSI) interval types such as INTERVAL '2' YEAR or INTERVAL '10' HOUR. In these cases, each interval is converted to one of the two supported types.

In the example in the slide, a WARRANTY table is created, which contains a warranty_time column that takes the INTERVAL YEAR(3) TO MONTH data type. Different values are inserted into it to indicate years and months for various products. When these rows are retrieved from the table, you see a year value separated from the month value by a (-).

INTERVAL DAY TO SECOND Data Type: Example

```
CREATE TABLE lab
( exp_id number, test_time INTERVAL DAY(2) TO SECOND);

INSERT INTO lab VALUES (100012, '90 00:00:00');
INSERT INTO lab VALUES (56098,
  INTERVAL '6 03:30:16' DAY  TO SECOND);
```

```
SELECT * FROM lab;
```

	EXP_ID	TEST_TIME
1	100012	90 0:0:0.0
2	56098	6 3:30:16.0

ORACLE

In the example in the slide, you create the `lab` table with a `test_time` column of the `INTERVAL DAY TO SECOND` data type. You then insert into it the value `'90 00:00:00'` to indicate 90 days and 0 hours, 0 minutes, and 0 seconds, and `INTERVAL '6 03:30:16' DAY TO SECOND` to indicate 6 days, 3 hours, 30 minutes, and 16 seconds. The `SELECT` statement shows how this data is displayed in the database.

Lesson Agenda

- CURRENT_DATE, CURRENT_TIMESTAMP, and LOCALTIMESTAMP
- INTERVAL data types
- Using the following functions:
  - EXTRACT
  - TZ_OFFSET
  - FROM_TZ
  - TO_TIMESTAMP
  - TO_YMINTERVAL
  - TO_DSINTERVAL

ORACLE

EXTRACT

- Display all employees who were hired after 2010.

```
SELECT last_name, employee_id, hire_date
  FROM employees
  WHERE EXTRACT(YEAR FROM TO_DATE(hire_date, 'DD-MON-RR')) > 2010
  ORDER BY hire_date;
```

- Display the MONTH component from HIRE_DATE for those employees whose MANAGER_ID is 100.

```
        SELECT last_name, hire_date,
               EXTRACT (MONTH FROM HIRE_DATE)
        FROM employees
        WHERE manager_id = 100;
```

	LAST_NAME	HIRE_DATE	EXTRACT(MONTHFROMHIRE_DATE)
1	Kochhar	21-SEP-13	9
2	De Haan	13-JAN-09	1
3	Mourgos	16-NOV-15	11
4	Zlotkey	29-JAN-16	1
5	Hartstein	17-FEB-12	2

ORACLE®

The EXTRACT expression extracts and returns the value of a specified datetime field from a datetime or interval value expression. You can extract any of the components mentioned in the following syntax by using the EXTRACT function. The syntax of the EXTRACT function is:

```
SELECT EXTRACT( { YEAR | MONTH | DAY | HOUR | MINUTE | SECOND
  | TIMEZONE_HOUR
  | TIMEZONE_MINUTE
  | TIMEZONE_REGION
  | TIMEZONE_ABBR }
 FROM { expr } )
```

When you extract a TIMEZONE_REGION or TIMEZONE_ABBR (abbreviation), the value returned is a string containing the appropriate time zone name or abbreviation. When you extract any of the other values, the value returned is a date in the Gregorian calendar. When extracting from a datetime with a time zone value, the value returned is in UTC.

In the first example in the slide, the EXTRACT function is used to select all employees who were hired after 2010. In the second example in the slide, the EXTRACT function is used to extract MONTH from the HIRE_DATE column of the EMPLOYEES table for those employees who report to the manager whose EMPLOYEE_ID is 100.

TZ_OFFSET

Display the time zone offset for the 'US/Eastern', 'Canada/Yukon', and 'Europe/London' time zones:

```
SELECT TZ_OFFSET('US/Eastern'),
       TZ_OFFSET('Canada/Yukon'),
       TZ_OFFSET('Europe/London')
 FROM DUAL;
```

	TZ_OFFSET('US/EASTERN')	TZ_OFFSET('CANADA/YUKON')	TZ_OFFSET('EUROPE/LONDON')
1	-04:00	-07:00	+01:00

The TZ_OFFSET function returns the time zone offset corresponding to the value entered. The return value is dependent on the date when the statement is executed. For example, if the TZ_OFFSET function returns a value –08:00, this value indicates that the time zone where the command was executed is eight hours behind UTC. You can enter a valid time zone name, a time zone offset from UTC (which simply returns itself), or the keyword SESSIONTIMEZONE or DBTIMEZONE. The syntax of the TZ_OFFSET function is:

```
TZ_OFFSET({ 'time_zone_name'| '{ + | - } hh : mi'
          | SESSIONTIMEZONE
          | DBTMEZONE
          })
```

The Fold Motor Company has its headquarters in Michigan, USA, which is in the US/Eastern time zone. The company president, Mr. Fold, wants to have a conference call with the vice president of Canadian operations and the vice president of European operations, who are in the Canada/Yukon and Europe/London time zones, respectively. Mr. Fold wants to know the time in each of these places to make sure that his senior management will be available to attend the meeting. His secretary, Mr. Scott, helps by issuing the queries shown in the example and gets the following results:

- The 'US/Eastern' time zone is four hours behind UTC.
- The 'Canada/Yukon' time zone is seven hours behind UTC.
- The 'Europe/London' time zone is one hour ahead of UTC.

For a listing of valid time zone name values, you can query the `V$TIMEZONE_NAMES` dynamic performance view.

```
SELECT * FROM V$TIMEZONE_NAMES;
```

	TZNAME	TZABBREV	CON_ID
1	Africa/Abidjan	LMT	0
2	Africa/Abidjan	GMT	0
3	Africa/Accra	LMT	0
4	Africa/Accra	GMT	0
5	Africa/Accra	GHST	0

...

FROM_TZ

Display the TIMESTAMP value '2000-07-12 08:00:00' as a TIMESTAMP WITH TIME ZONE value for the 'Australia/North' time zone region.

```
SELECT FROM_TZ(TIMESTAMP
       '2000-07-12 08:00:00', 'Australia/North')
FROM DUAL;
```

	FROM_TZ(TIMESTAMP'2000-07-1208:00:00','AUSTRALIA/NORTH')
1	12-JUL-00 08.00.00.000000000 AM AUSTRALIA/NORTH

The FROM_TZ function converts a TIMESTAMP value to a TIMESTAMP WITH TIME ZONE value.

The syntax of the FROM_TZ function is as follows:

FROM_TZ(timestamp_value, time_zone_value)

where time_zone_value is a character string in the format 'TZH:TZM' or a character expression that returns a string in TZR (time zone region) with an optional TZD format. TZD is an abbreviated time zone string with daylight saving information. TZR represents the time zone region in datetime input strings. Examples are 'Australia/North', 'PST' for US/Pacific standard time, 'PDT' for US/Pacific daylight time, and so on.

The example in the slide converts a TIMESTAMP value to TIMESTAMP WITH TIME ZONE.

Note: To see a listing of valid values for the TZR and TZD format elements, query the V$TIMEZONE_NAMES dynamic performance view.

TO_TIMESTAMP

Display the character string '2016-03-06 11:00:00'
as a TIMESTAMP value:

```
SELECT TO_TIMESTAMP ('2016-03-06 11:00:00',
                     'YYYY-MM-DD HH:MI:SS')
FROM DUAL;
```

	TO_TIMESTAMP('2016-03-0611:00:00','YYYY-MM-DDHH:MI:SS')
1	06-MAR-16 11.00.00.000000000 AM

ORACLE

The TO_TIMESTAMP function converts a string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type to a value of the TIMESTAMP data type. The syntax of the TO_TIMESTAMP function is:

```
TO_TIMESTAMP(char [, fmt [, 'nlsparam' ] ])
```

The optional fmt specifies the format of char. If you omit fmt, the string must be in the default format of the TIMESTAMP data type. The optional nlsparam specifies the language in which month and day names, and abbreviations, are returned. The argument can have the following form:

```
'NLS_DATE_LANGUAGE = language'
```

If you omit nlsparams, the function uses the default date language for your session.

The example in the slide converts a character string to a value of TIMESTAMP.

Note: You use the TO_TIMESTAMP_TZ function to convert a string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type to a value of the TIMESTAMP WITH TIME ZONE data type. For more information about this function, see *Oracle Database SQL Language Reference* for Oracle Database 12*c*.

TO_YMINTERVAL

Display a date that is one year and two months after the hire date for employees working in the department with DEPARTMENT_ID 20.

```
SELECT hire_date,
       hire_date + TO_YMINTERVAL('01-02') AS
       HIRE_DATE_YMININTERVAL
FROM   employees
WHERE department_id = 20;
```

	HIRE_DATE	HIRE_DATE_YMININTERVAL
1	17-FEB-12	17-APR-13
2	17-AUG-13	17-OCT-14

ORACLE

The TO_YMINTERVAL function converts a character string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type to an INTERVAL YEAR TO MONTH data type. The INTERVAL YEAR TO MONTH data type stores a period of time by using the YEAR and MONTH datetime fields. The format of INTERVAL YEAR TO MONTH is as follows:

```
INTERVAL YEAR [(year_precision)] TO MONTH
```

where year_precision is the number of digits in the YEAR datetime field. The default value of year_precision is 2.

The syntax of the TO_YMINTERVAL function is:

```
TO_YMINTERVAL (char)
```

where char is the character string to be converted.

The example in the slide calculates a date that is one year and two months after the hire date for employees working in department 20 of the EMPLOYEES table.

TO_DSINTERVAL

Display a date that is 100 days and 10 hours after the hire date for all employees.

```
SELECT last_name,
 TO_CHAR(hire_date, 'mm-dd-yy:hh:mi:ss') hire_date,
  TO_CHAR(hire_date +
   TO_DSINTERVAL('100 10:00:00'),
     'mm-dd-yy:hh:mi:ss') hiredate2
FROM employees;
```

	LAST_NAME	HIRE_DATE	HIREDATE2
1	King	06-17-11:12:00:00	09-25-11:10:00:00
2	Kochhar	09-21-13:12:00:00	12-30-13:10:00:00
3	De Haan	01-13-09:12:00:00	04-23-09:10:00:00
4	Hunold	01-03-14:12:00:00	04-13-14:10:00:00
5	Ernst	05-21-15:12:00:00	08-29-15:10:00:00
6	Lorentz	02-07-15:12:00:00	05-18-15:10:00:00
7	Mourgos	11-16-15:12:00:00	02-24-16:10:00:00
8	Rajs	10-17-11:12:00:00	01-25-12:10:00:00
9	Davies	01-29-13:12:00:00	05-09-13:10:00:00
10	Matos	03-15-14:12:00:00	06-23-14:10:00:00

…

ORACLE

`TO_DSINTERVAL` converts a character string of the `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2` data type to an `INTERVAL DAY TO SECOND` data type.

In the example in the slide, the date 100 days and 10 hours after the hire date is obtained.

Daylight Saving Time (DST)

- Start of Daylight Saving:
  - Time jumps from 01:59:59 AM to 03:00:00 AM.
  - Values from 02:00:00 AM to 02:59:59 AM are not valid.
- End of Daylight Saving:
  - Time jumps from 02:00:00 AM to 01:00:01 AM.
  - Values from 01:00:01 AM to 02:00:00 AM are ambiguous because they are visited twice.

ORACLE

Most western nations advance the clock ahead by one hour during the summer months. This period is called daylight saving time. Daylight saving time lasts from the start of Daylight Saving to the end of Daylight Saving in most of the United States, Mexico, and Canada. The nations of the European Union observe daylight saving time, but they call it the summer time period. Europe's summer time period begins a week earlier than its North American counterpart, but ends at the same time.

The Oracle database automatically determines, for any given time zone region, whether daylight saving time is in effect and returns local time values accordingly. The datetime value is sufficient for the Oracle database to determine whether daylight saving time is in effect for a given region in all cases except boundary cases. A boundary case occurs during the period when daylight saving time goes into or out of effect. For example, in the US/Eastern region, when daylight saving time goes into effect, the time changes from 01:59:59 AM to 03:00:00 AM. The one-hour interval between 02:00:00 AM and 02:59:59 AM. does not exist. When daylight saving time goes out of effect, the time changes from 02:00:00 AM back to 01:00:01 AM, and the one-hour interval between 01:00:01 AM and 02:00:00 AM is repeated.

ERROR_ON_OVERLAP_TIME

ERROR_ON_OVERLAP_TIME is a session parameter to notify the system to issue an error when it encounters a datetime that occurs in the overlapped period and no time zone abbreviation was specified to distinguish the period.

For example, daylight saving time ends on October 31, at 02:00:01 AM. The overlapped periods are:

- 10/31/2016 01:00:01 AM to 10/31/2016 02:00:00 AM (EDT)
- 10/31/2016 01:00:01 AM to 10/31/2016 02:00:00 AM (EST)

If you input a datetime string that occurs in one of these two periods, you need to specify the time zone abbreviation (for example, EDT or EST) in the input string for the system to determine the period. Without this time zone abbreviation, the system does the following:

If the ERROR_ON_OVERLAP_TIME parameter is FALSE, it assumes that the input time is standard time (for example, EST). Otherwise, an error is raised.

Quiz

The `TIME_ZONE` session parameter may be set to:

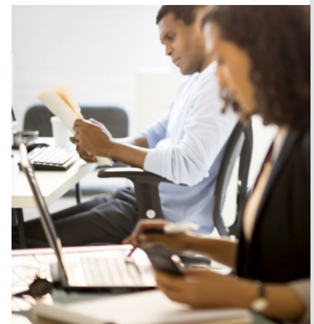
a. A relative offset
b. Database time zone
c. OS local time zone
d. A named region

Answer: b, c, d

Summary

In this lesson, you should have learned how to:

- Use data types similar to `DATE` that store fractional seconds and track time zones
- Use data types that store the difference between two datetime values
- Use the following datetime functions:
  - `CURRENT_DATE`
  - `CURRENT_TIMESTAMP`
  - `LOCALTIMESTAMP`
  - `DBTIMEZONE`
  - `SESSIONTIMEZONE`
  - `EXTRACT`
  - `TZ_OFFSET`
  - `FROM_TZ`
  - `TO_TIMESTAMP`
  - `TO_YMINTERVAL`
  - `TO_DSINTERVAL`



This lesson addressed some of the datetime functions available in the Oracle database.