

I Introduction

ORACLE®

Copyright © Oracle Corporation, 2001. All rights reserved.

Schedule:

Timing

60 minutes

60 minutes

Topic

Lecture

Total

Objectives

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

- List the features of Oracle9i
- Discuss the theoretical and physical aspects of a relational database
- Describe the Oracle implementation of the RDBMS and ORDBMS

ORACLE

I-2

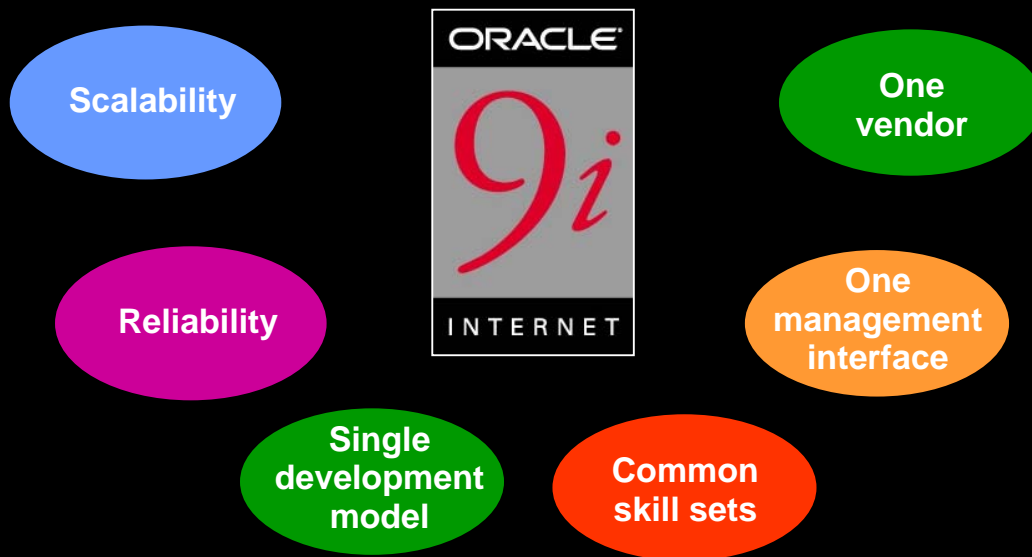
Copyright © Oracle Corporation, 2001. All rights reserved.

Lesson Aim

In this lesson, you gain an understanding of the **relational database management system (RDBMS)** and the **object relational database management system (ORDBMS)**. You are also introduced to the following:

- **SQL** statements that are specific to Oracle
- **iSQL*Plus**, which is used for executing SQL and for formatting and reporting purposes

Oracle9i



ORACLE

I-3

Copyright © Oracle Corporation, 2001. All rights reserved.

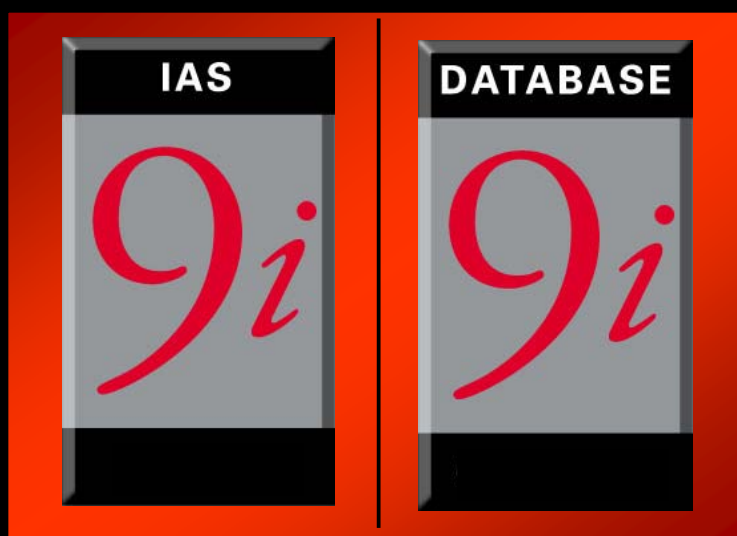
Oracle9i Features

Oracle offers a comprehensive high-performance infrastructure for **e-business**. It is called Oracle9i. Oracle9i includes everything needed to develop, deploy, and manage Internet applications.

Benefits include:

- Scalability from departments to enterprise e-business sites
- Robust, reliable, available, secure architecture
- One development model, easy deployment options
- Leverage an organization's current skillset throughout the Oracle platform (including SQL, PL/SQL, Java, and XML)
- One management interface for all applications
- Industry standard technologies, no proprietary lock-in

Oracle9i



ORACLE

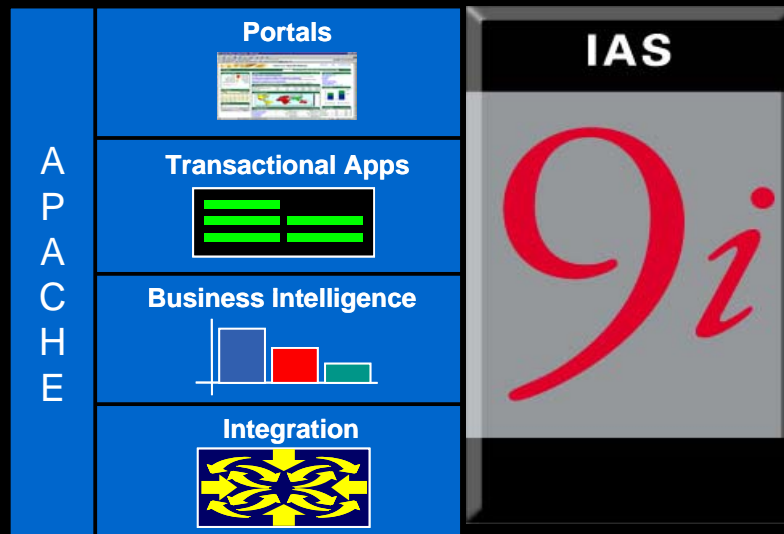
I-4

Copyright © Oracle Corporation, 2001. All rights reserved.

Oracle9i

There are two products, **Oracle9i Application Server** and **Oracle9i Database**, that provide a complete and simple infrastructure for Internet applications.

Oracle9i Application Server



ORACLE

I-5

Copyright © Oracle Corporation, 2001. All rights reserved.

Oracle9i Application Server

The **Oracle9i Application Server** (Oracle9iAS) runs all your applications. The **Oracle9i Database** stores all your data.

Oracle9i Application Server is the only application server to include services for all the different server applications you will want to run. Oracle9iAS can run your:

- Portals or Web sites
- Java transactional applications
- Business intelligence applications

It also provides integration between users, applications, and data throughout your organization.

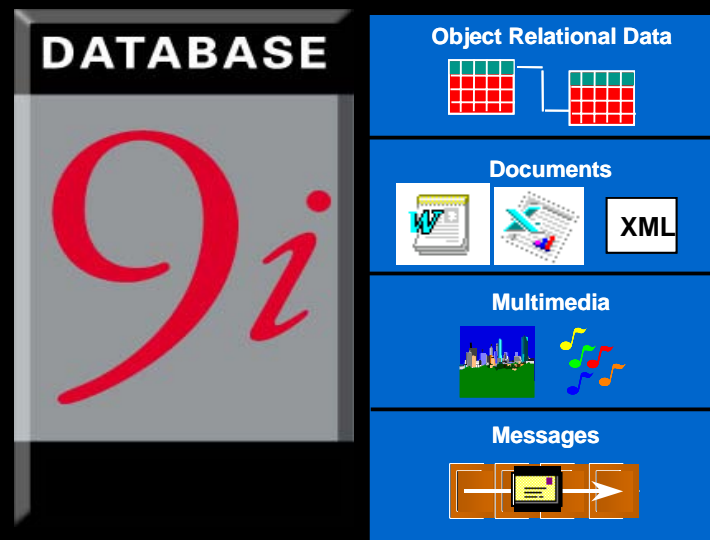
Instructor Note

Apache is used to maintain an open-source HTTP server software product for various modern desktop and server operating systems. It provides a secure, efficient and extensible server with HTTP services in synchronization with the current HTTP standards.

Oracle9iAS is powered by Apache, which is the de facto industry standard.

Introduction to Oracle9i: SQL I-5

Oracle9i Database



ORACLE

I-6

Copyright © Oracle Corporation, 2001. All rights reserved.

Oracle9i Database

The roles of the two products are very straightforward. **Oracle9i Database** manages all your data. This is not just the object relational data that you expect an enterprise database to manage. It can also be unstructured data like:

- Spreadsheets
- Word documents
- PowerPoint presentations
- XML
- Multimedia data types like MP3, graphics, video, and more

The data does not even have to be in the database. Oracle9i Database has services through which you can store metadata about information stored in file systems. You can use the database server to manage and serve information wherever it is located.

Instructor Note

XML (the Extensible Markup Language) was first ratified by the W3C (World Wide Web Consortium) as the standard for information exchange on the Internet in February 1998. Since then it has been rapidly gaining momentum as the development community has begun to appreciate its potential and as vendors have started to deliver tools to support it. XML specifies a rigorous, text-based way to represent the structure inherent in data so that it can be authored and interpreted unambiguously.

Relational and Object Relational Database Management System

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

ORACLE

I-7

Copyright © Oracle Corporation, 2001. All rights reserved.

About the Oracle Server

The Oracle9i server supports both the relational and object relation models.

The Oracle server extends the data modeling capabilities to support an object relational database model that brings **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 **online transaction processing (OLTP)** applications, such as better sharing of run-time data structures, larger buffer caches, and deferrable constraints. **Data warehouse applications** will benefit from enhancements such as parallel execution of insert, update, and delete operations; partitioning; and parallel-aware query optimization. Operating within the Network Computing Architecture (NCA) framework, Oracle9i supports client-server and Web-based applications that are distributed and multitiered.

Oracle9i can scale tens of thousands of concurrent users, support up to 512 petabytes of data (a petabyte is 1,000 terabytes), and can handle any type of data, including text, spatial, image, sound, video, and time series as well as traditional structured data.

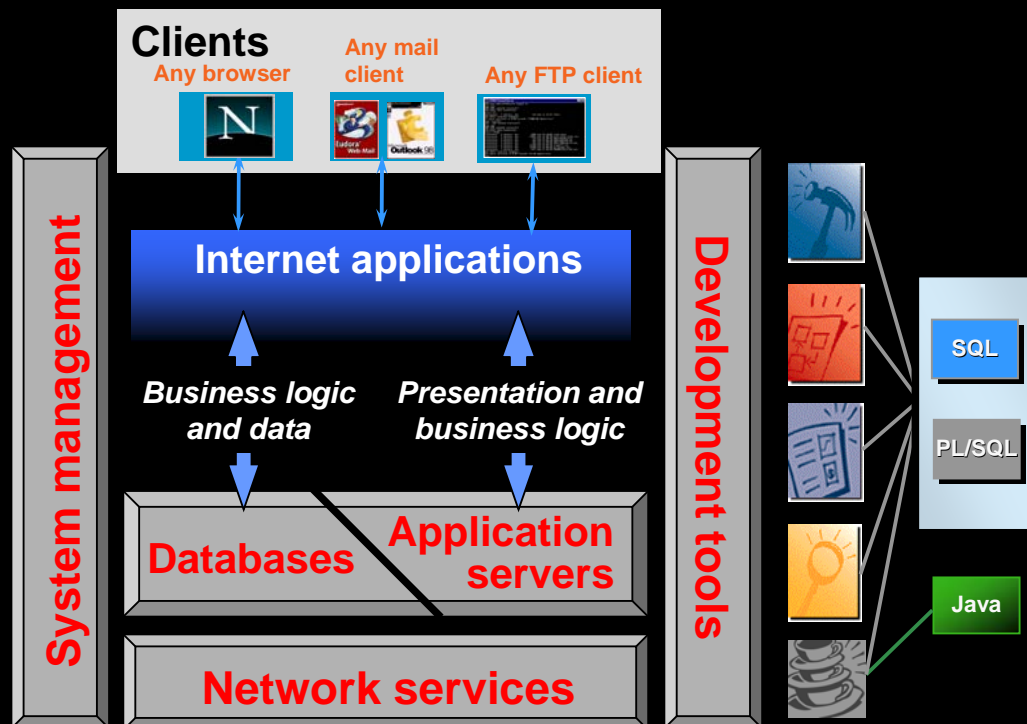
For more information, see *Oracle9i Concepts*.

Instructor Note

The Object Relation Database Management System features are available with release 8 and higher.

Oracle7 is a relational database management system and Oracle8, 8i, and 9i are object relational database management systems.

Oracle Internet Platform



ORACLE

I-8

Copyright © Oracle Corporation, 2001. All rights reserved.

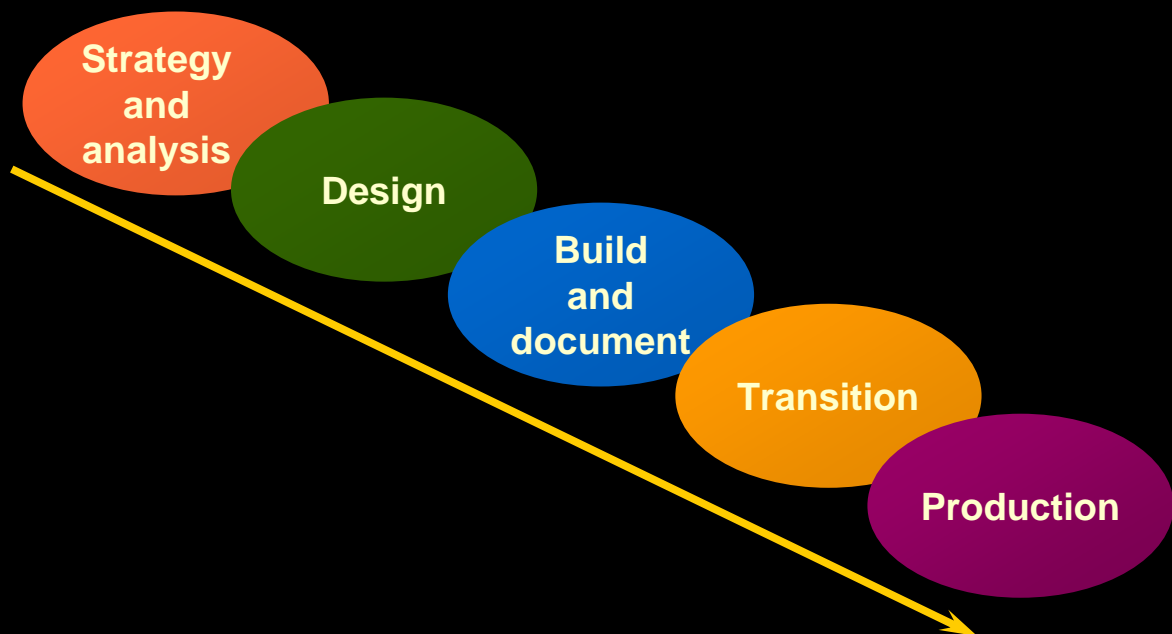
Oracle Internet Platform

Oracle offers a comprehensive high-performance Internet platform for **e-commerce** and data warehousing. This integrated platform includes everything needed to develop, deploy, and manage Internet applications. The Oracle Internet Platform is built on three core pieces:

- Browser-based clients to process presentation
- Application servers to execute business logic and serve presentation logic to browser-based clients
- Databases to execute database-intensive business logic and serve data

Oracle offers a wide variety of the most advanced graphical user interface (GUI) driven development tools to build business applications, as well as a large suite of software applications for many areas of business and industry. Stored procedures, functions, and packages can be written by using SQL, PL/SQL, or Java.

System Development Life Cycle



ORACLE

I-9

Copyright © Oracle Corporation, 2001. All rights reserved.

System Development Life Cycle

From concept to production, you can develop a database by using the **system development life cycle**, which contains multiple stages of development. This top-down, systematic approach to database development transforms business information requirements into an operational database.

Strategy and Analysis

- Study and analyze the business requirements. Interview users and managers to identify the information requirements. Incorporate the enterprise and application mission statements as well as any future system specifications.
- Build models of the system. Transfer the business narrative into a graphical representation of business information needs and rules. Confirm and refine the model with the analysts and experts.

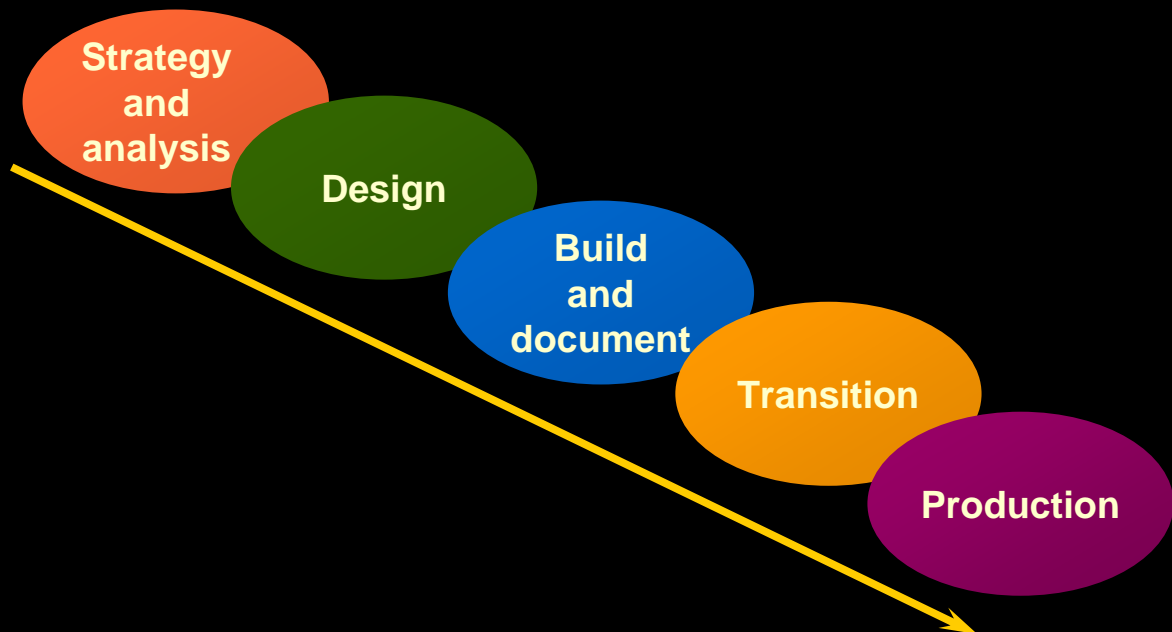
Design

Design the database based on the model developed in the strategy and analysis phase.

Build and Document

- Build the prototype system. Write and execute the commands to create the tables and supporting objects for the database.
- Develop user documentation, Help text, and operations manuals to support the use and operation of the system.

System Development Life Cycle



ORACLE

I-10

Copyright © Oracle Corporation, 2001. All rights reserved.

System Development Life Cycle (continued)

Transition

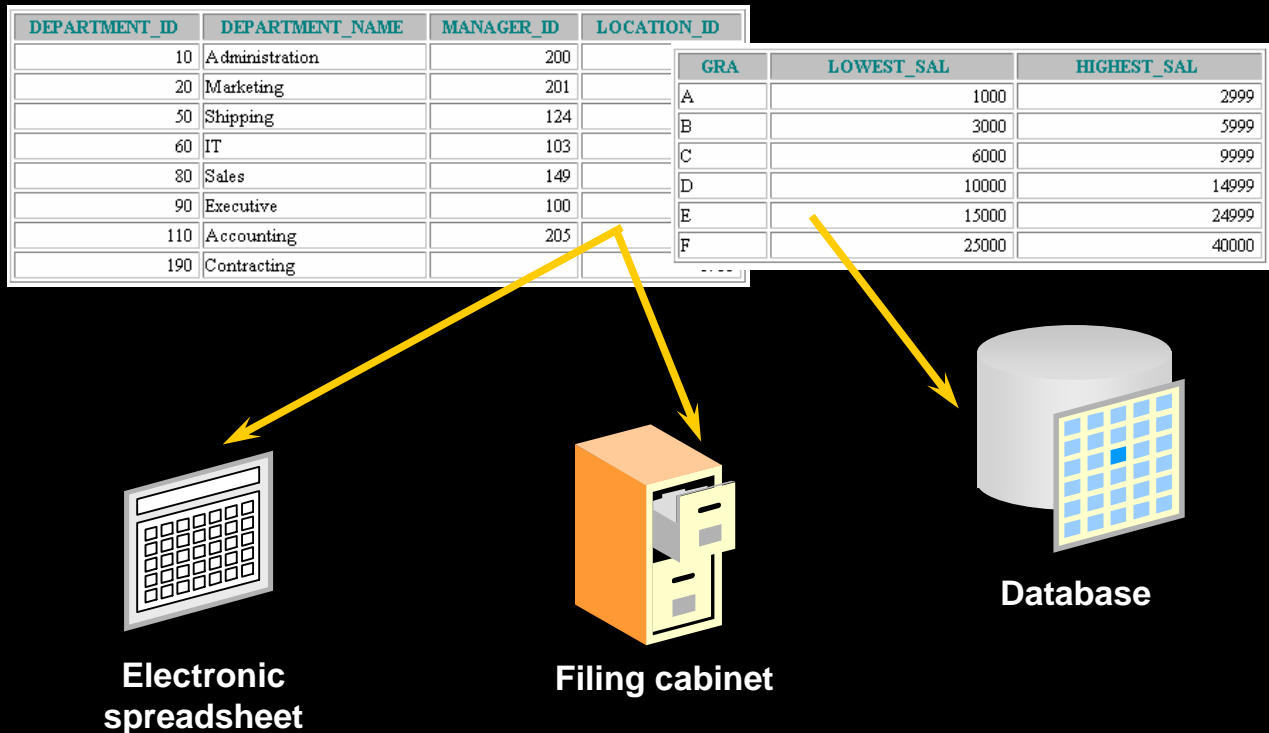
Refine the prototype. Move an application into production with user acceptance testing, conversion of existing data, and parallel operations. Make any modifications required.

Production

Roll out the system to the users. Operate the production system. Monitor its performance, and enhance and refine the system.

Note: The various phases of the system development life cycle can be carried out iteratively. This course focuses on the build phase of the system development life cycle.

Data Storage on Different Media



ORACLE

I-11

Copyright © Oracle Corporation, 2001. All rights reserved.

Storing Information

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 employees, departments, and salaries. These pieces of information are called *data*.

Organizations can store data on 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 database management systems (**DBMS**). A DBMS is a program that stores, retrieves, and modifies data in the database on request. There are four main types of databases: *hierarchical*, *network*, *relational*, and more recently *object relational*.

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

I-12

Copyright © Oracle Corporation, 2001. All rights reserved.

Relational Model

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

The more popular models used at that time were hierarchical and network, or even simple flat file data structures. **Relational database management systems** (RDBMS) soon became very popular, especially for their 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 products, providing a total solution.

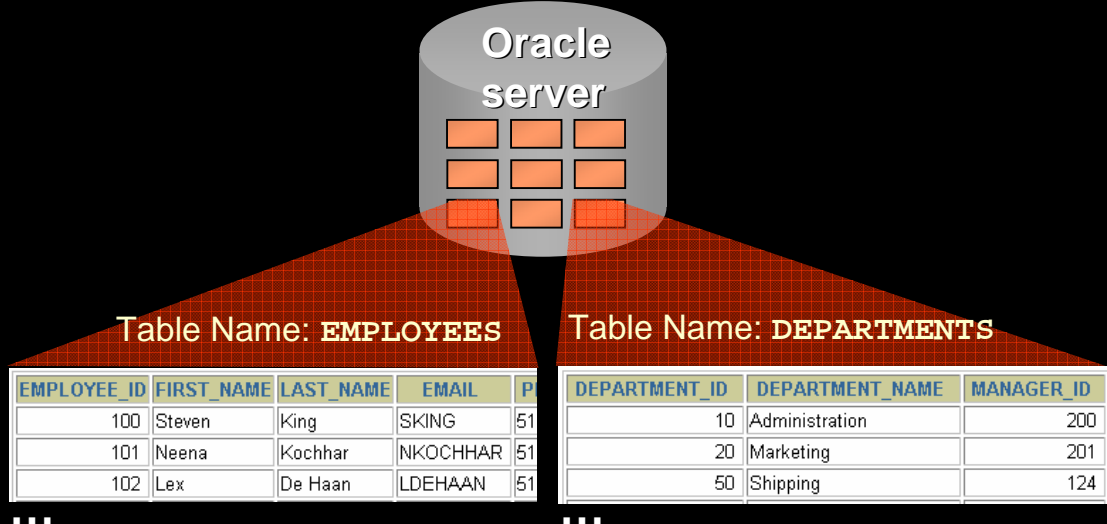
Components of the Relational Model

- 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

For more information, see E. F. Codd, *The Relational Model for Database Management Version 2* (Reading, Mass.: Addison-Wesley, 1990).

Definition of a Relational Database

A relational database is a collection of relations or two-dimensional tables.



ORACLE

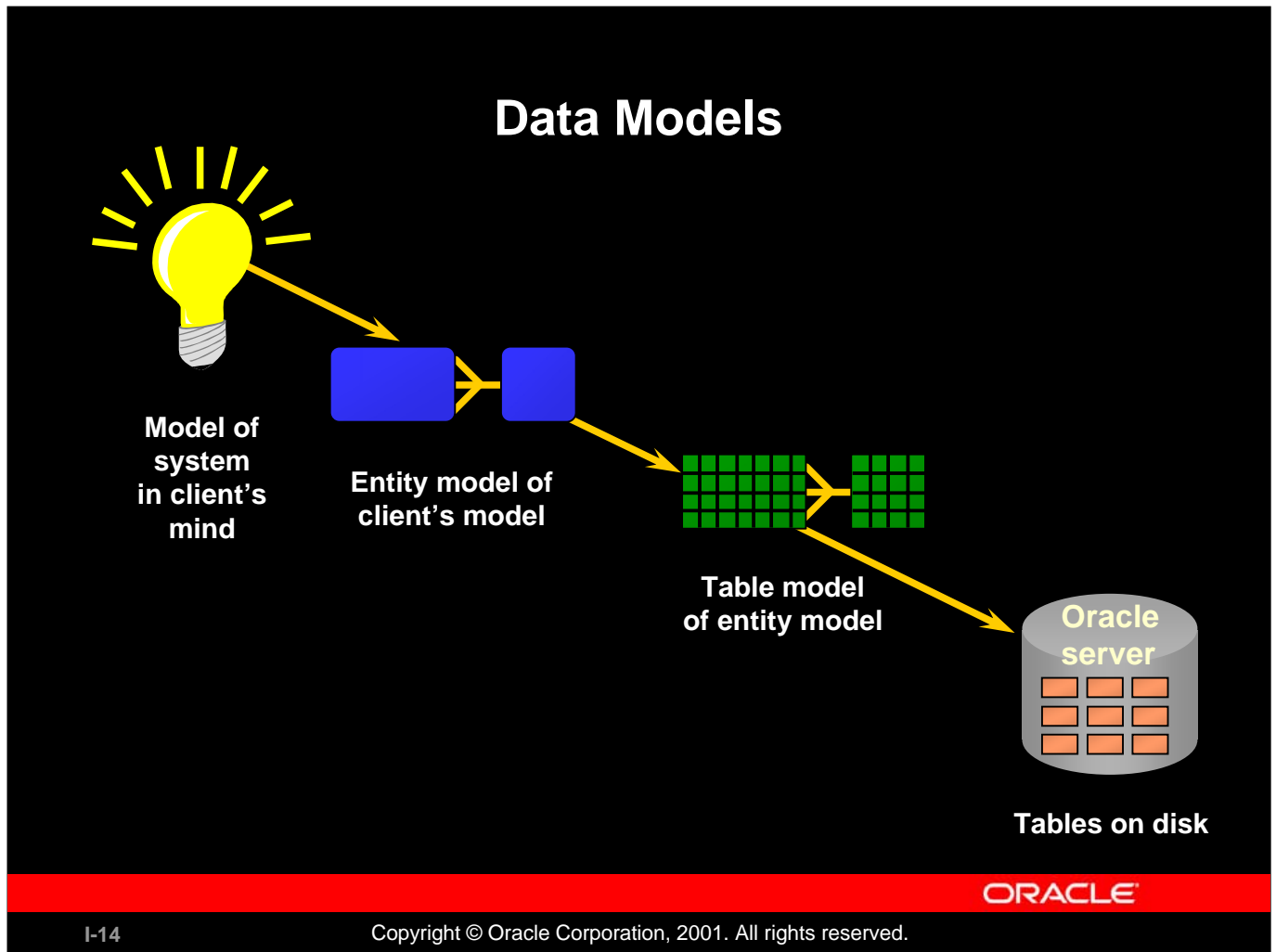
I-13

Copyright © Oracle Corporation, 2001. All rights reserved.

Definition of a Relational Database

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 a 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 the database design.

Purpose of Models

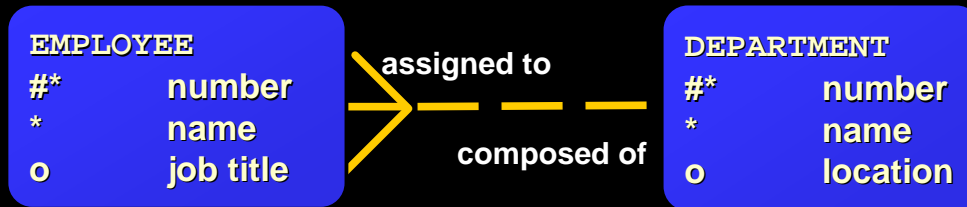
Models help communicate the concepts in people's minds. They can be used to do the following:

- 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

I-15

Copyright © Oracle Corporation, 2001. All rights reserved.

ER Modeling

In an effective system, data is divided into discrete categories or entities. An **entity relationship (ER)** model is an illustration of various entities in a business and the relationships between 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 a 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 the database design
- Offers an effective framework for integrating multiple applications

Key Components

- **Entity:** A thing of significance about which information needs to 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.

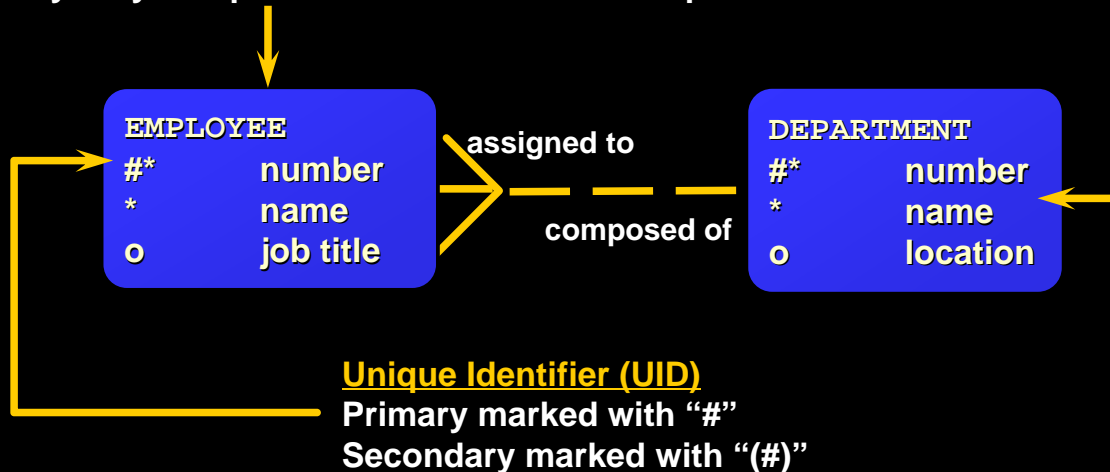
Entity Relationship Modeling Conventions

Entity

Soft box
Singular, unique name
Uppercase
Synonym in parentheses

Attribute

Singular name
Lowercase
Mandatory marked with “*”
Optional marked with “o”



ORACLE

I-16

Copyright © Oracle Corporation, 2001. All rights reserved.

ER Modeling (continued)

Entities

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

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

Attributes

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

- Use singular names in lowercase.
- Tag mandatory attributes, or values that must be known, with an asterisk: *.
- Tag optional attributes, or values that may be known, with the letter o.

Relationships

Symbol	Description
Dashed line	Optional element indicating “may be”
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”

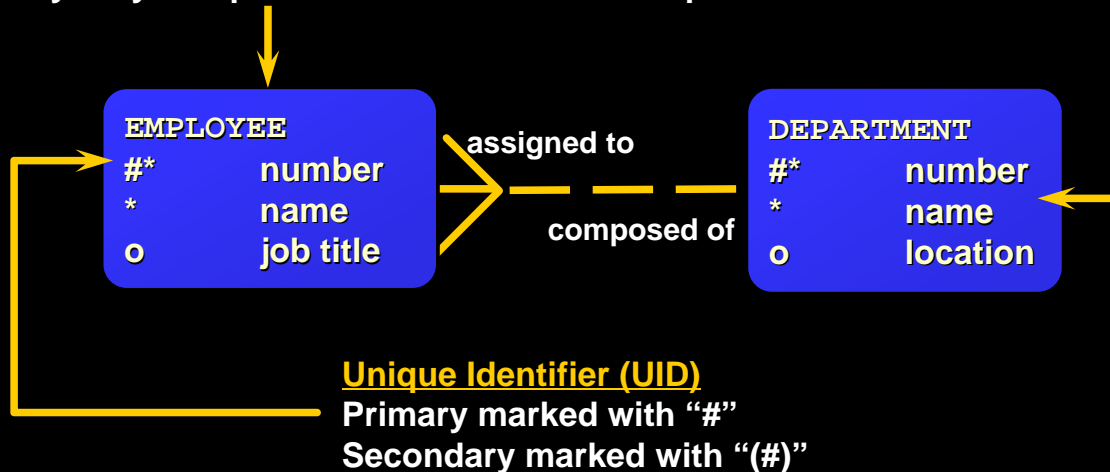
Entity Relationship Modeling Conventions

Entity

Soft box
Singular, unique name
Uppercase
Synonym in parentheses

Attribute

Singular name
Lowercase
Mandatory marked with “*”
Optional marked with “o”



ORACLE

I-17

Copyright © Oracle Corporation, 2001. All rights reserved.

ER Modeling (continued)

Relationships

Each direction of the relationship contains:

- A label, for example, *taught by* or *assigned to*
- An optionality, either *must be* or *may be*
- A degree, either *one and only one* or *one or more*

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

Each source entity {may be | must be} relationship name {one and only one | one or more} 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 number symbol: #
- Tag secondary UIDs with a number sign in parentheses: (#)

Relating Multiple Tables

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

Table Name: **EMPLOYEES**

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	DEPARTMENT_ID
174	Ellen	Abel	80
142	Curtis	Davies	50
102	Lex	De Haan	90
104	Bruce	Ernst	60
202	Pat	Fay	20
206	William	Gietz	110

...

Primary key

Foreign key

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		1700

Primary key

ORACLE

I-18

Copyright © Oracle Corporation, 2001. All rights reserved.

Relating Multiple Tables

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. 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 by using the foreign keys. A foreign key is a column or a set of columns that refer 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 to organize information in separate, manageable units. Employee data can be kept logically distinct from 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, or else be null.
- A foreign key must reference either a primary key or unique key column.

Relational Database Terminology

2			3		4	
	EMPLOYEE_ID	LAST_NAME	FIRST_NAME	SALARY	COMMISSION_PCT	DEPARTMENT_ID
	100	King	Steven	24000		90
	101	Kochhar	Neena	17000		90
	102	De Haan	Lex	17000		90
	103	Hunold	Alexander	9000		60
	104	Ernst	Bruce	6000		60
	107	Lorentz	Diana	4200		60
	124	Mourgos	Kevin	5800		50
	141	Rajs	Trenna	3500		50
	142	Davies	Curtis	3100		50
	143	Matos	Randall	2600		50
	144	Vargas	Peter	2500		50
	149	Zlotkey	Eleni	10500	.2	80
	174	Abel	Ellen	11000	.3	80
	176	Taylor	Jonathon	8600	.2	80
	178	Grant	Kimberely	7000	.15	
	200	Whalen	Jennifer	4400		10
1	201	Hartstein	Michael	13000		20
	202	Fay	Pat	6000		20
	205	Higgins	Shelley	12000		110
	206	Gietz	William	8300		110
				5		
						6

ORACLE

I-19

Copyright © Oracle Corporation, 2001. All rights reserved.

Terminology Used in a Relational Database

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 *table* representing all data required for a particular employee. Each row in a table should be identified by a primary key, which allows 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 the example, the salary 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 may have no value in it. This is called a *null value*. In the *EMPLOYEES* table, only employees who have a role of sales representative have a value in the *COMMISSION_PCT* (commission) field.
6. A *field* can be found at the intersection of a row and a column. There can be only one value in it.

Relational Database Properties

A relational database:

- Can be accessed and modified by executing structured query language (SQL) statements
- Contains a collection of tables with no physical pointers
- Uses a set of operators

ORACLE

I-20

Copyright © Oracle Corporation, 2001. All rights reserved.

Properties of a Relational Database

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.

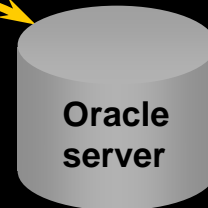
To access the database, you execute a **structured query language (SQL)** statement, which is the American National Standards Institute (ANSI) standard language for operating relational databases. The language contains a large set of operators for partitioning and combining relations. The database can be modified by using the SQL statements.

Communicating with a RDBMS Using SQL

SQL statement
is entered.

```
SELECT department_name  
FROM departments;
```

Statement is sent to
Oracle Server.



DEPARTMENT_NAME
Administration
Marketing
Shipping
IT
Sales
Executive
Accounting
Contracting

ORACLE

I-21

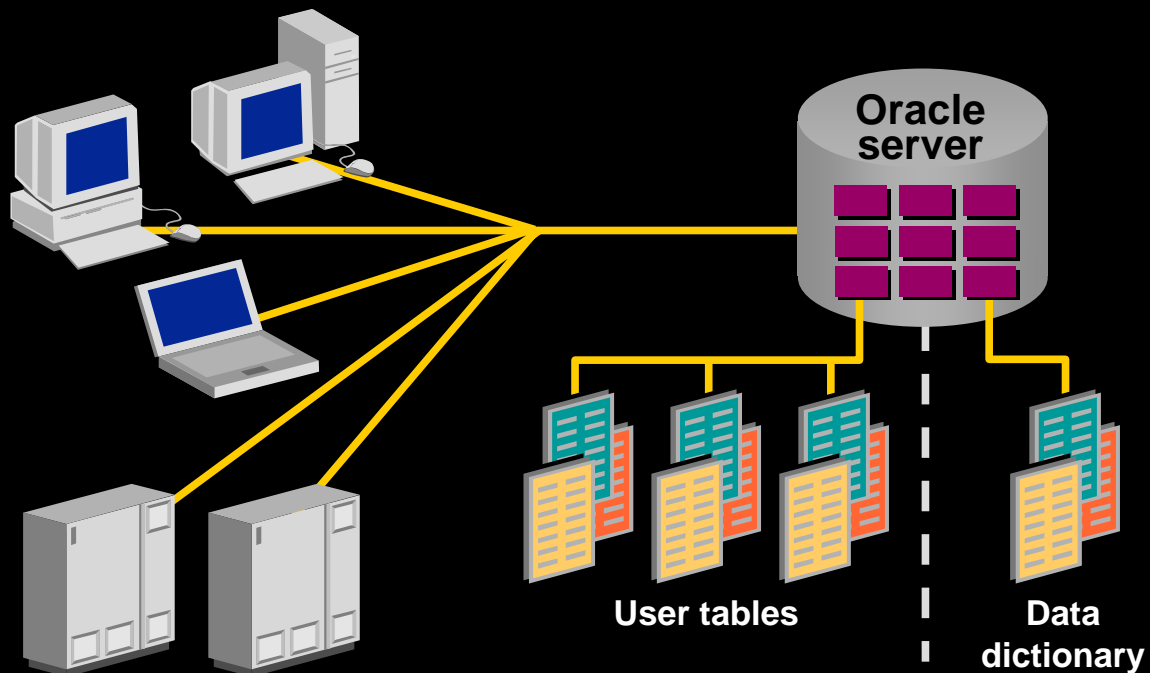
Copyright © Oracle Corporation, 2001. All rights reserved.

Structured Query Language

Using SQL, you can communicate with the Oracle server. SQL has the following advantages:

- Efficient
- Easy to learn and use
- Functionally complete (With SQL, you can define, retrieve, and manipulate data in the tables.)

Relational Database Management System



I-22

Copyright © Oracle Corporation, 2001. All rights reserved.

Relational Database Management System

Oracle provides a flexible **RDBMS** called Oracle9i. Using its features, you can store and manage data with all the advantages of a relational structure plus PL/SQL, an engine that provides you with the ability to store and execute program units. Oracle9i also supports **Java** and **XML**. The Oracle server offers the options of retrieving data based on optimization techniques. It includes security features that control how a database is accessed and used. Other features include consistency and protection of data through locking mechanisms.

The Oracle9i server provides an open, comprehensive, and integrated approach to information management. An Oracle server consists of an Oracle database and an Oracle server instance. Every time a database is started, a **system global area (SGA)** is allocated, and Oracle background processes are started. The system global area is an area of memory used for database information shared by the database users. The combination of the background processes and memory buffers is called an **Oracle instance**.

SQL Statements

SELECT	Data retrieval
INSERT UPDATE DELETE MERGE	Data manipulation language (DML)
CREATE ALTER DROP RENAME TRUNCATE	Data definition language (DDL)
COMMIT ROLLBACK SAVEPOINT	Transaction control
GRANT REVOKE	Data control language (DCL)

ORACLE

I-23

Copyright © Oracle Corporation, 2001. All rights reserved.

SQL Statements

Oracle SQL complies with industry-accepted standards. Oracle Corporation ensures future compliance with evolving standards by actively involving key personnel in SQL standards committees. Industry-accepted committees are the **American National Standards Institute (ANSI)** and the **International Standards Organization (ISO)**. Both ANSI and ISO have accepted SQL as the standard language for relational databases.

Statement	Description
SELECT	Retrieves data from the database
INSERT UPDATE DELETE MERGE	Enters new rows, changes existing rows, and removes unwanted rows from tables in the database, respectively. Collectively known as <i>data manipulation language (DML)</i> .
CREATE ALTER DROP RENAME TRUNCATE	Sets up, changes, and removes data structures from tables. Collectively known as <i>data definition language (DDL)</i> .
COMMIT ROLLBACK SAVEPOINT	Manages the changes made by DML statements. Changes to the data can be grouped together into logical transactions.
GRANT REVOKE	Gives or removes access rights to both the Oracle database and the structures within it. Collectively known as <i>data control language (DCL)</i> .

Tables Used in the Course

EMPLOYEES

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY
100	Steven	King	SKING	515.123.4567	17-JUN-87	AD_PRES	24000
101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-SEP-89	AD_VP	17000
102	Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-93	AD_VP	17000
103	Alexander	Hunold	AHUNOLD	590.423.4567	03-JAN-90	IT_PROG	9000
104	Bruce	Ernst	BERNST	590.423.4568	21-MAY-91	IT_PROG	6000
107	Diana	Lorentz	DLORENTZ	590.423.5567	07-FEB-99	IT_PROG	4200
124	Kevin	Mourgos	KMOURGOS	650.123.5234	16-NOV-99	ST_MAN	5800
141	Trenna	Rajs	TRAJS	650.121.8009	17-OCT-95	ST_CLERK	3500
142	Curtis	Davies	CDAVIES	650.121.2994	29-JAN-97	ST_CLERK	3100
143	Randall	Matos	RMATOS	650.121.2874	15-MAR-98	ST_CLERK	2600

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		1700

DEPARTMENTS

GRADE	LOWEST_SAL	HIGHEST_SAL
A	1000	2999
B	3000	5999
C	6000	9999
D	10000	14999
E	15000	24999
F	25000	40000

JOB_GRADES

ORACLE

Tables Used in the Course

The following main tables are used in this course:

- EMPLOYEES table, which gives details of all the employees
- DEPARTMENTS table, which gives details of all the departments
- JOB_GRADES table, which gives details of salaries for various grades

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

Instructor Note

This course uses the Sample Schema that is shipped with the Oracle9i database. The full Sample Schema consists of several subschemas. One of the subschemas is called HR (Human Resources). This course uses the HR portion of the Sample Schema. Some of the rows from the HR schema are removed to simplify examples in this courseware.

Have the students turn to Appendix B to review the tables used in the course.

Summary

- **The Oracle9i Server is the database for Internet computing.**
- **Oracle9i is based on the object relational database management system.**
- **Relational databases are composed of relations, managed by relational operations, and governed by data integrity constraints.**
- **With the Oracle Server, you can store and manage information by using the SQL language and PL/SQL engine.**

ORACLE

I-25

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary

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 relational database management system needs. The main products are the Oracle9i Database Server, with which you can store and manage information by using SQL, and the Oracle9i Application Server with which you can run all of your applications.

SQL

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

1

Writing Basic SQL SELECT Statements

ORACLE

Copyright © Oracle Corporation, 2001. All rights reserved.

Schedule:	Timing	Topic
	40 minutes	Lecture
	25 minutes	Practice
	65 minutes	Total

Objectives

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

- List the capabilities of SQL `SELECT` statements
- Execute a basic `SELECT` statement
- Differentiate between SQL statements and *iSQL*Plus* commands

ORACLE

1-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Lesson Aim

To extract data from the database, you need to use the **structured query language (SQL)** `SELECT` statement. You may need to restrict the columns that are displayed. This lesson describes all the SQL statements needed to perform these actions.

You may want to create `SELECT` statements that can be used more than once. This lesson also covers the *iSQL*Plus* environment where you execute SQL statements.

Note: *iSQL*Plus* is new in the Oracle9i product. It is a browser environment where you execute SQL commands. In prior releases of Oracle, `SQL*Plus` was the default environment where you executed SQL commands. `SQL*Plus` is still available and is described in Appendix C.

Capabilities of SQL `SELECT` Statements

Projection

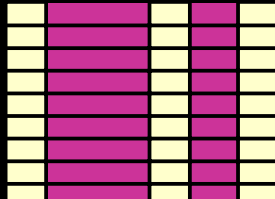


Table 1

Selection

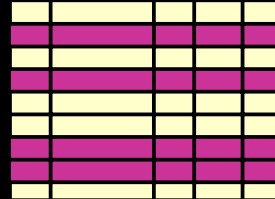


Table 1

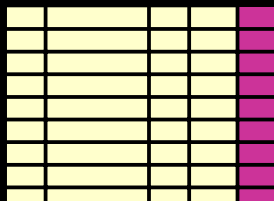


Table 1

Join

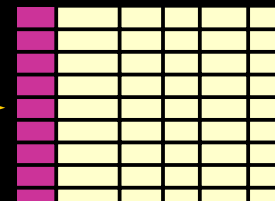


Table 2

ORACLE

Capabilities of SQL `SELECT` Statements

A `SELECT` statement retrieves information from the database. Using a `SELECT` statement, you can do the following:

- **Projection:** You can use the projection capability in SQL to choose the columns in a table that you want returned by your query. You can choose as few or as many columns of the table as you require.
- **Selection:** You can use the selection capability in SQL to choose the rows in a table that you want returned by a query. You can use various criteria to restrict the rows that you see.
- **Joining:** You can use the join capability in SQL to bring together data that is stored in different tables by creating a link between them. You learn more about joins in a later lesson.

Instructor Note

Inform students that selection and projection are often considered horizontal and vertical partitioning.

Basic SELECT Statement

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

- **SELECT** identifies *what* columns
- **FROM** identifies *which* table

ORACLE

1-4

Copyright © Oracle Corporation, 2001. All rights reserved.

Basic SELECT Statement

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 specifies the table containing the columns listed in the SELECT clause

In the syntax:

SELECT	is a list of one or more columns
*	selects all columns
DISTINCT	suppresses duplicates
column/expression	selects the named column or the expression
alias	gives selected columns different headings
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, ... is a clause.
- A **statement** is a combination of two or more clauses.
For example, SELECT * FROM employees is a SQL statement.

Introduction to Oracle9i: SQL 1-4

Selecting All Columns

```
SELECT *  
FROM 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		1700

8 rows selected.

ORACLE

1-5

Copyright © Oracle Corporation, 2001. All rights reserved.

Selecting All Columns of All Rows

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

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

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

Instructor Note

Let the students know that details of all the tables are given in Appendix B.

Selecting Specific Columns

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

DEPARTMENT_ID	LOCATION_ID
10	1700
20	1800
50	1500
60	1400
80	2500
90	1700
110	1700
190	1700

8 rows selected.

ORACLE

1-6

Copyright © Oracle Corporation, 2001. All rights reserved.

Selecting Specific Columns of All Rows

You can use the **SELECT statement** to display specific columns of the table by specifying the column names, separated by commas. The example on 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 going from left to right, you use the following statement:

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

LOCATION_ID	DEPARTMENT_ID
1700	10
1800	20
1500	50

■ ■ ■

8 rows selected.

Instructor Note

You can also select from pseudocolumns. A pseudocolumn behaves like a table column but is not actually stored in the table. You cannot insert or delete values of the pseudocolumns. Some available pseudocolumns are CURRVAL, NEXTVAL, LEVEL, ROWID, and ROWNUM.

Writing SQL Statements

- SQL statements are not case sensitive.
- SQL statements can be 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.

ORACLE

1-7

Copyright © Oracle Corporation, 2001. All rights reserved.

Writing SQL Statements

Using the following simple rules and guidelines, you can construct valid statements that are both easy to read and easy to 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, are entered in lowercase.

Executing SQL Statements

Using **iSQL*Plus**, click the **Execute button** to run the command or commands in the editing window.

Instructor Note

Although not required in **iSQL*Plus**, placing a semicolon (;) at the end of the last clause is recommended. Other environments, such as PL/SQL programs, require that the end of each statement contains a semicolon.

Column Heading Defaults

- **iSQL*Plus:**
 - Default heading justification: Center
 - Default heading display: Uppercase
- **SQL*Plus:**
 - Character and Date column headings are left-justified
 - Number column headings are right-justified
 - Default heading display: Uppercase

ORACLE

1-8

Copyright © Oracle Corporation, 2001. All rights reserved.

Column Heading Defaults

In iSQL*Plus, column headings are displayed in uppercase and centered.

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

LAST_NAME	HIRE_DATE	SALARY
King	17-JUN-87	24000
Kochhar	21-SEP-89	17000
De Haan	13-JAN-93	17000
Hunold	03-JAN-90	9000
Ernst	21-MAY-91	6000
■ ■ ■		
Higgins	07-JUN-94	12000
Gietz	07-JUN-94	8300

20 rows selected.

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

Arithmetic Expressions

Create expressions with number and date data by using arithmetic operators.

Operator	Description
+	Add
-	Subtract
*	Multiply
/	Divide

ORACLE

1-9

Copyright © Oracle Corporation, 2001. All rights reserved.

Arithmetic Expressions

You may need to modify the way in which data is displayed, perform **calculations**, or look at what-if scenarios. These are all possible 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 available in SQL. You can use **arithmetic operators** in any clause of a SQL statement except in the FROM clause.

Instructor Note

You can use the addition and subtraction operators only with DATE and TIMESTAMP data types.

Using Arithmetic Operators

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

LAST_NAME	SALARY	SALARY+300
King	24000	24300
Kochhar	17000	17300
De Haan	17000	17300
Hunold	9000	9300
Ernst	6000	6300

...

Hartstein	13000	13300
Fay	6000	6300
Higgins	12000	12300
Gietz	8300	8600

20 rows selected.

ORACLE

Using Arithmetic Operators

The example in the slide uses the addition operator to calculate a salary increase of \$300 for all employees and displays a new `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 Oracle9i server ignores blank spaces before and after the arithmetic operator.

Operator Precedence



- **Multiplication and division take priority over addition and subtraction.**
- **Operators of the same priority are evaluated from left to right.**
- **Parentheses are used to force prioritized evaluation and to clarify statements.**

ORACLE

1-11

Copyright © Oracle Corporation, 2001. All rights reserved.

Operator Precedence

If an arithmetic expression contains more than one operator, multiplication and division are evaluated first. If operators within an expression are of same priority, then evaluation is done from left to right.

You can use parentheses to force the expression within parentheses to be evaluated first.

Operator Precedence

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

LAST_NAME	SALARY	12*SALARY+100
King	24000	288100
Kochhar	17000	204100
De Haan	17000	204100
Hunold	9000	108100
Ernst	6000	72100

...

Hartstein	13000	156100
Fay	6000	72100
Higgins	12000	144100
Gietz	8300	99700

20 rows selected.

ORACLE

1-12

Copyright © Oracle Corporation, 2001. All rights reserved.

Operator Precedence (continued)

The example on the slide displays the last name, salary, and annual compensation of employees. It calculates the annual compensation as 12 multiplied by the monthly salary, plus a one-time bonus of \$100. Notice that multiplication is performed before addition.

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

Instructor Note

Demo: `1_prec1.sql`, `1_prec2.sql`

Purpose: To illustrate viewing a query containing no parentheses and executing a query with parentheses to override rules of precedence.

Using Parentheses

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

LAST_NAME	SALARY	12*(SALARY+100)
King	24000	289200
Kochhar	17000	205200
De Haan	17000	205200
Hunold	9000	109200
Ernst	6000	73200
...		
Hartstein	13000	157200
Fay	6000	73200
Higgins	12000	145200
Gietz	8300	100800

20 rows selected.

ORACLE

Using Parentheses

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

The example on the slide displays the last name, salary, and annual compensation of employees. It calculates the annual compensation as monthly salary plus a monthly bonus of \$100, multiplied by 12. Because of the parentheses, addition takes priority over multiplication.

Defining a Null Value

- A null is a value that is unavailable, unassigned, unknown, or inapplicable.
- A null is not the same as zero or a blank space.

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

LAST_NAME	JOB_ID	SALARY	COMMISSION_PCT
King	AD_PRES	24000	
Kochhar	AD_VP	17000	
...			
Zlotkey	SA_MAN	10500	.2
Abel	SA_REP	11000	.3
Taylor	SA_REP	8600	.2
...			
Gietz	AC_ACCOUNT	8300	

20 rows selected.

ORACLE

1-14

Copyright © Oracle Corporation, 2001. All rights reserved.

Null Values

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

A null is a value that is unavailable, unassigned, unknown, or inapplicable. A null is not the same as zero or a space. Zero is a number, and a space is a character.

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

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

Instructor Note

Demo: 1_null.sql

Purpose: To illustrate calculating with null values.

Null Values in Arithmetic Expressions

Arithmetic expressions containing a null value evaluate to null.

```
SELECT last_name, 12*salary*commission_pct  
FROM employees;
```

Kochhar	
King	
LAST_NAME	12*SALARY*COMMISSION_PCT
...	
Zlotkey	25200
Abel	39600
Taylor	20640
...	
Gietz	

20 rows selected.

ORACLE

1-15

Copyright © Oracle Corporation, 2001. All rights reserved.

Null Values (continued)

If any column value in an arithmetic expression is null, the result is **null**. For example, if you attempt to perform division with zero, you get an error. However, if you divide a number by null, the result is a null or unknown.

In the example on the slide, employee King does not get any commission. Because the COMMISSION_PCT column in the arithmetic expression is null, the result is null.

For more information, see *Oracle9i SQL Reference*, “Basic Elements of SQL.”

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 alias
- Requires double quotation marks if it contains spaces or special characters or is case sensitive

ORACLE

1-16

Copyright © Oracle Corporation, 2001. All rights reserved.

Column Aliases

When displaying the result of a query, *iSQL*Plus* normally uses the name of the selected column as the column heading. This heading may not be descriptive and hence 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 a space as a separator. By default, **alias headings** appear in uppercase. If the alias contains spaces or special characters (such as # or \$), or is case sensitive, enclose the alias in double quotation marks (" ").

Instructor Note

Within a SQL statement, a column alias can be used in both the **SELECT** clause and the **ORDER BY** clause. You cannot use column aliases in the **WHERE** clause. Both alias features comply with the ANSI SQL 92 standard.

Demo: `1_alias.sql`

Purpose: To illustrate the use of aliases in expressions.

Using Column Aliases

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

NAME	COMM
King	
Kochhar	
De Haan	

20 rows selected.

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

Name	Annual Salary
King	288000
Kochhar	204000
De Haan	204000

20 rows selected.

ORACLE

Column Aliases (continued)

The first example displays the names and the commission percentages of all the employees. Notice 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 notice 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 a previous 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 contain a space, it has been enclosed in double quotation marks. Notice that the column heading in the output is exactly the same as the column alias.

Instructor Note

Point out the optional AS keyword in the first example and the double quotation marks in the second example. Also show that the aliases always appear in uppercase, unless enclosed within double quotation marks.

Concatenation Operator

A concatenation operator:

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

ORACLE

1-18

Copyright © Oracle Corporation, 2001. All rights reserved.

Concatenation Operator

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.

Using the Concatenation Operator

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

Employees
KingAD_PRES
KochharAD_VP
De HaanAD_VP
HunoldIT_PROG
ErnstIT_PROG
LorentzIT_PROG
MourgosST_MAN
RajsST_CLERK

...

20 rows selected.

ORACLE

Concatenation Operator (continued)

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

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

Literal Character Strings

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

ORACLE

1-20

Copyright © Oracle Corporation, 2001. All rights reserved.

Literal Character Strings

A **literal** is a character, a number, or a date that is included in the `SELECT` list and that 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.

Date and character literals *must* be enclosed within single quotation marks (' '); number literals need not.

Using Literal Character Strings

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

Employee Details
King is a AD_PRES
Kochhar is a AD_VP
De Haan is a AD_VP
Hunold is a IT_PROG
Ernst is a IT_PROG
Lorentz is a IT_PROG
Mourgos is a ST_MAN
Rajs is a ST_CLERK
...
20 rows selected.

ORACLE

1-21

Copyright © Oracle Corporation, 2001. All rights reserved.

Literal Character Strings (continued)

The example on the slide displays last names and job codes of all employees. The column has the heading Employee Details. Notice 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;
```

MONTHLY
King: 1 Month salary = 24000
Kochhar: 1 Month salary = 17000
De Haan: 1 Month salary = 17000
Hunold: 1 Month salary = 9000
Ernst: 1 Month salary = 6000
Lorentz: 1 Month salary = 4200
Mourgos: 1 Month salary = 5800
Rajs: 1 Month salary = 3500
...

20 rows selected.

Duplicate Rows

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

```
SELECT department_id
FROM   employees;
```

DEPARTMENT_ID	
	90
	90
	90
	60
	60
	60
	50
	50
	50
...	
20 rows selected.	

ORACLE

Duplicate Rows

Unless you indicate otherwise, *iSQL*Plus* displays the results of a query without eliminating duplicate rows. The example on the slide displays all the department numbers from the `EMPLOYEES` table.

Notice that the department numbers are repeated.

Eliminating Duplicate Rows

Eliminate duplicate rows by using the **DISTINCT** keyword in the **SELECT** clause.

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

DEPARTMENT_ID	
	10
	20
	50
	60
	80
	90
	110
8 rows selected.	

ORACLE

1-23

Copyright © Oracle Corporation, 2001. All rights reserved.

Duplicate Rows (continued)

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

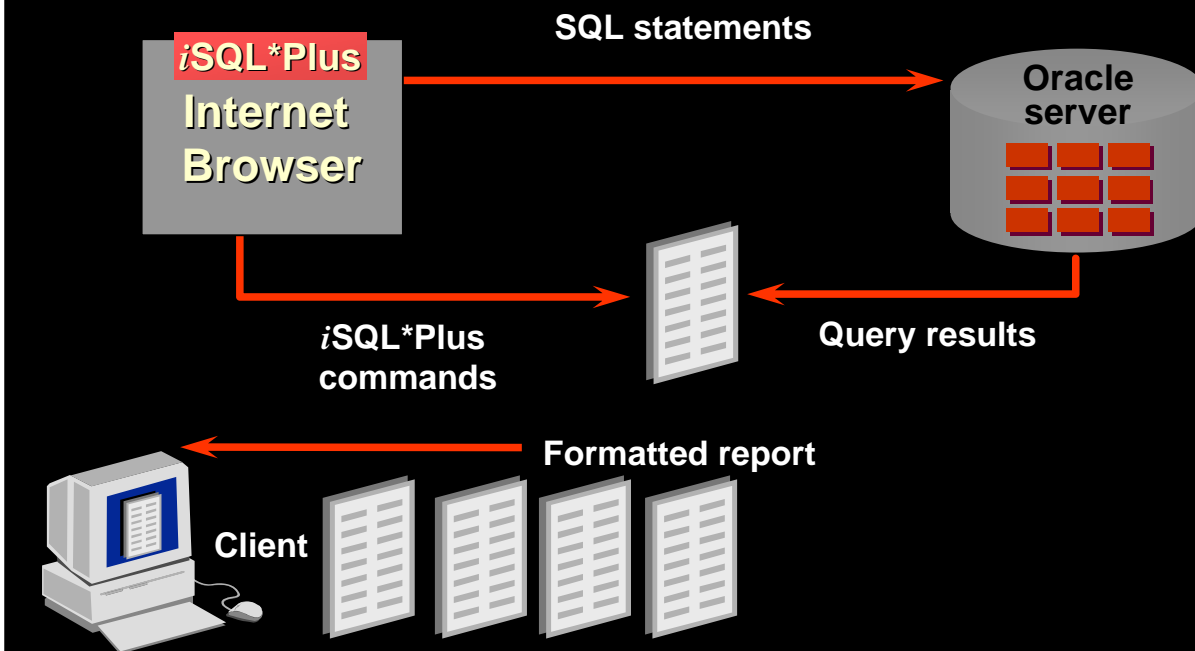
You can specify multiple columns after the **DISTINCT** qualifier. The **DISTINCT** qualifier affects all the selected columns, and the result is every distinct combination of the columns.

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

DEPARTMENT_ID	JOB_ID
10	AD_ASST
20	MK_MAN
20	MK_REP
50	ST_CLERK
50	ST_MAN
60	IT_PROG
...	
	SA_REP

13 rows selected.

SQL and iSQL*Plus Interaction



ORACLE

1-24

Copyright © Oracle Corporation, 2001. All rights reserved.

SQL and iSQL*Plus

SQL is a command language for communication with the Oracle server from any tool or application. Oracle SQL contains many extensions.

iSQL*Plus is an Oracle tool that recognizes and submits SQL statements to the Oracle server for execution and contains its own command language.

Features of SQL

- Can be used by a range of users, including those with little or no programming experience
- Is a nonprocedural language
- Reduces the amount of time required for creating and maintaining systems
- Is an English-like language

Features of iSQL*Plus

- Accessed from a browser
- Accepts ad hoc entry of statements
- Provides online editing for modifying SQL statements
- Controls environmental settings
- Formats query results into a basic report
- Accesses local and remote databases

SQL Statements Versus *i*SQL*Plus Commands

SQL

- A language
- ANSI standard
- Keyword cannot be abbreviated
- Statements manipulate data and table definitions in the database

**SQL
statements**

*i*SQL*Plus

- An environment
- Oracle proprietary
- Keywords can be abbreviated
- Commands do not allow manipulation of values in the database
- Runs on a browser
- Centrally loaded, does not have to be implemented on each machine

***i*SQL*Plus
commands**

ORACLE

1-25

Copyright © Oracle Corporation, 2001. All rights reserved.

SQL and *i*SQL*Plus (continued)

The following table compares **SQL** and ***i*SQL*Plus**:

SQL	<i>i</i> SQL*Plus
Is a language for communicating with the Oracle server to access data	Recognizes SQL statements and sends them to the server
Is based on American National Standards Institute (ANSI) standard SQL	Is the Oracle proprietary interface for executing SQL statements
Manipulates data and table definitions in the database	Does not allow manipulation of values in the database
Does not have a continuation character	Has a dash (-) as a continuation character if the command is longer than one line
Cannot be abbreviated	Can be abbreviated
Uses functions to perform some formatting	Uses commands to format data

Overview of *iSQL*Plus*

After you log into *iSQL*Plus*, you can:

- Describe the table structure
- Edit your SQL statement
- Execute SQL from *iSQL*Plus*
- Save SQL statements to files and append SQL statements to files
- Execute statements stored in saved files
- Load commands from a text file into the *iSQL*Plus* Edit window

ORACLE

1-26

Copyright © Oracle Corporation, 2001. All rights reserved.

*iSQL*Plus*

*iSQL*Plus* is an environment in which you can do the following:

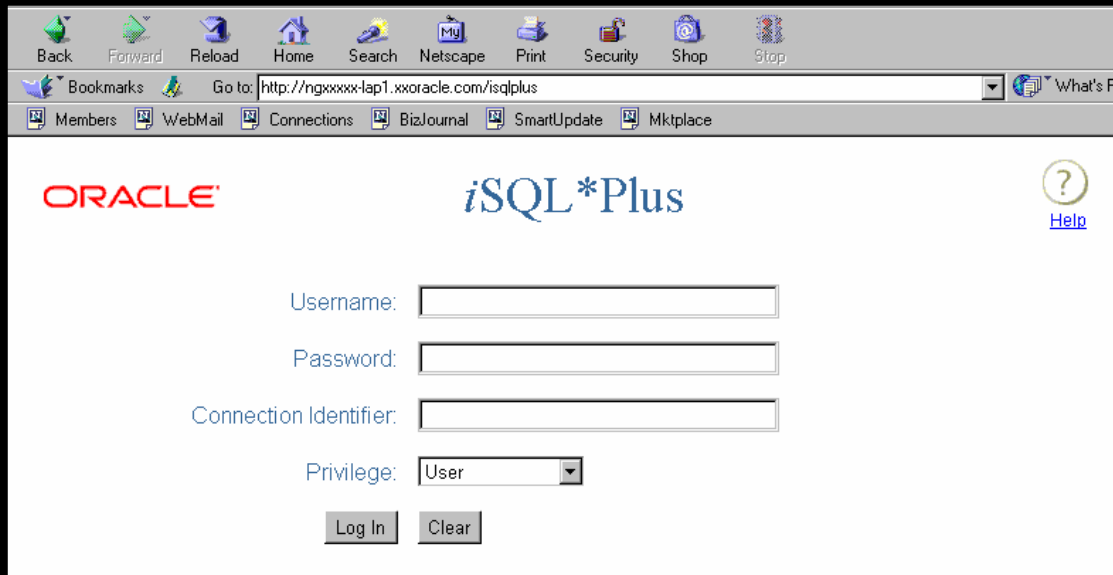
- **Execute SQL** statements to retrieve, modify, add, and remove data from the database
- Format, perform calculations on, store, and print query results in the form of reports
- **Create script** files to store SQL statements for repetitive use in the future

*iSQL*Plus* commands can be divided into the following main categories:

Category	Purpose
Environment	Affects the general behavior of SQL statements for the session
Format	Formats query results
File manipulation	Saves statements into text script files, and runs statements from text script files
Execution	Sends SQL statements from the browser to Oracle server
Edit	Modifies SQL statements in the Edit window
Interaction	Allows you to create and pass variables to SQL statements, print variable values, and print messages to the screen
Miscellaneous	Has various commands to connect to the database, manipulate the <i>iSQL*Plus</i> environment, and display column definitions

Logging In to *iSQL*Plus*

From your Windows browser environment:



The screenshot shows a Netscape browser window with the address bar displaying `http://ngxxxxx-lap1.xxoracle.com/isqlplus`. The browser's toolbar includes buttons for Back, Forward, Reload, Home, Search, Netscape, Print, Security, Shop, and Stop. Below the toolbar, there are bookmarks for Members, WebMail, Connections, BizJournal, SmartUpdate, and Mktplace. The main content area displays the Oracle logo and the *iSQL*Plus* title. A login form contains the following fields and controls:

- Username:
- Password:
- Connection Identifier:
- Privilege:
- Log In button
- Clear button

A Help link with a question mark icon is located in the top right corner of the page.

ORACLE

1-27

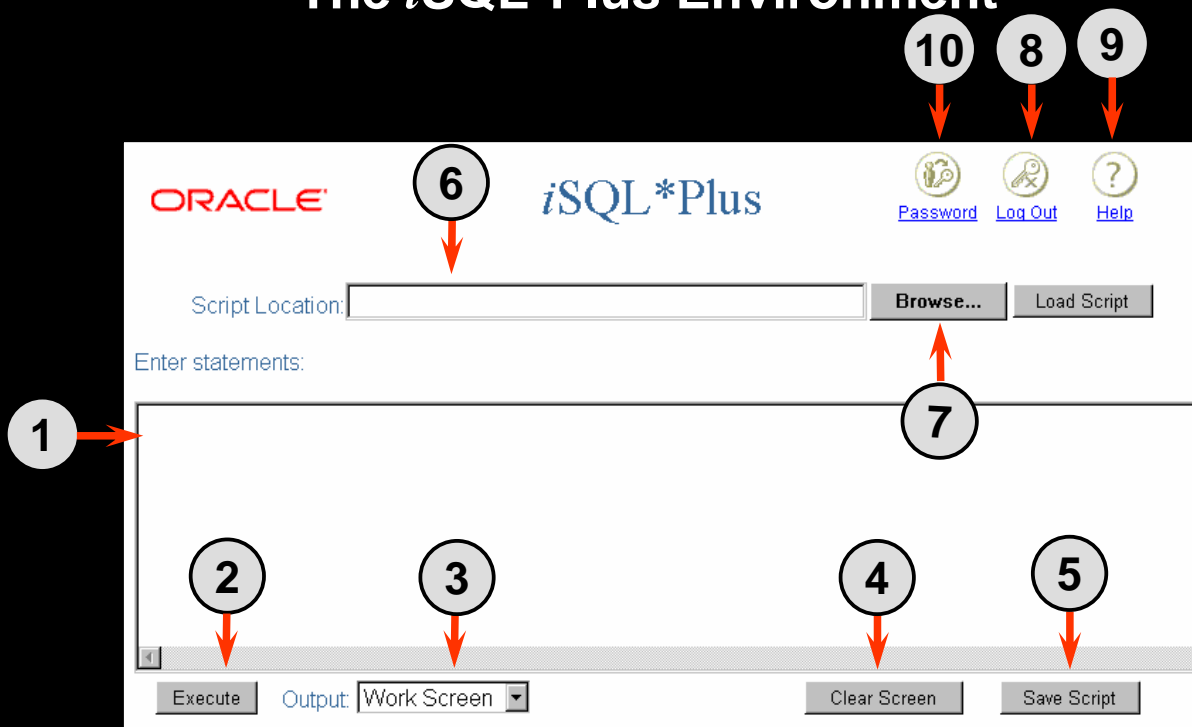
Copyright © Oracle Corporation, 2001. All rights reserved.

Logging In to *iSQL*Plus*

To log in through a browser environment:

1. Start the browser.
2. Enter the URL address of the *iSQL*Plus* environment.
3. Fill in the username, password and Oracle Connection Identifier fields.

The *iSQL*Plus* Environment



ORACLE

1-28

Copyright © Oracle Corporation, 2001. All rights reserved.

The *iSQL*Plus* Environment

Within the Windows browser, the *iSQL*Plus* window has several key areas:

1. Edit window: The area where you type the SQL statements and *iSQL*Plus* commands.
2. Execute button: Click to execute the statements and commands in the edit window.
3. Output Option: Defaults to Work Screen, which displays the results of the SQL statement beneath the edit window. The other options are File or Window. File saves the contents to a specified file. Window places the output on the screen, but in a separate window.
4. Clear Screen button: Click to clear text from the edit window.
5. Save Script button: Saves the contents of the edit window to a file.
6. Script Locator: Identifies the name and location of a script file that you want to execute.
7. Browse button: Used to search for a script file using the Windows File Open dialog box.
8. Exit icon: Click to end the *iSQL*Plus* session and return to the *iSQL*Plus* LogOn window.
9. Help icon: Provides access to *iSQL*Plus* Help documentation.
10. Password button: Is used to change your password.

Displaying Table Structure

Use the *iSQL*Plus* **DESCRIBE** command to display the structure of a table.

```
DESC[RIBE] tablename
```

ORACLE

1-29

Copyright © Oracle Corporation, 2001. All rights reserved.

Displaying the Table Structure

In *iSQL*Plus*, you can display the structure of a table using the **DESCRIBE command**. The command shows the column names and data types, as well as whether a column *must* contain data.

In the syntax:

tablename is the name of any existing table, view, or synonym accessible to the user

Displaying Table Structure

```
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)

ORACLE

1-30

Copyright © Oracle Corporation, 2001. All rights reserved.

Displaying the Table Structure (continued)

The example on the slide displays the information about the structure of the DEPARTMENTS table.

In the result:

Null? indicates whether a column *must* contain data; NOT NULL indicates that a column must contain data

Type displays the data type for a column

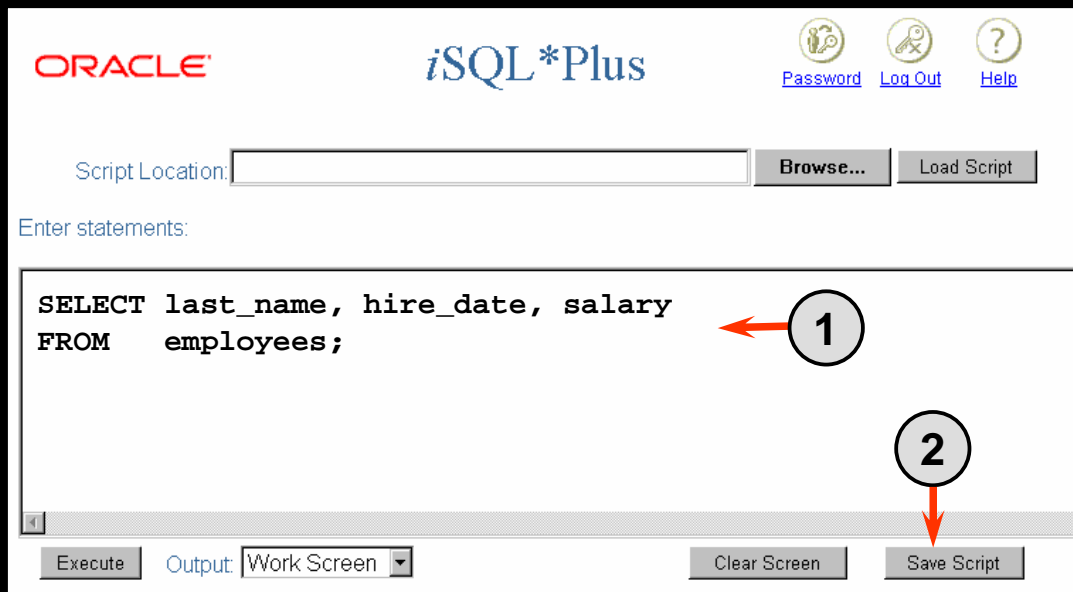
The data types are described in the following table:

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

Instructor Note

Inform students that the column sequence in `DESCRIBE tablename` is the same as that in `SELECT * FROM tablename`. The order in which the columns are displayed is determined when the table is created.

Interacting with Script Files



ORACLE

1-31

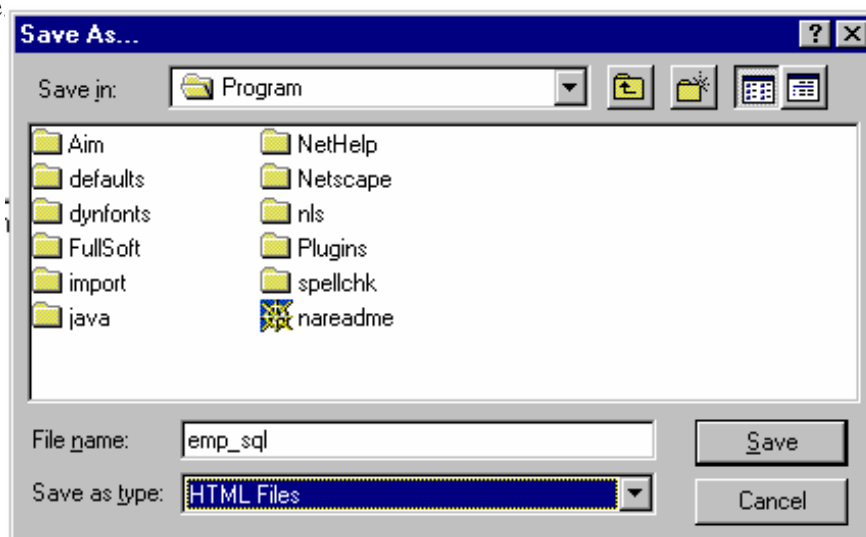
Copyright © Oracle Corporation, 2001. All rights reserved.

Interacting with Script Files

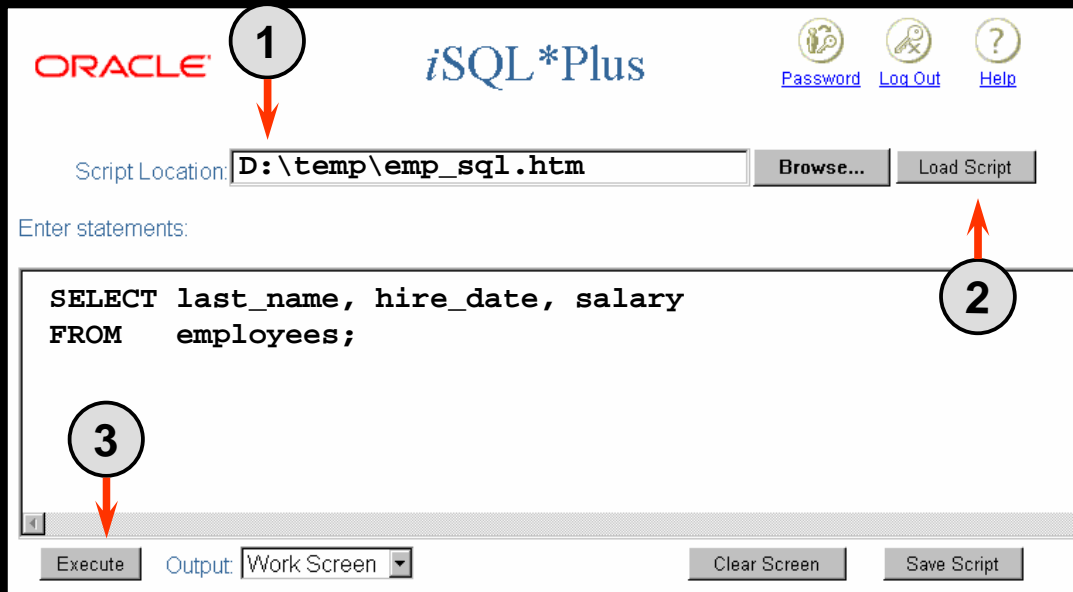
Placing Statements and Commands into a Text Script File

You can save commands and statements from the Edit window in *iSQL*Plus* to a text script file as follows:

1. Type the SQL statements into the edit window in *iSQL*Plus*.
2. Click the Save Script button. This opens the Windows File Save dialog box. Identify the name of the file. It defaults to .html extension. You can change the file type to a text file or save it as a .sql file.



Interacting with Script Files



ORACLE

1-32

Copyright © Oracle Corporation, 2001. All rights reserved.

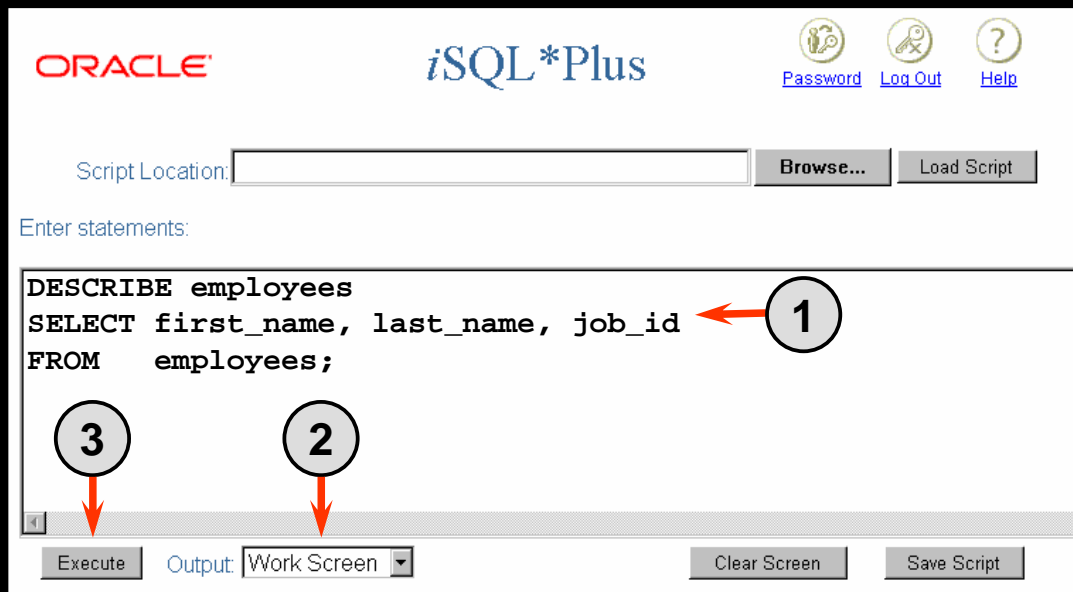
Interacting with Script Files

Using Statements and Commands from a Script File in iSQL*Plus

You can use previously saved commands and statements from a script file in *iSQL*Plus* as follows:

1. Type in the script name and location. Or, you can click the Browse button to find the script name and location.
2. Click the **Load Script** button. The file contents are loaded into the *iSQL*Plus* edit window.
3. Click the **Execute** button to run the contents of the *iSQL*Plus* edit window.

Interacting with Script Files



ORACLE

1-33

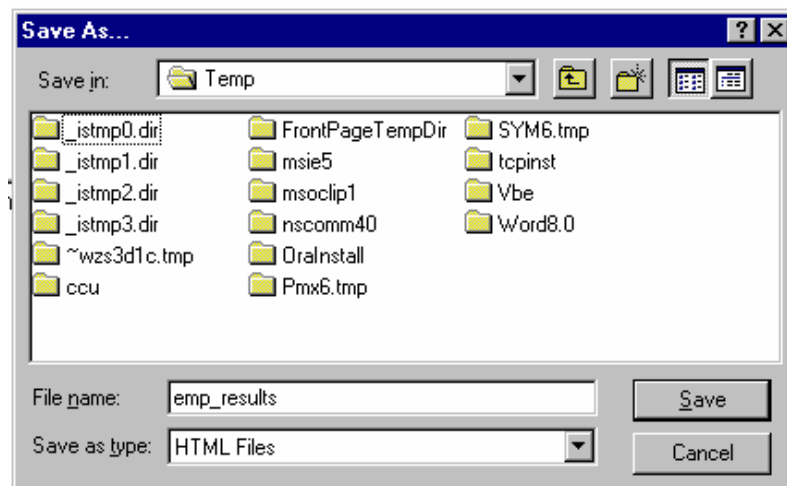
Copyright © Oracle Corporation, 2001. All rights reserved.

Interacting with Script Files

Saving Output to a File

You can save the results generated from a SQL statement or *iSQL*Plus* command to a file:

1. Type the SQL statements and *iSQL*Plus* commands into the edit window in *iSQL*Plus*.
2. Change the output option to Save.
3. Click the Execute button to run the contents of the *iSQL*Plus* edit window. This opens the Windows File Save dialog box. Identify the name of the file. It defaults to a .html extension. You can change the file type. The results are sent to the file specified.



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 give descriptive column headings
- Use the **iSQL*Plus** environment to write, save, and execute SQL statements and **iSQL*Plus** commands.

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

ORACLE

1-34

Copyright © Oracle Corporation, 2001. All rights reserved.

SELECT Statement

In this lesson, you should have learned about retrieving data from a database table with the **SELECT** statement.

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

In the syntax:

SELECT	is a list of one or more columns
*	selects all columns
DISTINCT	suppresses duplicates
<i>column/expression</i>	selects the named column or the expression
<i>alias</i>	gives selected columns different headings
FROM table	specifies the table containing the columns

iSQL*Plus

iSQL*Plus is an execution environment that you can use to send SQL statements to the database server and to edit and save SQL statements. Statements can be executed from the SQL prompt or from a script file.

Note: The SQL*Plus environment is covered in Appendix C.

Introduction to Oracle9i: SQL 1-34

Practice 1 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**
- **Using *iSQL*Plus***

ORACLE

1-35

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 1 Overview

This is the first of many practices. The solutions (if you require them) can be found in Appendix A. Practices are intended to introduce all topics covered in the lesson. Questions 2–4 are paper-based.

In any practice, there may be “if you have time” or “if you want an extra challenge” questions. Do these only if you have completed all other questions 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, attract the instructor’s attention.

Paper-Based Questions

For questions 2–4, circle either True or False.

Instructor Note

Let the students know that to get a listing of the tables they can access during the course, the command is:

```
SELECT * FROM TAB;
```

Practice 1

1. Initiate an *iSQL*Plus* session using the user ID and password provided by the instructor.
2. *iSQL*Plus* commands access the database.

True/False

3. The following `SELECT` statement executes successfully:

```
SELECT last_name, job_id, salary AS Sal
FROM   employees;
```

True/False

4. The following `SELECT` statement executes successfully:

```
SELECT *
FROM   job_grades;
```

True/False

5. There are four coding errors in this statement. Can you identify them?

```
SELECT      employee_id, last_name
sal x 12    ANNUAL SALARY
FROM        employees;
```

6. Show the structure of the `DEPARTMENTS` table. Select all data from the table.

Name	Null?	Type
DEPARTMENT_ID	NOT NULL	NUMBER(4)
DEPARTMENT_NAME	NOT NULL	VARCHAR2(30)
MANAGER_ID		NUMBER(6)
LOCATION_ID		NUMBER(4)

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		1700

8 rows selected.

Practice 1 (continued)

7. Show the structure of the EMPLOYEES table. Create a query to display the last name, job code, hire date, and employee number for each employee, with employee number appearing first. Provide an alias STARTDATE for the HIRE_DATE column. Save your SQL statement to a file named lab1_7.sql.

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)

8. Run your query in the file lab1_7.sql.

EMPLOYEE_ID	LAST_NAME	JOB_ID	STARTDATE
100	King	AD_PRES	17-JUN-87
101	Kochhar	AD_VP	21-SEP-89
102	De Haan	AD_VP	13-JAN-93
103	Hunold	IT_PROG	03-JAN-90
104	Ernst	IT_PROG	21-MAY-91
107	Lorentz	IT_PROG	07-FEB-99
124	Mourgos	ST_MAN	16-NOV-99
141	Rajs	ST_CLERK	17-OCT-95
142	Davies	ST_CLERK	29-JAN-97
143	Matos	ST_CLERK	15-MAR-98
144	Vargas	ST_CLERK	09-JUL-98
149	Zlotkey	SA_MAN	29-JAN-00
174	Abel	SA_REP	11-MAY-96
176	Taylor	SA_REP	24-MAR-98
■ ■ ■			
206	Gietz	AC_ACCOUNT	07-JUN-94

20 rows selected.

Practice 1 (continued)

9. Create a query to display unique job codes from the EMPLOYEES table.

JOB_ID
AC_ACCOUNT
AC_MGR
AD_ASST
AD PRES
AD_VP
IT_PROG
MK_MAN
MK_REP
SA_MAN
SA_REP
ST_CLERK
ST_MAN

12 rows selected.

If you have time, complete the following exercises:

10. Copy the statement from lab1_7.sql into the iSQL*Plus Edit window. Name the column headings Emp #, Employee, Job, and Hire Date, respectively. Run your query again.

Emp #	Employee	Job	Hire Date
100	King	AD PRES	17-JUN-87
101	Kochhar	AD_VP	21-SEP-89
102	De Haan	AD_VP	13-JAN-93
103	Hunold	IT_PROG	03-JAN-90
104	Ernst	IT_PROG	21-MAY-91
107	Lorentz	IT_PROG	07-FEB-99
124	Mourgos	ST_MAN	16-NOV-99
141	Rajs	ST_CLERK	17-OCT-95
142	Davies	ST_CLERK	29-JAN-97
143	Matos	ST_CLERK	15-MAR-98
144	Vargas	ST_CLERK	09-JUL-98
...			
206	Gietz	AC_ACCOUNT	07-JUN-94

20 rows selected.

Practice 1 (continued)

11. Display the last name concatenated with the job ID, separated by a comma and space, and name the column `Employee and Title`.

Employee and Title
King, AD_PRES
Kochhar, AD_VP
De Haan, AD_VP
Hunold, IT_PROG
Ernst, IT_PROG
Lorentz, IT_PROG
Mourgos, ST_MAN
Rajs, ST_CLERK
Davies, ST_CLERK
■ ■ ■
Gietz, AC_ACCOUNT

20 rows selected.

If you want an extra challenge, complete the following exercise:

12. Create a query to display all the data from the `EMPLOYEES` table. Separate each column by a comma. Name the column `THE_OUTPUT`.

THE_OUTPUT
100,Steven,King,SKING,515.123.4567,AD_PRES,,17-JUN-87,24000,,90
101,Neena,Kochhar,NKOCHHAR,515.123.4568,AD_VP,100,21-SEP-89,17000,,90
102,Lex,De Haan,LDEHAAN,515.123.4569,AD_VP,100,13-JAN-93,17000,,90
103,Alexander,Hunold,AHUNOLD,590.423.4567,IT_PROG,102,03-JAN-90,9000,,60
104,Bruce,Ernst,BERNST,590.423.4568,IT_PROG,103,21-MAY-91,6000,,60
107,Diana,Lorentz,DLORENTZ,590.423.5567,IT_PROG,103,07-FEB-99,4200,,60
124,Kevin,Mourgos,KMOURGOS,650.123.5234,ST_MAN,100,16-NOV-99,5800,,50
141,Trenna,Rajs,TRAJS,650.121.8009,ST_CLERK,124,17-OCT-95,3500,,50
142,Curtis,Davies,CDAVIES,650.121.2994,ST_CLERK,124,29-JAN-97,3100,,50
143,Randall,Matos,RMATOS,650.121.2874,ST_CLERK,124,15-MAR-98,2600,,50
144,Peter,Vargas,PVARGAS,650.121.2004,ST_CLERK,124,09-JUL-98,2500,,50
■ ■ ■
206,William,Gietz,WGIETZ,515.123.8181,AC_ACCOUNT,205,07-JUN-94,8300,,110

20 rows selected.

2

Restricting and Sorting Data

ORACLE

Copyright © Oracle Corporation, 2001. All rights reserved.

Schedule:

Timing

45 minutes

30 minutes

75 minutes

Topic

Lecture

Practice

Total

Objectives

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

- **Limit the rows retrieved by a query**
- **Sort the rows retrieved by a query**

ORACLE

2-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Lesson Aim

While retrieving data from the database, you may need to **restrict the rows** of data that are displayed or specify the **order** in which the rows are displayed. This lesson explains the SQL statements that you use to perform these actions.

Limiting Rows Using a Selection


EMPLOYEES

EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
100	King	AD_PRES	90
101	Kochhar	AD_VP	90
102	De Haan	AD_VP	90
103	Hunold	IT_PROG	60
104	Ernst	IT_PROG	60
107	Lorentz	IT_PROG	60
124	Mourgos	ST_MAN	50

...

20 rows selected.

**“retrieve all
employees
in department 90”**



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

ORACLE

Limiting Rows Using a Selection

In the example on 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 returned. This method of restriction is the basis of the **WHERE clause** in SQL.

Limiting the Rows Selected

- Restrict the rows returned by using the **WHERE** clause.

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

- The **WHERE** clause follows the **FROM** clause.

ORACLE

2-4

Copyright © Oracle Corporation, 2001. All rights reserved.

Limiting the Rows Selected

You can restrict the rows returned from the query by using the **WHERE clause**. A **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
<i>condition</i>	is composed of column names, expressions, constants, and a comparison operator

The **WHERE** clause can compare values in columns, literal values, 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
100	King	AD_PRES	90
101	Kochhar	AD_VP	90
102	De Haan	AD_VP	90

ORACLE

Using the WHERE Clause

In the example, the SELECT statement retrieves the name, job ID, and department number of all employees whose job ID is SA_REP.

Note that the job title SA_REP has been specified in uppercase to ensure that it matches the job ID column in the EMPLOYEES table. **Character strings** are case sensitive.

Character Strings and Dates

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

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

ORACLE

2-6

Copyright © Oracle Corporation, 2001. All rights reserved.

Character Strings and Dates

Character strings and dates in the **WHERE clause** must be enclosed in single quotation marks (' '). Number constants, however, should not be enclosed in 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 DD-MON-RR.

Note: Changing the default date format is covered in a subsequent lesson.

Instructor Note

Some students may ask how to override the case sensitivity. Later in the course, we cover the use of single-row functions such as UPPER and LOWER to override the case sensitivity.

Comparison Conditions

Operator	Meaning
=	Equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
<>	Not equal to

ORACLE

2-7

Copyright © Oracle Corporation, 2001. All rights reserved.

Comparison Conditions

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

Syntax

```
... WHERE expr operator value
```

For Example

```
... WHERE hire_date='01-JAN-95'  
... WHERE salary>=6000  
... WHERE last_name='Smith'
```

An **alias cannot** be used in the WHERE clause.

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

Using Comparison Conditions

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

LAST_NAME	SALARY
Matos	2600
Vargas	2500

ORACLE

2-8

Copyright © Oracle Corporation, 2001. All rights reserved.

Using the Comparison Conditions

In the example, the **SELECT statement** retrieves the last name and salary from the EMPLOYEES table, where the employee salary is less than or equal to 3000. 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.

Other Comparison Conditions

Operator	Meaning
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

Using the BETWEEN Condition

Use the **BETWEEN** condition 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
Rajs	3500
Davies	3100
Matos	2600
Vargas	2500

ORACLE

2-10

Copyright © Oracle Corporation, 2001. All rights reserved.

The BETWEEN Condition

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

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

Values specified with the **BETWEEN** condition are inclusive. You must specify the lower limit first.

Instructor Note

Emphasize that the values specified with the **BETWEEN** operator in the example are inclusive. Explain that **BETWEEN ... AND ...** is actually translated by Oracle server to a pair of **AND** conditions: **(a >= lower limit) AND (a <= higher limit)**. So using **BETWEEN ... AND ...** has no performance benefits, and it is used for logical simplicity.

Demo: 2_betw.sql

Purpose: To illustrate using the **BETWEEN** operator.

Using the IN Condition

Use the **IN** membership condition 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
202	Fay	6000	201
200	Whalen	4400	101
205	Higgins	12000	101
101	Kochhar	17000	100
102	De Haan	17000	100
124	Mourgos	5800	100
149	Zlotkey	10500	100
201	Hartstein	13000	100

8 rows selected.

ORACLE

2-11

Copyright © Oracle Corporation, 2001. All rights reserved.

The IN Condition

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

The slide example 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.

The IN condition 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 the list, they must be enclosed in single quotation marks (' ').

Instructor Note

Explain that IN (...) is actually translated by Oracle server to a set of OR conditions: a = value1 OR a = value2 OR a = value3. So using IN (...) has no performance benefits, and it is used for logical simplicity.

Demo: 2_in.sql

Purpose: To illustrate using the IN operator.

Using the LIKE Condition

- Use the **LIKE** condition to perform wildcard searches of valid search string values.
- Search conditions can contain either literal characters or numbers:
 - % denotes zero or many characters.
 - _ denotes one character.

```
SELECT    first_name
FROM      employees
WHERE     first_name LIKE 'S%';
```

ORACLE

2-12

Copyright © Oracle Corporation, 2001. All rights reserved.

The LIKE Condition

You may not always know the exact value to search for. You can select rows that match a character pattern by using the **LIKE condition**. 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 on the slide returns the employee first name from the **EMPLOYEES** table for any employee whose first name begins with an *S*. Note the uppercase *S*. Names beginning with an *s* are not returned.

The **LIKE** condition 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 1995 and December 1995:

```
SELECT last_name, hire_date
FROM   employees
WHERE  hire_date LIKE '%95';
```

Using the LIKE Condition

- You can combine pattern-matching characters.

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

LAST_NAME
Kochhar
Lorentz
Mourgos

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

ORACLE

2-13

Copyright © Oracle Corporation, 2001. All rights reserved.

Combining Wildcard Characters

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

The **ESCAPE** Option

When you need to have an exact match for the actual % and _ characters, use the **ESCAPE option**. This option specifies what the escape character is. If you want to search for strings that contain 'SA_', you can use the following SQL statement:

```
SELECT employee_id, last_name, job_id
FROM   employees
WHERE  job_id LIKE '%SA\_%' ESCAPE '\';
```

EMPLOYEE_ID	LAST_NAME	JOB_ID
149	Zlotkey	SA_MAN
174	Abel	SA_REP
176	Taylor	SA_REP
178	Grant	SA_REP

The **ESCAPE** option identifies the backslash (\) as the escape character. In the pattern, the escape character precedes the underscore (_). This causes the Oracle Server to interpret the underscore literally.

Using the 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
King	

ORACLE

2-14

Copyright © Oracle Corporation, 2001. All rights reserved.

The NULL Conditions

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 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 slide example retrieves the last names and managers of all employees who do not have a manager.

For another example, to display last name, job ID, and commission for all employees who are NOT entitled to get a commission, use the following SQL statement:

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

LAST_NAME	JOB_ID	COMMISSION_PCT
King	AD_PRES	
Kochhar	AD_VP	
...		
Higgins	AC_MGR	
Gietz	AC_ACCOUNT	

16 rows selected.

Logical Conditions

Operator	Meaning
AND	Returns TRUE if <i>both</i> component conditions are true
OR	Returns TRUE if <i>either</i> component condition is true
NOT	Returns TRUE if the following condition is false

ORACLE

2-15

Copyright © Oracle Corporation, 2001. All rights reserved.

Logical Conditions

A **logical condition** combines the result of two component conditions to produce a single result based on them or 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 one WHERE clause using the AND and OR operators.

Using the AND Operator

AND requires both 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
149	Zlotkey	SA_MAN	10500
201	Hartstein	MK_MAN	13000

ORACLE

2-16

Copyright © Oracle Corporation, 2001. All rights reserved.

The AND Operator

In the example, both conditions must be true for any record to be selected. Therefore, only 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. No rows are returned if MAN is not in uppercase. Character strings must be enclosed in 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

Instructor Note

Demo: [2_and.sql](#)

Purpose: To illustrate using the AND operator.

Using the OR Operator

OR requires either 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
100	King	AD_PRES	24000
101	Kochhar	AD_VP	17000
102	De Haan	AD_VP	17000
124	Mourgos	ST_MAN	5800
149	Zlotkey	SA_MAN	10500
174	Abel	SA_REP	11000
201	Hartstein	MK_MAN	13000
205	Higgins	AC_MGR	12000

8 rows selected.

ORACLE

2-17

Copyright © Oracle Corporation, 2001. All rights reserved.

The OR Operator

In the example, either condition can be true for any record to be selected. Therefore, any employee who has a job ID containing MAN *or* earns \$10,000 or more is selected.

The 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

Instructor Note

Demo: 2_or.sql

Purpose: To illustrate using the OR operator.

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
King	AD_PRES
Kochhar	AD_VP
De Haan	AD_VP
Mourgos	ST_MAN
Zlotkey	SA_MAN
Whalen	AD_ASST
Hartstein	MK_MAN
Fay	MK_REP
Higgins	AC_MGR
Gietz	AC_ACCOUNT

10 rows selected.

ORACLE

2-18

Copyright © Oracle Corporation, 2001. All rights reserved.

The NOT Operator

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

The NOT Truth Table

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

NOT	TRUE	FALSE	NULL
	FALSE	TRUE	NULL

Note: The NOT operator can also be used with other SQL operators, such as BETWEEN, LIKE, and NULL.

```
... WHERE job_id NOT IN ('AC_ACCOUNT', 'AD_VP')
... WHERE salary NOT BETWEEN 10000 AND 15000
... WHERE last_name NOT LIKE '%A%'
... WHERE commission_pct IS NOT NULL
```

Rules of Precedence

Order Evaluated	Operator
1	Arithmetic operators
2	Concatenation operator
3	Comparison conditions
4	IS [NOT] NULL, LIKE, [NOT] IN
5	[NOT] BETWEEN
6	NOT logical condition
7	AND logical condition
8	OR logical condition

Override rules of precedence by using parentheses.

ORACLE

2-19

Copyright © Oracle Corporation, 2001. All rights reserved.

Rules of Precedence

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

Rules of Precedence

```
SELECT last_name, job_id, salary
FROM   employees
WHERE  job_id = 'SA_REP'
OR     job_id = 'AD_PRES'
AND    salary > 15000;
```

LAST_NAME	JOB_ID	SALARY
King	AD_PRES	24000
Abel	SA_REP	11000
Taylor	SA_REP	8600
Grant	SA_REP	7000

ORACLE

2-20

Copyright © Oracle Corporation, 2001. All rights reserved.

Example of the Precedence of the AND Operator

In the slide example, there are two conditions:

- The first condition is that the job ID is AD_PRES *and* the salary is greater than 15,000.
- The second condition is that the job ID is SA_REP.

Therefore, the SELECT statement reads as follows:

“Select the row if an employee is a president *and* earns more than \$15,000, *or* if the employee is a sales representative.”

Instructor Note

Demo: 2_sal1.sql

Purpose: To illustrate the rules of precedence.

Rules of Precedence

Use parentheses to force priority.

```
SELECT last_name, job_id, salary
FROM employees
WHERE (job_id = 'SA_REP'
OR job_id = 'AD_PRES')
AND salary > 15000;
```

LAST_NAME	JOB_ID	SALARY
King	AD_PRES	24000

ORACLE

2-21

Copyright © Oracle Corporation, 2001. All rights reserved.

Using Parentheses

In the example, there are two conditions:

- The first condition is that the job ID is AD_PRES *or* SA_REP.
- The second condition is that salary is greater than \$15,000.

Therefore, the SELECT statement reads as follows:

“Select the row if an employee is a president *or* a sales representative, *and* if the employee earns more than \$15,000.”

Instructor Note

Demo: 2_sal2.sql

Purpose: To illustrate the rules of precedence.

ORDER BY Clause

- Sort rows with the ORDER BY clause
 - ASC: ascending order, default
 - DESC: descending order
- The ORDER BY clause comes last in the SELECT statement.

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

LAST_NAME	JOB_ID	DEPARTMENT_ID	HIRE_DATE
King	AD_PRES	90	17-JUN-87
Whalen	AD_ASST	10	17-SEP-87
Kochhar	AD_VP	90	21-SEP-89
Hunold	IT_PROG	60	03-JAN-90
Ernst	IT_PROG	60	21-MAY-91

...

20 rows selected.

ORACLE

2-22

Copyright © Oracle Corporation, 2001. All rights reserved.

The ORDER BY Clause

The **order of rows** returned in a query result is undefined. The **ORDER BY clause** can be used to sort the rows. If you use the ORDER BY clause, it must be the last clause of the SQL statement. You can specify an expression, or an alias, or column position as the sort condition.

Syntax

```
SELECT          expr
FROM            table
[WHERE          condition(s)]
[ORDER BY {column, expr} [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.

Instructor Note

Let the students know that the ORDER BY clause is executed last in query execution. It is placed last unless the FOR UPDATE clause is used.

Sorting in Descending Order

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

LAST_NAME	JOB_ID	DEPARTMENT_ID	HIRE_DATE
Zlotkey	SA_MAN	80	29-JAN-00
Mourgos	ST_MAN	50	16-NOV-99
Grant	SA_REP		24-MAY-99
Lorentz	IT_PROG	60	07-FEB-99
Vargas	ST_CLERK	50	09-JUL-98
Taylor	SA_REP	80	24-MAR-98
Matos	ST_CLERK	50	15-MAR-98
Fay	MK_REP	20	17-AUG-97
Davies	ST_CLERK	50	29-JAN-97

...

20 rows selected.

ORACLE

2-23

Copyright © Oracle Corporation, 2001. All rights reserved.

Default Ordering of Data

The **default sort order** is ascending:

- Numeric values are displayed with the lowest values first—for example, 1–999.
- Date values are displayed with the earliest value first—for example, 01-JAN-92 before 01-JAN-95.
- Character values are displayed in alphabetical order—for example, A first and Z last.
- Null values are displayed last for ascending sequences and first for descending sequences.

Reversing the Default Order

To reverse the order in which rows are displayed, specify the **DESC keyword** after the column name in the **ORDER BY** clause. The slide example sorts the result by the most recently hired employee.

Instructor Note

Let the students know that you can also sort by a column number in the **SELECT** list. The following example sorts the output in the descending order by salary:

```
SELECT last_name, salary
FROM employees
ORDER BY 2 DESC;
```

Sorting by Column Alias

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

EMPLOYEE_ID	LAST_NAME	ANNSAL
144	Vargas	30000
143	Matos	31200
142	Davies	37200
141	Rajs	42000
107	Lorentz	50400
200	Whalen	52800
124	Mourgos	69600
104	Ernst	72000
202	Fay	72000
178	Grant	84000

...

20 rows selected.

ORACLE

2-24

Copyright © Oracle Corporation, 2001. All rights reserved.

Sorting by Column Aliases

You can use a column **alias** in the **ORDER BY clause**. The slide example sorts the data by annual salary.

Instructor Note

Internally, the order of execution for a SELECT statement is as follows:

FROM clause

WHERE clause

SELECT clause

ORDER BY clause

Sorting by Multiple Columns

- The order of **ORDER BY** list is the order of sort.

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

LAST_NAME	DEPARTMENT_ID	SALARY
Whalen	10	4400
Hartstein	20	13000
Fay	20	6000
Mourgos	50	5800
Rajs	50	3500
Davies	50	3100
Matos	50	2600
Vargas	50	2500

...

20 rows selected.

- You can sort by a column that is not in the **SELECT** list.

ORACLE

2-25

Copyright © Oracle Corporation, 2001. All rights reserved.

Sorting by Multiple Columns

You can sort query results by more than one column. The sort limit is the number of columns in the given table.

In the **ORDER BY clause**, specify the columns, and separate the column names using commas. If you want to reverse the order of a column, specify **DESC** after its name. You can also order by columns that are not included in the **SELECT** clause.

Example

Display the last names and salaries of all employees. Order the result by department number, and then in descending order by salary.

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

Instructor Note

Show that the **DEPARTMENT_ID** column is sorted in ascending order and the **SALARY** column in descending order.

Summary

In this lesson, you should have learned how to:

- Use the **WHERE** clause to restrict rows of output
 - Use the comparison conditions
 - Use the **BETWEEN**, **IN**, **LIKE**, and **NULL** conditions
 - Apply the logical **AND**, **OR**, and **NOT** operators
- Use the **ORDER BY** clause to sort rows of output

```
SELECT      *|{[DISTINCT] column/expr [alias],...}  
FROM        table  
[WHERE      condition(s)]  
[ORDER BY   {column, expr, alias} [ASC|DESC]];
```

ORACLE

Summary

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

Practice 2 Overview

This practice covers the following topics:

- **Selecting data and changing the order of rows displayed**
- **Restricting rows by using the `WHERE` clause**
- **Sorting rows by using the `ORDER BY` clause**

ORACLE

2-27

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 2 Overview

This practice gives you a variety of exercises using the `WHERE` clause and the `ORDER BY` clause.

Practice 2

1. Create a query to display the last name and salary of employees earning more than \$12,000. Place your SQL statement in a text file named `lab2_1.sql`. Run your query.

LAST_NAME	SALARY
King	24000
Kochhar	17000
De Haan	17000
Hartstein	13000

2. Create a query to display the employee last name and department number for employee number 176.

LAST_NAME	DEPARTMENT_ID
Taylor	80

3. Modify `lab2_1.sql` to display the last name and salary for all employees whose salary is not in the range of \$5,000 and \$12,000. Place your SQL statement in a text file named

LAST_NAME	SALARY
King	24000
Kochhar	17000
De Haan	17000
Lorentz	4200
Rajs	3500
Davies	3100
Matos	2600
Vargas	2500
Whalen	4400
Hartstein	13000

10 rows selected.

Practice 2 (continued)

4. Display the employee last name, job ID, and start date of employees hired between February 20, 1998, and May 1, 1998. Order the query in ascending order by start date.

LAST_NAME	JOB_ID	HIRE_DATE
Matos	ST_CLERK	15-MAR-98
Taylor	SA_REP	24-MAR-98

5. Display the last name and department number of all employees in departments 20 and 50 in alphabetical order by name.

LAST_NAME	DEPARTMENT_ID
Davies	50
Fay	20
Hartstein	20
Matos	50
Mourgos	50
Rajs	50
Vargas	50

7 rows selected.

6. Modify lab2_3.sql to list the last name and salary of employees who earn between \$5,000 and \$12,000, and are in department 20 or 50. Label the columns Employee and Monthly Salary, respectively. Resave lab2_3.sql as lab2_6.sql. Run the statement in lab2_6.sql.

Employee	Monthly Salary
Mourgos	5800
Fay	6000

Practice 2 (continued)

7. Display the last name and hire date of every employee who was hired in 1994.

LAST_NAME	HIRE_DATE
Higgins	07-JUN-94
Gietz	07-JUN-94

8. Display the last name and job title of all employees who do not have a manager.

LAST_NAME	JOB_ID
King	AD_PRES

9. Display the last name, salary, and commission for all employees who earn commissions. Sort data in descending order of salary and commissions.

LAST_NAME	SALARY	COMMISSION_PCT
Abel	11000	.3
Zlotkey	10500	.2
Taylor	8600	.2
Grant	7000	.15

If you have time, complete the following exercises:

10. Display the last names of all employees where the third letter of the name is an *a*.

LAST_NAME
Grant
Whalen

11. Display the last name of all employees who have an *a* and an *e* in their last name.

LAST_NAME
De Haan
Davies
Whalen
Hartstein

Practice 2 (continued)

If you want an extra challenge, complete the following exercises:

12. Display the last name, job, and salary for all employees whose job is sales representative or stock clerk and whose salary is not equal to \$2,500, \$3,500, or \$7,000.

LAST_NAME	JOB_ID	SALARY
Davies	ST_CLERK	3100
Matos	ST_CLERK	2600
Abel	SA_REP	11000
Taylor	SA_REP	8600

13. Modify lab2_6.sql to display the last name, salary, and commission for all employees whose commission amount is 20%. Resave lab2_6.sql as lab2_13.sql. Rerun the statement in lab2_13.sql.

Employee	Monthly Salary	COMMISSION_PCT
Zlotkey	10500	.2
Taylor	8600	.2

3

Single-Row Functions

ORACLE

Copyright © Oracle Corporation, 2001. All rights reserved.

Schedule:	Timing	Topic
	55 minutes	Lecture
	30 minutes	Practice
	85 minutes	Total

Objectives

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

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

ORACLE

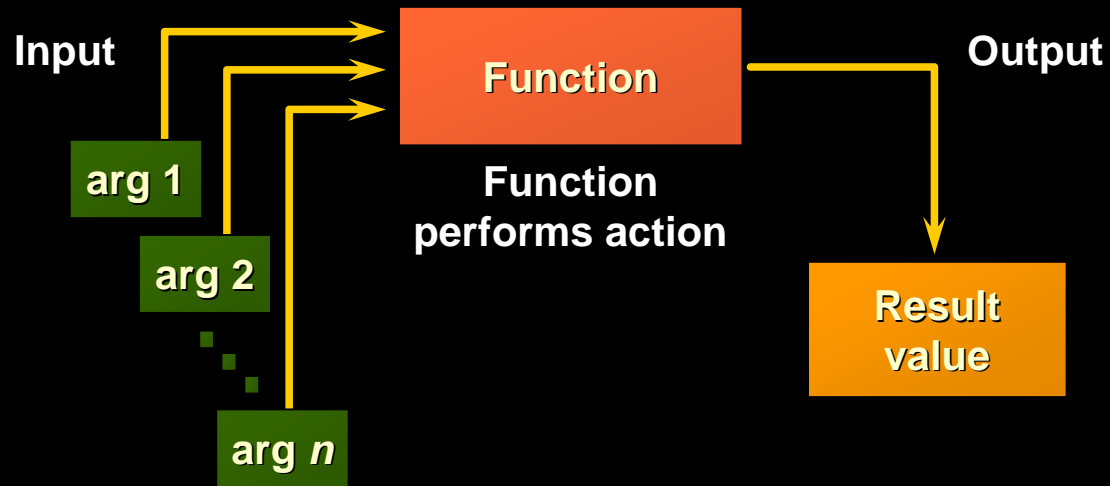
3-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Lesson Aim

Functions make the basic query block more powerful and 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, as well as those functions that convert data from one type to another, for example, character data to numeric data.

SQL Functions



ORACLE

3-3

Copyright © Oracle Corporation, 2001. All rights reserved.

SQL Functions

Functions are a very powerful feature of SQL and 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

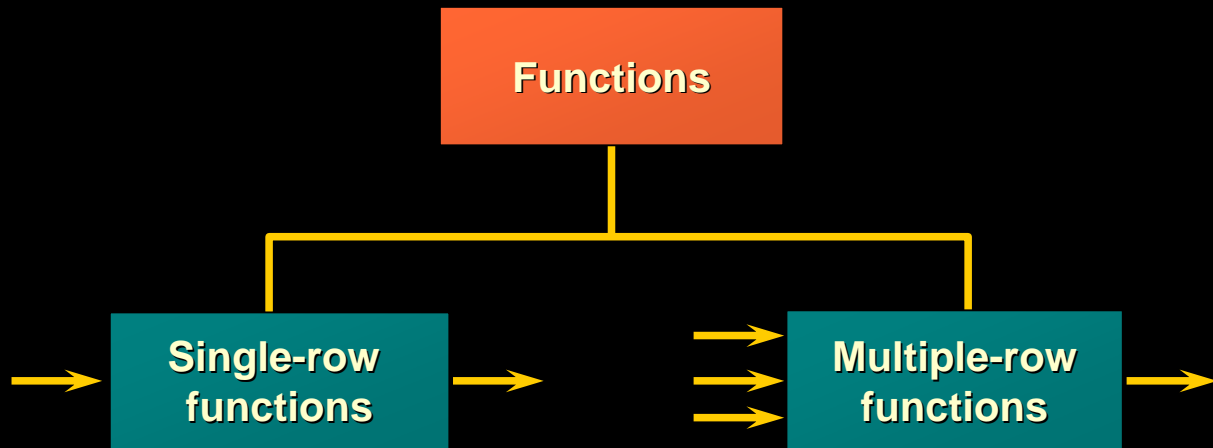
SQL functions sometimes take **arguments** and always **return a value**.

Note: Most of the functions described in this lesson are specific to Oracle's version of SQL.

Instructor Note

This lesson does not discuss all functions in great detail. It presents the most common functions with a brief explanation of them.

Two Types of SQL Functions



ORACLE

3-4

Copyright © Oracle Corporation, 2001. All rights reserved.

SQL Functions (continued)

There are two distinct 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 ones:

- Character
- Number
- Date
- Conversion

Multiple-Row Functions

Functions can manipulate groups of rows to give one result per group of rows. These functions are known as group functions. This is covered in a later lesson.

For more information, see *Oracle9i SQL Reference* for the complete list of available functions and their syntax.

Single-Row Functions

Single row functions:

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

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

ORACLE

3-5

Copyright © Oracle Corporation, 2001. All rights reserved.

Single-Row Functions

Single-row functions are used to manipulate data items. They accept one or more arguments and return one value for each row 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:

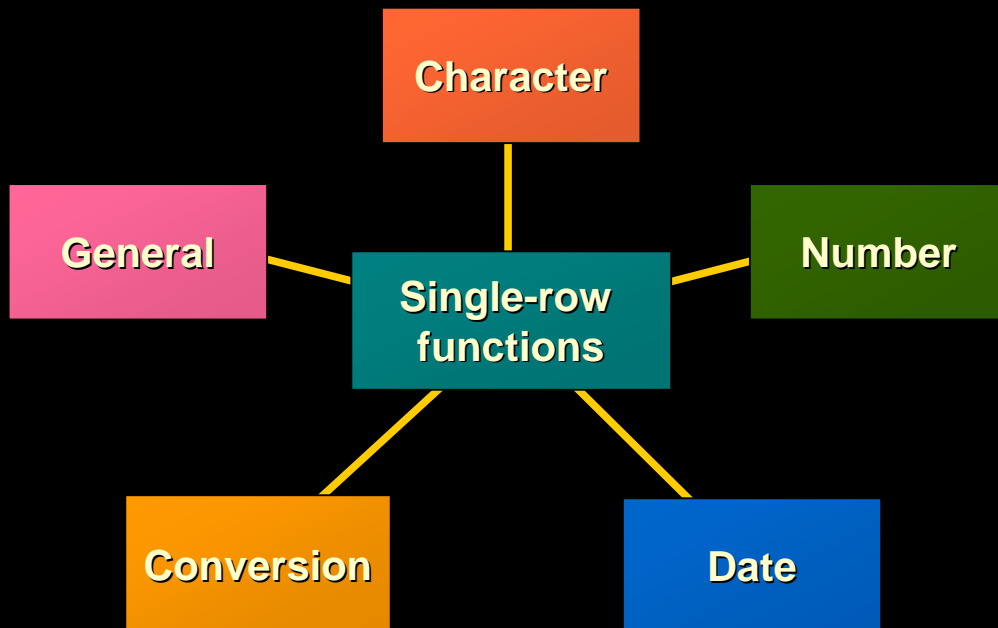
- Acting on each row returned in the query
- Returning one result per row
- Possibly returning a data value of a different type than that referenced
- Possibly expecting one or more arguments
- Can be used in SELECT, WHERE, and ORDER BY clauses; can be nested

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



ORACLE

3-6

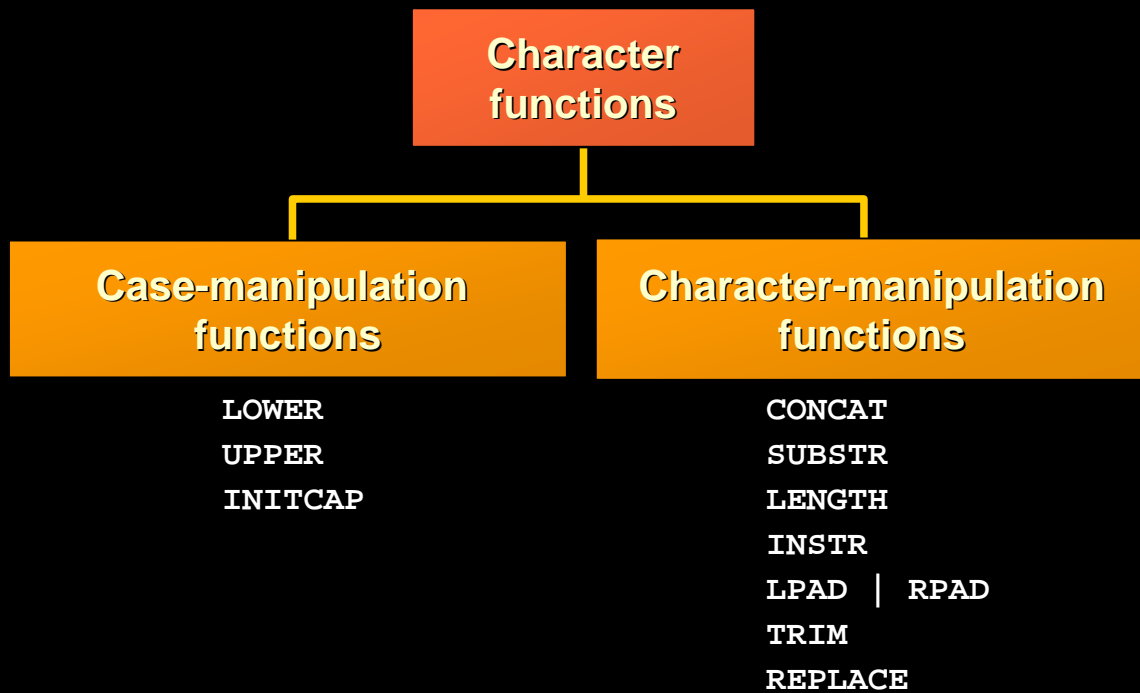
Copyright © Oracle Corporation, 2001. All rights reserved.

Single-Row Functions (continued)

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** (All date functions return a value of DATE data type except the **MONTHS_BETWEEN** function, which returns a number.)
- Conversion functions: Convert a value from one data type to another
- General functions:
 - **NVL**
 - **NVL2**
 - **NULLIF**
 - **COALSECE**
 - **CASE**
 - **DECODE**

Character Functions



ORACLE

3-7

Copyright © Oracle Corporation, 2001. All rights reserved.

Character Functions

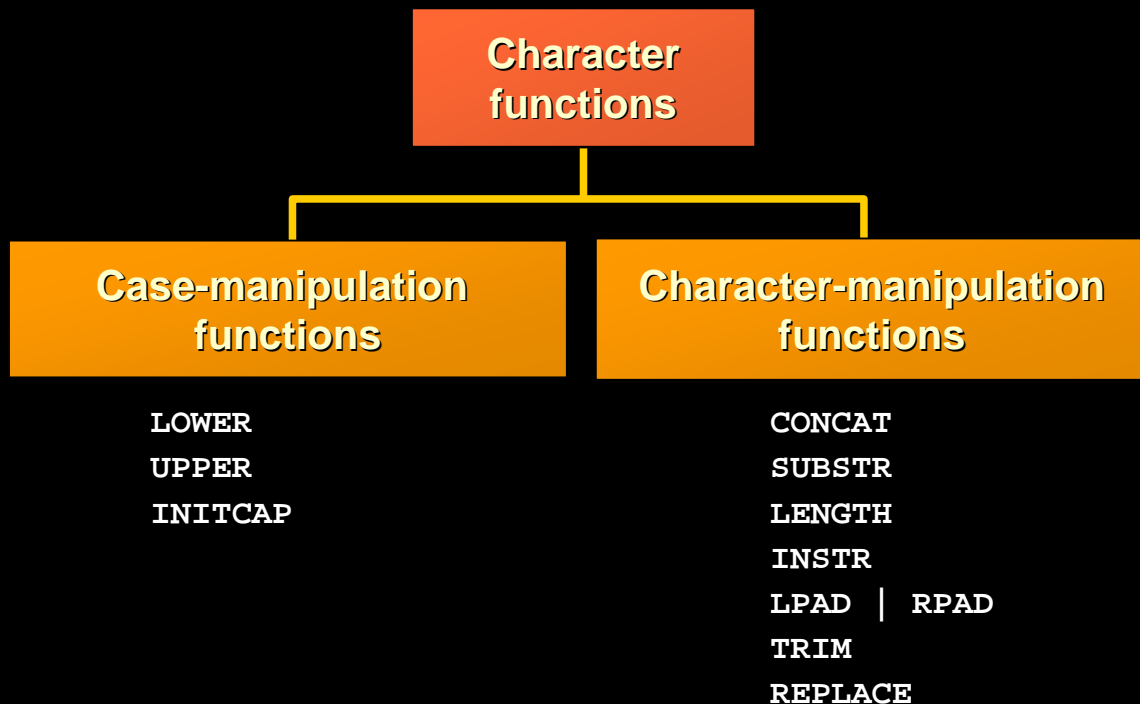
Single-row character functions accept character data as input and can return both character and numeric values. Character functions can be divided into the following:

- Case-manipulation functions
- Character-manipulation functions

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

Character Functions



ORACLE

3-8

Copyright © Oracle Corporation, 2001. All rights reserved.

Character Functions (continued)

Function	Purpose
LENGTH(<i>column</i> / <i>expression</i>)	Returns the number of characters in the expression
INSTR(<i>column</i> / <i>expression</i> , ' <i>string</i> ', [, <i>m</i>], [<i>n</i>])	Returns the numeric position of a named string. Optionally, you can provide a position <i>m</i> to start searching, and the occurrence <i>n</i> of the string. <i>m</i> and <i>n</i> default to 1, meaning start the search at the beginning of the search and report the first occurrence.
LPAD(<i>column</i> <i>expression</i> , <i>n</i> , ' <i>string</i> ') RPAD(<i>column</i> <i>expression</i> , <i>n</i> , ' <i>string</i> ')	Pads the character value right-justified to a total width of <i>n</i> character positions Pads the character value left-justified to a total width of <i>n</i> character positions
TRIM(<i>leading</i> / <i>trailing</i> / <i>both</i> , <i>trim_character</i> FROM <i>trim_source</i>)	Enables you to trim heading or trailing characters (or both) from a character string. If <i>trim_character</i> or <i>trim_source</i> is a character literal, you must enclose it in single quotes. This is a feature available from Oracle8i and later.
REPLACE(<i>text</i> , <i>search_string</i> , <i>replacement_string</i>)	Searches a text expression for a character string and, if found, replaces it with a specified replacement string

Case Manipulation Functions

These functions convert case for character strings.

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

ORACLE

3-9

Copyright © Oracle Corporation, 2001. All rights reserved.

Case Manipulation Functions

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 remaining letters to lowercase

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

EMPLOYEE DETAILS
The job id for KING is ad_pres
The job id for KOCHHAR is ad_vp
The job id for DE HAAN is ad_vp
■ ■ ■
The job id for HIGGINS is ac_mgr
The job id for GIETZ is ac_account

20 rows selected.

Using Case Manipulation 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';
no rows selected
```

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

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
205	Higgins	110

ORACLE

3-10

Copyright © Oracle Corporation, 2001. All rights reserved.

Case Manipulation Functions (continued)

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`, converting the `LAST_NAME` column to lowercase for comparison purposes. Since both names are lowercase now, 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 capitalized, 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
<code>CONCAT('Hello', 'World')</code>	HelloWorld
<code>SUBSTR('HelloWorld',1,5)</code>	Hello
<code>LENGTH('HelloWorld')</code>	10
<code>INSTR('HelloWorld', 'W')</code>	6
<code>LPAD(salary,10,'*')</code>	*****24000
<code>RPAD(salary, 10, '*')</code>	24000*****
<code>TRIM('H' FROM 'HelloWorld')</code>	elloWorld

ORACLE

3-11

Copyright © Oracle Corporation, 2001. All rights reserved.

Character Manipulation Functions

`CONCAT`, `SUBSTR`, `LENGTH`, `INSTR`, `LPAD`, `RPAD`, and `TRIM` are the character manipulation functions 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 numeric position of a named character
- `LPAD`: Pads the character value right-justified
- `RPAD`: Pads the character value left-justified
- `TRIM`: Trims heading 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 quotes.)

Instructor Note

Be sure to point out `RPAD` to the students, because this function is needed in a practice exercise. Also, `TRIM`, which was a new function in Oracle8i, does the job of both the `LTRIM` and the `RTRIM` functions.

Using the Character-Manipulation Functions

```
SELECT employee_id, CONCAT(first_name, last_name) NAME,
       job_id, LENGTH (last_name),
       INSTR(last_name, 'a') "Contains 'a'?"
FROM   employees
WHERE  SUBSTR(job_id, 4) = 'REP';
```

EMPLOYEE_ID	NAME	JOB_ID	LENGTH(LAST_NAME)	Contains 'a'?
174	EllenAbel	SA_REP	4	0
176	JonathonTaylor	SA_REP	6	2
178	KimberelyGrant	SA_REP	5	3
202	PatFay	MK_REP	3	2

ORACLE

3-12

Copyright © Oracle Corporation, 2001. All rights reserved.

Character-Manipulation Functions (continued)

The slide example displays employee first names and last names joined together, the length of the employee last name, and the numeric position of the letter *a* in the employee last name for all employees who have the string REP contained in the job ID starting at the fourth position of the job ID.

Example

Modify the SQL statement on the slide to display the data for those employees whose last names end with an *n*.

```
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';
```

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

Number Functions


- **ROUND:** Rounds value to specified decimal

`ROUND(45.926, 2)`  `45.93`

- **TRUNC:** Truncates value to specified decimal

`TRUNC(45.926, 2)`  `45.92`

- **MOD:** Returns remainder of division

`MOD(1600, 300)`  `100`

ORACLE

3-13

Copyright © Oracle Corporation, 2001. All rights reserved.

Number Functions

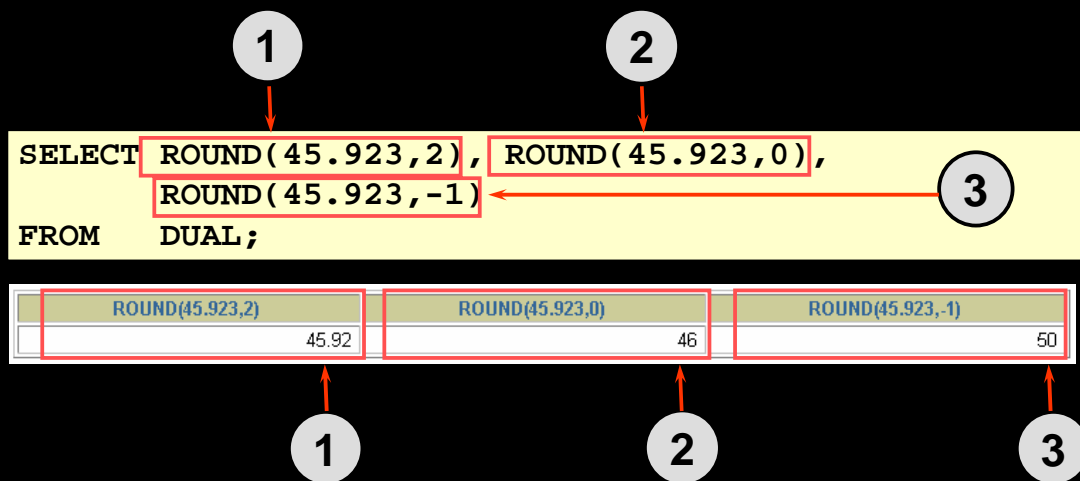
Number functions accept numeric input and return numeric values. This section describes some of the number functions.

Function	Purpose
<code>ROUND(column expression, n)</code>	Rounds the column, expression, or value to <i>n</i> decimal places, or, if <i>n</i> is omitted, no decimal places. (If <i>n</i> is negative, numbers to left of the decimal point are rounded.)
<code>TRUNC(column expression, n)</code>	Truncates the column, expression, or value to <i>n</i> decimal places, or, if <i>n</i> is omitted, then <i>n</i> defaults to zero
<code>MOD(m, n)</code>	Returns the remainder of <i>m</i> divided by <i>n</i>

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

For more information, see *Oracle9i SQL Reference*, “Number Functions.”

Using the ROUND Function



DUAL is a dummy table you can use to view results from functions and calculations.

ORACLE

3-14

Copyright © Oracle Corporation, 2001. All rights reserved.

ROUND Function

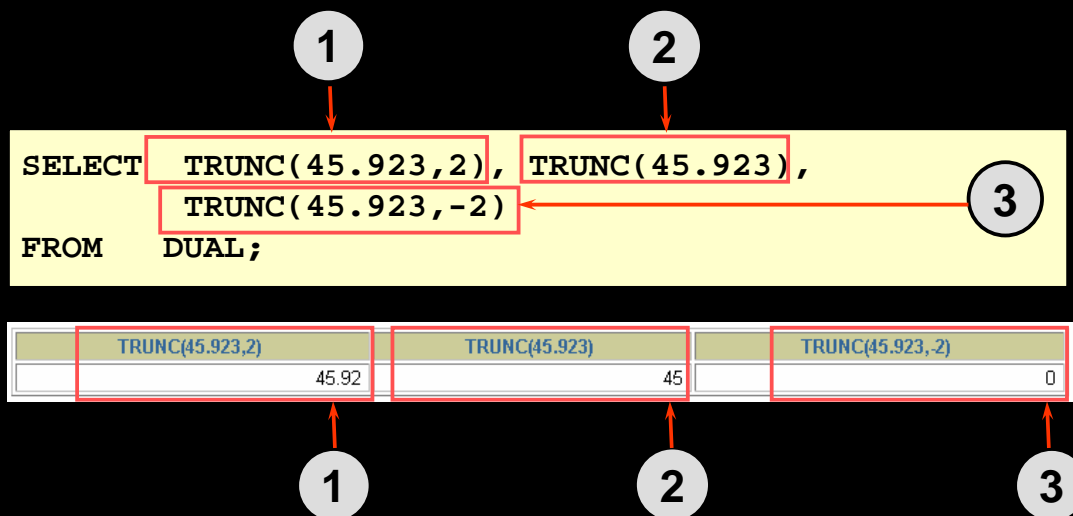
The **ROUND function** rounds the column, expression, or value to *n* decimal places. If the second argument is 0 or is missing, the value is rounded to zero decimal places. If the second argument is 2, the value is rounded to two decimal places. Conversely, if the second argument is -2, the value is rounded to two decimal places to the left.

The ROUND function can also be used with date functions. You will see examples later in this lesson.

The 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 once only, for instance, 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 SELECT clause syntax completeness, because both SELECT and FROM clauses are mandatory, and several calculations do not need to select from actual tables.

Using the TRUNC Function



ORACLE

3-15

Copyright © Oracle Corporation, 2001. All rights reserved.

TRUNC Function

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.

Like the ROUND function, the TRUNC function can be used with date functions.

Using the MOD Function

Calculate the remainder of a salary after it is divided by 5000 for all employees whose job title is sales representative.

```
SELECT last_name, salary, MOD(salary, 5000)
FROM   employees
WHERE  job_id = 'SA_REP';
```

LAST_NAME	SALARY	MOD(SALARY,5000)
Abel	11000	1000
Taylor	8600	3600
Grant	7000	2000

ORACLE

3-16

Copyright © Oracle Corporation, 2001. All rights reserved.

MOD Function

The **MOD function** finds the remainder of value1 divided by value2. The slide example calculates the remainder of the salary after dividing it by 5,000 for all employees whose job ID is SA_REP.

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

Instructor Note (for page 3-17)

You can change the default date display setting for a user session by executing the command:

```
ALTER SESSION SET NLS_DATE_FORMAT = 'date format model';
```

The DBA can set the date format for a database to a different format from the default. In either case, changing these settings is usually not a developer's role.

Working with Dates

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

```
SELECT last_name, hire_date
FROM employees
WHERE last_name like 'G%';
```

LAST_NAME	HIRE_DATE
Gietz	07-JUN-94
Grant	24-MAY-99

ORACLE

3-17

Copyright © Oracle Corporation, 2001. All rights reserved.

Oracle Date Format

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 for the employee Gietz is displayed in the default format DD-MON-RR. However, dates are not stored in this format. All the components of the date and time are stored. So, although a HIRE_DATE such as 07-JUN-94 is displayed as day, month, and year, there is also *time* and *century* information associated with it. The complete data might be June 7th, 1994 5:10:43 p.m.

This data is stored internally as follows:

CENTURY	YEAR	MONTH	DAY	HOURL	MINUTE	SECOND
19	94	06	07	5	10	43

Centuries and the Year 2000

The Oracle server is **year 2000 compliant**. 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 always stores year information as a four-digit number internally: two digits for the century and two digits for the year. For example, the Oracle database stores the year as 1996 or 2001, and not just as 96 or 01.

Working with Dates

SYSDATE is a function that returns:

- **Date**
- **Time**

ORACLE

3-18

Copyright © Oracle Corporation, 2001. All rights reserved.

The SYSDATE Function

SYSDATE is a date function that returns the current database server date and time. You can use SYSDATE just as you would use any other column name. For example, you can display the current date by selecting SYSDATE from a table. It is customary to select SYSDATE from a dummy table called **DUAL**.

Example

Display the current date using the DUAL table.

```
SELECT SYSDATE  
FROM   DUAL;
```

SYSDATE
28-SEP-01

Arithmetic with Dates

- Add or subtract a number to or 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.

ORACLE

3-19

Copyright © Oracle Corporation, 2001. All rights reserved.

Arithmetic with Dates

Since 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.

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

Using Arithmetic Operators with Dates

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

LAST_NAME	WEEKS
King	744.245395
Kochhar	626.102538
De Haan	453.245395

ORACLE

3-20

Copyright © Oracle Corporation, 2001. All rights reserved.

Arithmetic with Dates (continued)

The example on 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.

Note: **SYSDATE** is a SQL function that returns the current date and time. Your results may differ from the example.

If a more current date is subtracted from an older date, the difference is a negative number.

Date Functions

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

ORACLE

3-21

Copyright © Oracle Corporation, 2001. All rights reserved.

Date Functions

Date functions operate on Oracle dates. All date functions return a value of 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 date of the next specified day of the week ('*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*.
- **ROUND**(*date*[, '*fmt*']): Returns *date* rounded to the unit 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 specified by the format model *fmt*. If the format model *fmt* is omitted, *date* is truncated to the nearest day.

This list is a subset of the available date functions. The format models are covered later in this lesson. Examples of format models are month and year.

Using Date Functions

- **MONTHS_BETWEEN** ('01-SEP-95' , '11-JAN-94')
→ 19.6774194
- **ADD_MONTHS** ('11-JAN-94' , 6) → '11-JUL-94 '
- **NEXT_DAY** ('01-SEP-95' , 'FRIDAY')
→ '08-SEP-95 '
- **LAST_DAY**('01-FEB-95') → '28-FEB-95 '

ORACLE

3-22

Copyright © Oracle Corporation, 2001. All rights reserved.

Date Functions (continued)

For example, display the employee number, hire date, number of months employed, six-month review date, first Friday after hire date, and last day of the hire month for all employees employed for fewer than 36 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) < 36;
```

EMPLOYEE_ID	HIRE_DATE	TENURE	REVIEW	NEXT_DAY(LAST_DAY(
107	07-FEB-99	31.6982407	07-AUG-99	12-FEB-99	28-FEB-99
124	16-NOV-99	22.4079182	16-MAY-00	19-NOV-99	30-NOV-99
149	29-JAN-00	19.9885633	29-JUL-00	04-FEB-00	31-JAN-00
178	24-MAY-99	28.1498536	24-NOV-99	28-MAY-99	31-MAY-99

Using Date Functions

Assume SYSDATE = '25-JUL-95':

- ROUND(SYSDATE, 'MONTH') → 01-AUG-95
- ROUND(SYSDATE, 'YEAR') → 01-JAN-96
- TRUNC(SYSDATE, 'MONTH') → 01-JUL-95
- TRUNC(SYSDATE, 'YEAR') → 01-JAN-95

ORACLE

3-23

Copyright © Oracle Corporation, 2001. All rights reserved.

Date Functions (continued)

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.

Example

Compare the hire dates for all employees who started in 1997. Display the employee number, hire date, and start 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 '%97';
```

EMPLOYEE_ID	HIRE_DATE	ROUND(HIR	TRUNC(HIR
142	29-JAN-97	01-FEB-97	01-JAN-97
202	17-AUG-97	01-SEP-97	01-AUG-97

Instructor Note

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 with January 1st of the current year. Months 7-12 result in January 1st of the next year.

This is a good point to break the lesson in half. Have the students do Practice 3 - Part 1 (1-5) now.

Introduction to Oracle9i: SQL 3-23

Practice 3, Part One: Overview

This practice covers the following topics:

- **Writing a query that displays the current date**
- **Creating queries that require the use of numeric, character, and date functions**
- **Performing calculations of years and months of service for an employee**

ORACLE

3-24

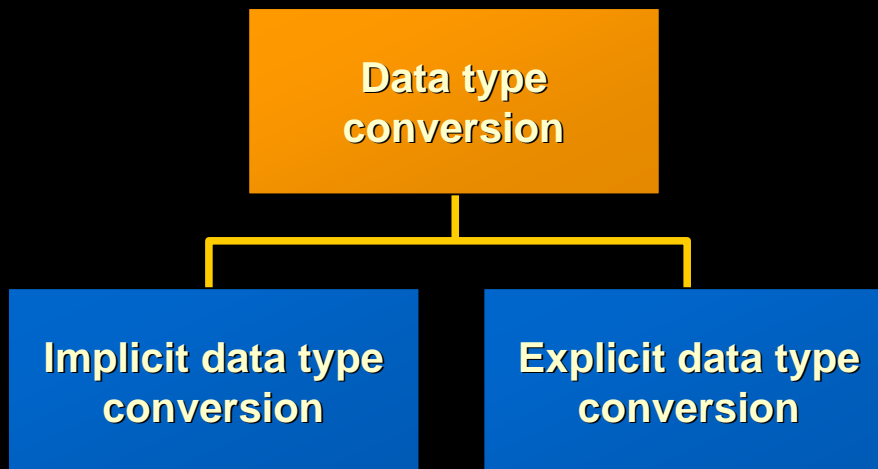
Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 3, Part One: Overview

This practice is designed to give you a variety of exercises using different functions available for character, number, and date data types.

Complete questions 1-5 at the end of this lesson.

Conversion Functions



ORACLE

3-25

Copyright © Oracle Corporation, 2001. All rights reserved.

Conversion Functions

In addition to Oracle data types, columns of tables in an Oracle9i database can be defined using ANSI, DB2, and SQL/DS **data types**. However, the Oracle server internally converts such data types to Oracle data types.

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

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

Explicit data type conversions are done 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; the last data type is the output.

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

Implicit Data Type Conversion

For assignments, the Oracle server can automatically convert the following:

From	To
VARCHAR2 or CHAR	NUMBER
VARCHAR2 or CHAR	DATE
NUMBER	VARCHAR2
DATE	VARCHAR2

ORACLE

3-26

Copyright © Oracle Corporation, 2001. All rights reserved.

Implicit Data Type Conversion

The assignment succeeds if the Oracle server can convert the data type of the value used in the assignment to that of the assignment target.

Instructor Note

There are several new data types available in the Oracle9i release pertaining to time. These include: `TIMESTAMP`, `TIMESTAMP WITH TIME ZONE`, `TIMESTAMP WITH LOCAL TIME ZONE`, `INTERVAL YEAR`, `INTERVAL DAY`. These are discussed later in the course.

You can also refer students to the *Oracle9i SQL Reference*, “Basic Elements of Oracle SQL.”

Implicit Data Type Conversion

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

From	To
VARCHAR2 or CHAR	NUMBER
VARCHAR2 or CHAR	DATE

ORACLE

3-27

Copyright © Oracle Corporation, 2001. All rights reserved.

Implicit Data Type Conversion (continued)

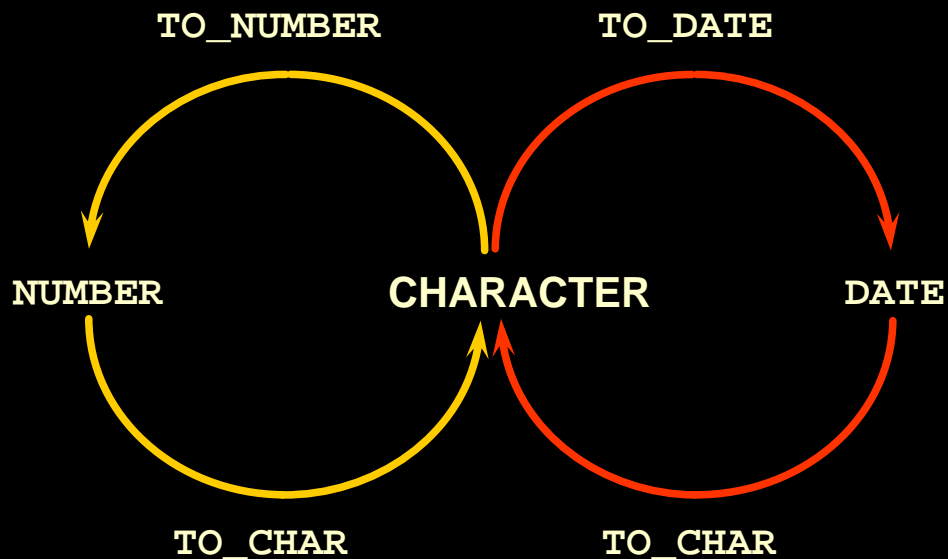
In general, the Oracle server uses the rule for expressions when a data type conversion is needed in places not covered by a rule for assignment conversions.

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

Instructor Note

Implicit data conversion is not solely performed on the data types mentioned. Other implicit data conversions can also be done. For example, VARCHAR2 can be implicitly converted to ROWID.

Explicit Data Type Conversion



ORACLE

3-28

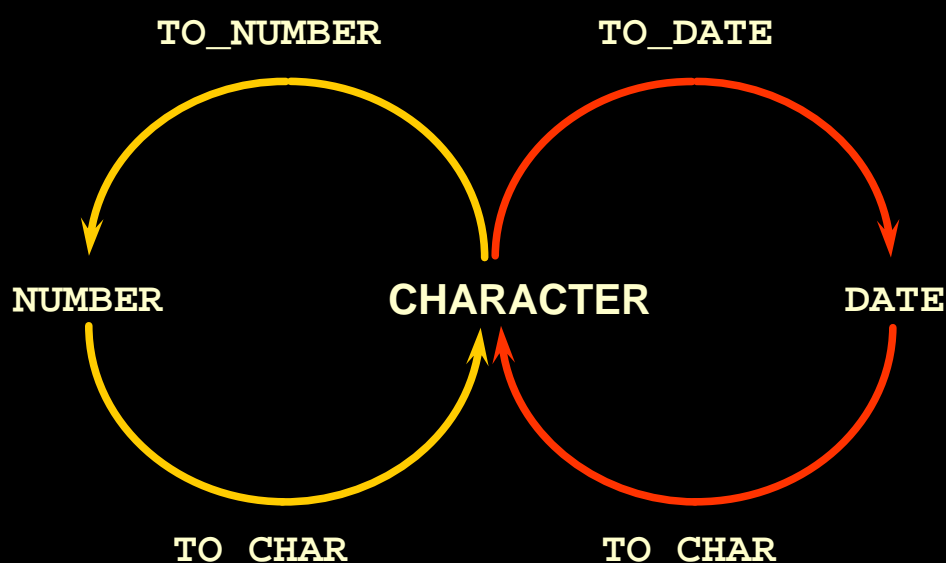
Copyright © Oracle Corporation, 2001. All rights reserved.

Explicit Data Type Conversion

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

Function	Purpose
<code>TO_CHAR(<i>number</i> <i>date</i>, [<i>fmt</i>], [<i>nlsparams</i>])</code>	<p>Converts a number or date value to a VARCHAR2 character string with format model <i>fmt</i>.</p> <p>Number Conversion: The <i>nlsparams</i> parameter specifies the following characters, which are returned by number format elements:</p> <ul style="list-style-type: none">• Decimal character• Group separator• Local currency symbol• International currency symbol <p>If <i>nlsparams</i> or any other parameter is omitted, this function uses the default parameter values for the session.</p>

Explicit Data Type Conversion



ORACLE

3-29

Copyright © Oracle Corporation, 2001. All rights reserved.

Explicit Data Type Conversion (continued)

Function	Purpose
<code>TO_CHAR(<i>number</i> <i>date</i>, [<i>fmt</i>], [<i>nlsparams</i>])</code>	Date Conversion: The <code>nlsparams</code> parameter specifies the language in which month and day names and abbreviations are returned. If this parameter is omitted, this function uses the default date languages for the session.
<code>TO_NUMBER(<i>char</i>, [<i>fmt</i>], [<i>nlsparams</i>])</code>	Converts a character string containing digits to a number in the format specified by the optional format model <i>fmt</i> . The <code>nlsparams</code> parameter has the same purpose in this function as in the <code>TO_CHAR</code> function for number conversion.
<code>TO_DATE(<i>char</i>, [<i>fmt</i>], [<i>nlsparams</i>])</code>	Converts a character string representing a date to a date value according to the <i>fmt</i> specified. If <i>fmt</i> is omitted, the format is DD-MON-YY. The <code>nlsparams</code> parameter has the same purpose in this function as in the <code>TO_CHAR</code> function for date conversion.

Explicit Data Type Conversion (continued)

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

For more information, see *Oracle9i SQL Reference*, “Conversion Functions.”

Using the TO_CHAR Function with Dates

```
TO_CHAR(date, 'format_model')
```

The format model:

- Must be enclosed in single quotation marks and is case sensitive
- Can include any valid date format element
- Has an *fm* element to remove padded blanks or suppress leading zeros
- Is separated from the date value by a comma

ORACLE

3-31

Copyright © Oracle Corporation, 2001. All rights reserved.

Displaying a Date in a Specific Format

Previously, all Oracle date values were displayed in the DD-MON-YY format. You can use the **TO_CHAR function** to convert a date from this default format to one specified by you.

Guidelines

- The **format model** must be enclosed in single quotation marks and is case sensitive.
- The format model can include any valid date format element. Be sure to separate the date value from the format model by 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**.
- You can format the resulting character field with the *iSQL*Plus* COLUMN command covered in a later lesson.

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

EMPLOYEE_ID	MONTH
205	06/94

Elements of the Date Format Model

YYYY	Full year in numbers
YEAR	Year spelled out
MM	Two-digit value for 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

Sample Format Elements of Valid Date Formats

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 digits of 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	B.C./D. indicator
B.C. or A.D.	B.C./A.D. indicator with periods
Q	Quarter of year
MM	Month: two-digit value
MONTH	Name of month padded with blanks to length of nine characters
MON	Name of month, three-letter abbreviation
RM	Roman numeral month
WW or W	Week of year or month
DDD or DD or D	Day of year, month, or week
DAY	Name of day padded with blanks to a length of nine characters
DY	Name of day; three-letter abbreviation
J	Julian day; the number of days since 31 December 4713 B.C.

Instructor Note

Emphasize the format D, as the students need it for practice 10. The D format returns a value from 1 to 7 representing the day of the week. Depending on the NLS date setting options, the value 1 may represent Sunday or Monday. In the United States, the value 1 represents Sunday.

Elements of the Date Format Model

- Time elements format the time portion of the date.

HH24:MI:SS AM

15:45:32 PM

- Add character strings by enclosing them in double quotation marks.

DD "of" MONTH

12 of OCTOBER

- Number suffixes spell out numbers.

ddspth

fourteenth

ORACLE

3-34

Copyright © Oracle Corporation, 2001. All rights reserved.

Date Format Elements - Time Formats

Use the formats 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 or HH24	Hour of day, or hour (1–12), or hour (0–23)
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, DDSF 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
King	17 June 1987
Kochhar	21 September 1989
De Haan	13 January 1993
Hunold	3 January 1990
Ernst	21 May 1991
Lorentz	7 February 1999
Mourgos	16 November 1999

20 rows selected.

ORACLE

3-36

Copyright © Oracle Corporation, 2001. All rights reserved.

The TO_CHAR Function with Dates

The SQL statement on the slide displays the last names and hire dates for all the employees. The hire date appears as 17 June 1987.

Example

Modify the slide example to display the dates in a format that appears as Seventh of June 1994 12:00:00 AM.

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

LAST_NAME	HIREDATE
King	Seventeenth of June 1987 12:00:00 AM
Kochhar	Twenty-First of September 1989 12:00:00 AM
Higgins	Seventh of June 1994 12:00:00 AM
Gietz	Seventh of June 1994 12:00:00 AM

20 rows selected.

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

Using the TO_CHAR Function with Numbers

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

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

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 thousand indicator

ORACLE

3-37

Copyright © Oracle Corporation, 2001. All rights reserved.

The TO_CHAR Function with Numbers

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
.	Decimal point in position specified	999999.99	1234.00
,	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
V	Multiply by 10 <i>n</i> times (<i>n</i> = number of 9s after V)	9999V99	123400
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
\$6,000.00

ORACLE

3-38

Copyright © Oracle Corporation, 2001. All rights reserved.

Guidelines

- The Oracle server displays a string of **hash 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 spaces provided in the format model.

Instructor Note (for page 3-39)

You can demonstrate the code using the `fx` modifier in the file 3_39n. Run the file with the `fx` modifier present, then remove the `fx` modifier and run the statement again.

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 matching for the character argument and date format model of a TO_DATE function

ORACLE

3-39

Copyright © Oracle Corporation, 2001. All rights reserved.

The TO_NUMBER and TO_DATE Functions

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** functions. The format model you choose is based on the previously demonstrated format elements.

The “**fx**” modifier specifies exact matching 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**, Oracle 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**, numbers in the character argument can omit leading zeroes.

The TO_NUMBER and TO_DATE Functions (continued)

Example

Display the names and hire dates of all the employees who joined on May 24, 1999. 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, 1999', 'fxMonth DD, YYYY');
```

```
WHERE hire_date = TO_DATE('May 24, 1999', 'fxMonth DD, YYYY')
      *
```

ERROR at line 3:

ORA-01858: a non-numeric character was found where a numeric was expected

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

ORACLE

3-41

Copyright © Oracle Corporation, 2001. All rights reserved.

The RR Date Format Element

The **RR date format** is similar to the YY element, but you can use it to specify different centuries. You can 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 on 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

Instructor Note

RR is available in Oracle7, not Oracle version 6. NLS parameters can be added to the `init.ora` file to set default date formats and language names and abbreviations. For more information, see *Oracle9i SQL Reference*, “ALTER SESSION”.

Demo: `3_hire.sql`

Purpose: To illustrate date format model elements.

Example of RR Date Format

To find employees hired prior to 1990, use the RR 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-90', 'DD-Mon-RR');
```

LAST_NAME	TO_CHAR(HIR
King	17-Jun-1987
Kochhar	21-Sep-1989
Whalen	17-Sep-1987

ORACLE

The RR Date Format Element Example

To find employees who were hired prior to 1990, the **RR format** can be used. Since the year is now greater than 1999, the RR format interprets the year portion of the date from 1950 to 1999.

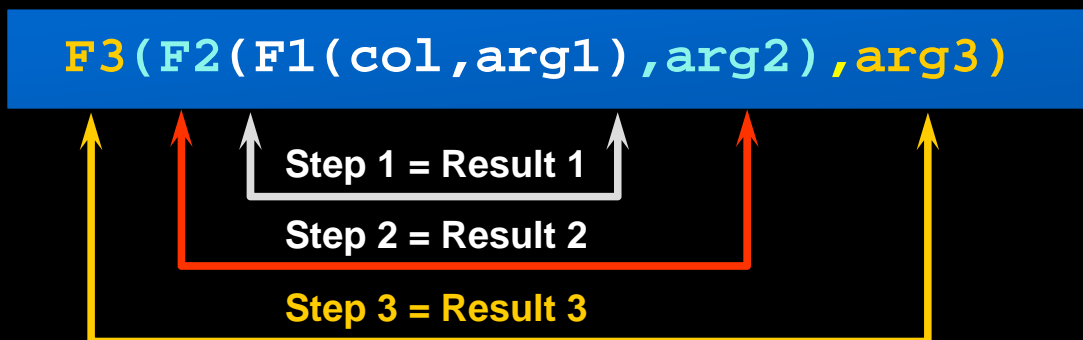
The following command, on the other hand, results in no rows being selected because the YY format interprets the year portion of the date in the current century (2090).

```
SELECT last_name, TO_CHAR(hire_date, 'DD-Mon-yyyy')
FROM   employees
WHERE  TO_DATE(hire_date, 'DD-Mon-yy') < '01-Jan-1990';
```

no rows selected

Nesting Functions

- Single-row functions can be nested to any level.
- Nested functions are evaluated from deepest level to the least deep level.



ORACLE

Nesting 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

```
SELECT last_name,  
       NVL(TO_CHAR(manager_id), 'No Manager')  
FROM   employees  
WHERE  manager_id IS NULL;
```

LAST_NAME	NVL(TO_CHAR(MANAGER_ID), 'NOMANAGER')
King	No Manager

ORACLE

3-44

Copyright © Oracle Corporation, 2001. All rights reserved.

Nesting Functions (continued)

The slide example displays the head of the company, who has no manager. The evaluation of the SQL statement involves two steps:

1. Evaluate the inner function to convert a number value to a character string.
 - Result1 = TO_CHAR(manager_id)
2. Evaluate the outer function to replace the null value with a text string.
 - NVL(Result1, 'No Manager')

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

Example

Display the date of the next Friday that is six months from the hire date. The resulting date should appear as Friday, August 13th, 1999. Order the results by hire date.

```
SELECT      TO_CHAR(NEXT_DAY(ADD_MONTHS  
                        (hire_date, 6), 'FRIDAY'),  
                        'fmDay, Month DDth, YYYY')  
            "Next 6 Month Review"  
FROM        employees  
ORDER BY    hire_date;
```

Instructor Note

Demo: [3_nest.sql](#)

Purpose: To illustrate nesting of several single row functions

General Functions

These 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)

ORACLE

3-45

Copyright © Oracle Corporation, 2001. All rights reserved.

General Functions

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, or the first expression if they are not equal
COALESCE	Returns the first non-null expression in the expression list

Note: For more information on the hundreds of functions available, see *Oracle9i SQL Reference*, “Functions.”

NVL Function

Converts a null 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-97')`
 - `NVL(job_id,'No Job Yet')`

ORACLE

3-46

Copyright © Oracle Corporation, 2001. All rights reserved.

The NVL Function

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 to convert 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	<code>NVL(number_column,9)</code>
DATE	<code>NVL(date_column, '01-JAN-95')</code>
CHAR or VARCHAR2	<code>NVL(character_column, 'Unavailable')</code>

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
King	24000	0	288000
Kochhar	17000	0	204000
De Haan	17000	0	204000
Hunold	9000	0	108000
Ernst	6000	0	72000
Lorentz	4200	0	50400
Mourgos	5800	0	69600
Rajs	3500	0	42000

20 rows selected.

ORACLE

3-47

Copyright © Oracle Corporation, 2001. All rights reserved.

The NVL Function

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

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

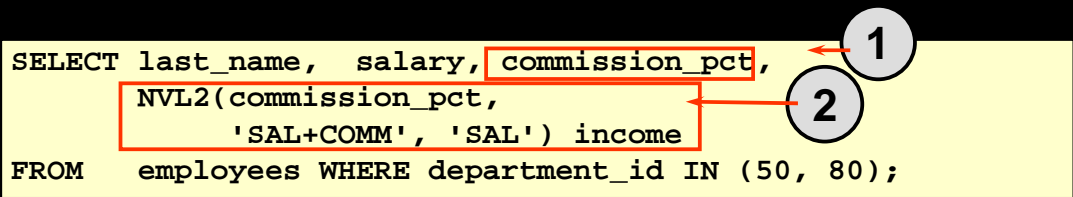
LAST_NAME	SALARY	COMMISSION_PCT	AN_SAL
Vargas	2500		
Zlotkey	10500	.2	151200
Abel	11000	.3	171600
Taylor	8600	.2	123840
Gietz	8300		

20 rows selected.

Notice that the annual compensation is calculated only for 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 on the slide, the NVL function is used to convert null values to zero.

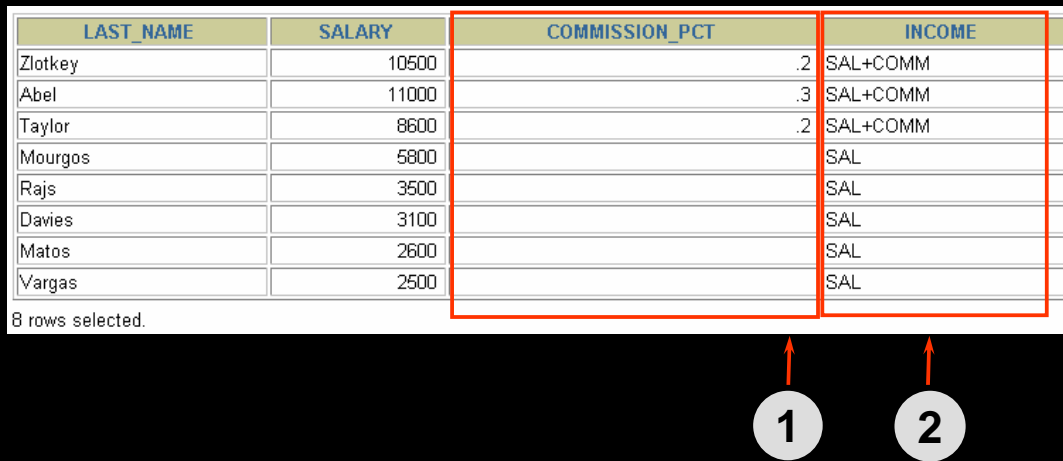
Using the NVL2 Function

```
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
Zlotkey	10500	.2	SAL+COMM
Abel	11000	.3	SAL+COMM
Taylor	8600	.2	SAL+COMM
Mourgos	5800		SAL
Rajs	3500		SAL
Davies	3100		SAL
Matos	2600		SAL
Vargas	2500		SAL

8 rows selected.



ORACLE

The NVL2 Function

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

Syntax

`NVL(expr1, expr2, expr3)`

In the syntax:

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

In the example shown, the COMMISSION_PCT column is examined. If a value is detected, the second expression of SAL+COMM is returned. If the COMMISSION_PCT column holds a null values, the third expression of SAL is returned.

The argument *expr1* can have any data type. The arguments *expr2* and *expr3* can have any data types except LONG. If the data types of *expr2* and *expr3* are different, The Oracle server converts *expr3* to the data type of *expr2* before comparing them unless *expr3* is a null constant. In that case, a data type conversion is not necessary.

The data type of the return value is always the same as the data type of *expr2*, unless *expr2* is character data, in which case the return value's data type is VARCHAR2.

Using the NULLIF Function

1

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

2

3

FIRST_NAME	expr1	LAST_NAME	expr2	RESULT
Steven	6	King	4	6
Neena	5	Kochhar	7	5
Lex	3	De Haan	7	3
Alexander	9	Hunold	6	9
Bruce	5	Ernst	5	
Diana	5	Lorentz	7	5
Kevin	5	Mourgos	7	5
Trenna	6	Rajs	4	6
Curtis	6	Davies	6	

...
20 rows selected.

1

2

3

ORACLE

3-49

Copyright © Oracle Corporation, 2001. All rights reserved.

The NULLIF Function

The **NULLIF** function compares two expressions. If they are equal, the function returns null. If they are not equal, the function returns the first expression. You cannot specify the literal NULL for first expression.

Syntax

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

In the syntax:

expr1 is the source value compared to *expr2*

expr2 is the source value compared with *expr1*. (If it is not equal to *expr1*, *expr1* is returned.)

In the example shown, the job ID in the EMPLOYEES table is compared to the job ID in the JOB_HISTORY table for any employee who is in both tables. The output shows the employee's current job. If the employee is listed more than once, that means the employee has held at least two jobs previously.

Note: The NULLIF function is logically equivalent to the following CASE expression. The CASE expression is discussed in a subsequent page:

```
CASE WHEN expr1 = expr 2 THEN NULL ELSE expr1 END
```

Using the COALESCE Function

- The advantage of the COALESCE function over the NVL function is that the COALESCE function can take multiple alternate values.
- If the first expression is not null, it returns that expression; otherwise, it does a COALESCE of the remaining expressions.

ORACLE

3-50

Copyright © Oracle Corporation, 2001. All rights reserved.

The COALESCE Function

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

Syntax

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

In the syntax:

<i>expr1</i>	returns this expression if it is not null
<i>expr2</i>	returns this expression if the first expression is null and this expression is not null
<i>exprn</i>	returns this expression if the preceding expressions are null

Using the COALESCE Function

```
SELECT    last_name,  
          COALESCE(commission_pct, salary, 10) comm  
FROM      employees  
ORDER BY  commission_pct;
```

LAST_NAME	COMM
Grant	.15
Zlotkey	.2
Taylor	.2
Abel	.3
King	24000
Kochhar	17000
De Haan	17000
Hunold	9000

20 rows selected.

ORACLE

The COALESCE Function

In the example shown, if the COMMISSION_PCT value is not null, it is shown. If the COMMISSION_PCT value is null, then the SALARY is shown. If the COMMISSION_PCT and SALARY values are null, then the value 10 is shown.

Conditional Expressions

- Provide the use of IF-THEN-ELSE logic within a SQL statement
- Use two methods:
 - CASE expression
 - DECODE function

ORACLE

3-52

Copyright © Oracle Corporation, 2001. All rights reserved.

Conditional Expressions

Two methods used to implement **conditional processing** (IF-THEN-ELSE logic) within a SQL statement are the CASE expression and the DECODE function.

Note: The **CASE expression** is new in the Oracle9i Server release. The CASE expression complies with ANSI SQL; **DECODE** is specific to Oracle syntax.

The 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
```

ORACLE

3-53

Copyright © Oracle Corporation, 2001. All rights reserved.

The CASE Expression

CASE expressions let you use IF-THEN-ELSE logic in SQL statements without having to invoke procedures.

In a simple CASE expression, Oracle 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 an ELSE clause exists, then Oracle returns *else_expr*. Otherwise, Oracle returns null. You cannot specify the literal NULL for all the *return_exprs* and the *else_expr*.

All of the expressions (*expr*, *comparison_expr*, and *return_expr*) must be of the same data type, which can be CHAR, VARCHAR2, NCHAR, or NVARCHAR2.

Instructor Note

There is also a searched CASE expression. Oracle searches from left to right until it finds an occurrence of a condition that is true, and then returns *return_expr*. If no condition is found to be true, and an ELSE clause exists, Oracle returns *else_expr*. Otherwise Oracle returns null. For more information, see *Oracle9i SQL Reference*, “Expressions.”

Using the CASE Expression

Facilitates conditional inquiries by doing the work of an IF-THEN-ELSE statement:

```
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
...			
Lorentz	IT_PROG	4200	4620
Mourgos	ST_MAN	5800	5800
Rajs	ST_CLERK	3500	4025
...			
Gietz	AC_ACCOUNT	8300	8300

20 rows selected.

ORACLE

Using the CASE Expression

In the preceding SQL statement, 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.

The DECODE Function

Facilitates conditional inquiries by doing the work of a CASE or IF-THEN-ELSE statement:

```
DECODE(col/expression, search1, result1  
      [, search2, result2, ..., ]  
      [, default])
```

ORACLE

3-55

Copyright © Oracle Corporation, 2001. All rights reserved.

The DECODE Function

The **DECODE function** decodes an expression in a way similar to the IF-THEN-ELSE logic 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 where 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
...			
Lorentz	IT_PROG	4200	4620
Mourgos	ST_MAN	5800	5800
Rajs	ST_CLERK	3500	4025
...			
Gietz	AC_ACCOUNT	8300	8300

20 rows selected.

ORACLE

3-56

Copyright © Oracle Corporation, 2001. All rights reserved.

Using the DECODE Function

In the preceding SQL statement, 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;
```

ORACLE

3-57

Copyright © Oracle Corporation, 2001. All rights reserved.

Example

This slide shows another example using the DECODE function. In this example, we determine the tax rate for each employee in department 80 based on the monthly salary. The tax rates are as per the values mentioned in the following data.

<i>Monthly Salary Range</i>	<i>Rate</i>
\$0.00 - 1999.99	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%

LAST_NAME	SALARY	TAX_RATE
Zlotkey	10500	.42
Abel	11000	.42
Taylor	8600	.4

Summary

In this lesson, you should have learned how to:

- **Perform calculations on data using functions**
- **Modify individual data items using functions**
- **Manipulate output for groups of rows using functions**
- **Alter date formats for display using functions**
- **Convert column data types using functions**
- **Use NVL functions**
- **Use IF-THEN-ELSE logic**

ORACLE

3-58

Copyright © Oracle Corporation, 2001. All rights reserved.

Single-Row Functions

Single-row functions can be nested to any level. Single-row functions can manipulate the following:

- Character data: LOWER, UPPER, INITCAP, CONCAT, SUBSTR, INSTR, LENGTH
- Number data: ROUND, TRUNC, MOD
- Date data: MONTHS_BETWEEN, ADD_MONTHS, NEXT_DAY, LAST_DAY, ROUND, TRUNC
- Date values can also use arithmetic operators.
- Conversion functions can convert character, date, and numeric values: TO_CHAR, TO_DATE, TO_NUMBER
- There are several functions that pertain to nulls, including NVL, NVL2, NULLIF, and COALESCE.
- IF-THEN-ELSE logic can be applied within a SQL statement by using the CASE expression or the DECODE function.

SYSDATE and DUAL

SYSDATE is a date function that returns the current date and time. It is customary to select SYSDATE from a dummy table called DUAL.

Practice 3, Part Two: Overview

This practice covers the following topics:

- **Creating queries that require the use of numeric, character, and date functions**
- **Using concatenation with functions**
- **Writing case-insensitive queries to test the usefulness of character functions**
- **Performing calculations of years and months of service for an employee**
- **Determining the review date for an employee**

ORACLE

3-59

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 3, Part Two: Overview

This practice is designed to give you a variety of exercises using different functions available for character, number, and date data types.

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

Instructor Note

This practice should be done in two parts. Part 1 contains questions 1-5 which cover the material from pages 1-23. Part 2 contains questions 6-14 and matches the material for the remainder of the lesson.

Practice question 6: Be sure to tell the students that their results may differ from the one provided, because `SYSDATE` is used in the exercise.

Instructor hint for practice question 10: The `ORDER BY` clause in the solution sorts on `TO_CHAR(hiredate-1, 'd')`. The format element 'd' returns a '1' for Sunday, '2' for Monday, and so forth. The expression `hiredate-1` effectively "shifts" each hiredate to the previous day, so that an employee hired on a Monday appears to have been hired on Sunday. The `TO_CHAR` function returns a '1' for that employee and the result set is sorted beginning with those employees hired on Monday.

Practice 3 - Part One (continued)

1. Write a query to display the current date. Label the column `Date`.

Date
28-SEP-01

2. For each employee, display the employee number, `last_name`, salary, and salary increased by 15% and expressed as a whole number. Label the column `New Salary`. Place your SQL statement in a text file named `lab3_2.sql`.
3. Run your query in the file `lab3_2.sql`.

EMPLOYEE_ID	LAST_NAME	SALARY	New Salary
100	King	24000	27600
101	Kochhar	17000	19550
102	De Haan	17000	19550
103	Hunold	9000	10350
...			
202	Fay	6000	6900
205	Higgins	12000	13800
206	Gietz	8300	9545

20 rows selected.

4. Modify your query `lab3_2.sql` to add a column that subtracts the old salary from the new salary. Label the column `Increase`. Save the contents of the file as `lab3_4.sql`. Run the revised query.

EMPLOYEE_ID	LAST_NAME	SALARY	New Salary	Increase
100	King	24000	27600	3600
101	Kochhar	17000	19550	2550
102	De Haan	17000	19550	2550
103	Hunold	9000	10350	1350
104	Ernst	6000	6900	900
107	Lorentz	4200	4830	630
124	Mourgos	5800	6670	870
141	Rajs	3500	4025	525
142	Davies	3100	3565	465
143	Matos	2600	2990	390
...				
201	Hartstein	13000	14950	1950
202	Fay	6000	6900	900
205	Higgins	12000	13800	1800
206	Gietz	8300	9545	1245

20 rows selected.

Practice 3, Part One (continued)

5. Write a query that displays the employee's last names with the first letter capitalized and all other letters lowercase, and the length of the names, for all employees whose name starts with J, A, or M. Give each column an appropriate label. Sort the results by the employees' last names.

Name	Length
Abel	4
Matos	5
Mourgos	7

Practice 3 - Part Two

6. For each employee, display the employee's last name, and calculate the number of months between today and the date the employee was hired. Label the column MONTHS_WORKED. Order your results by the number of months employed. Round the number of months up to the closest whole number.

Note: Your results will differ.

LAST_NAME	MONTHS_WORKED
Zlotkey	20
Mourgos	22
Grant	28
Lorentz	32
Vargas	39
Taylor	42
Matos	42
Fay	49
Davies	56
Abel	65
Hartstein	67
Rajs	71
Higgins	88
Gietz	88
LAST_NAME	MONTHS_WORKED
De Haan	105
Ernst	124
Hunold	141
Kochhar	144
Whalen	168
King	171

20 rows selected.

Practice 3 - Part Two (continued)

7. Write a query that produces the following for each employee:
 <employee last name> earns <salary> monthly but wants <3 times salary>. Label the column Dream Salaries.

Dream Salaries
King earns \$24,000.00 monthly but wants \$72,000.00.
Kochhar earns \$17,000.00 monthly but wants \$51,000.00.
De Haan earns \$17,000.00 monthly but wants \$51,000.00.
Hunold earns \$9,000.00 monthly but wants \$27,000.00.
Ernst earns \$6,000.00 monthly but wants \$18,000.00.
Lorentz earns \$4,200.00 monthly but wants \$12,600.00.
Mourgos earns \$5,800.00 monthly but wants \$17,400.00.
Rajs earns \$3,500.00 monthly but wants \$10,500.00.
Davies earns \$3,100.00 monthly but wants \$9,300.00.
Matos earns \$2,600.00 monthly but wants \$7,800.00.
Vargas earns \$2,500.00 monthly but wants \$7,500.00.
■ ■ ■
Gietz earns \$8,300.00 monthly but wants \$24,900.00.

20 rows selected.

If you have time, complete the following exercises:

8. Create a query to display the last name and salary for all employees. Format the salary to be 15 characters long, left-padded with \$. Label the column SALARY.

LAST_NAME	SALARY
King	\$\$\$\$\$\$\$\$\$24000
Kochhar	\$\$\$\$\$\$\$\$\$17000
De Haan	\$\$\$\$\$\$\$\$\$17000
Hunold	\$\$\$\$\$\$\$\$\$9000
Ernst	\$\$\$\$\$\$\$\$\$6000
Lorentz	\$\$\$\$\$\$\$\$\$4200
Mourgos	\$\$\$\$\$\$\$\$\$5800
Rajs	\$\$\$\$\$\$\$\$\$3500
■ ■ ■	
Higgins	\$\$\$\$\$\$\$\$\$12000
Gietz	\$\$\$\$\$\$\$\$\$8300

20 rows selected.

Practice 3 - Part Two (continued)

9. Display each employee's last name, hire date, and salary review date, which is the first Monday after six months of service. Label the column REVIEW. Format the dates to appear in the format similar to "Monday, the Thirty-First of July, 2000."

LAST_NAME	HIRE_DATE	REVIEW
King	17-JUN-87	Monday, the Twenty-First of December, 1987
Kochhar	21-SEP-89	Monday, the Twenty-Sixth of March, 1990
De Haan	13-JAN-93	Monday, the Nineteenth of July, 1993
Hunold	03-JAN-90	Monday, the Ninth of July, 1990
Ernst	21-MAY-91	Monday, the Twenty-Fifth of November, 1991
Lorentz	07-FEB-99	Monday, the Ninth of August, 1999
Mourgos	16-NOV-99	Monday, the Twenty-Second of May, 2000
Rajs	17-OCT-95	Monday, the Twenty-Second of April, 1996
Davies	29-JAN-97	Monday, the Fourth of August, 1997
■ ■ ■		
Gietz	07-JUN-94	Monday, the Twelfth of December, 1994

20 rows selected.

10. Display the last name, hire date, and day of the week on which the employee started. Label the column DAY. Order the results by the day of the week starting with Monday.

LAST_NAME	HIRE_DATE	DAY
Grant	24-MAY-99	MONDAY
Ernst	21-MAY-91	TUESDAY
Mourgos	16-NOV-99	TUESDAY
Taylor	24-MAR-98	TUESDAY
Rajs	17-OCT-95	TUESDAY
Gietz	07-JUN-94	TUESDAY
Higgins	07-JUN-94	TUESDAY
King	17-JUN-87	WEDNESDAY
De Haan	13-JAN-93	WEDNESDAY
■ ■ ■		
Abel	11-MAY-96	SATURDAY
Lorentz	07-FEB-99	SUNDAY
Fay	17-AUG-97	SUNDAY
Matos	15-MAR-98	SUNDAY

20 rows selected.

Practice 3 - Part Two (continued)

If you want an extra challenge, complete the following exercises:

11. Create a query that displays the employees’ last names and commission amounts. If an employee does not earn commission, put “No Commission.” Label the column COMM.

LAST_NAME	COMM
King	No Commission
Kochhar	No Commission
De Haan	No Commission
Hunold	No Commission
Ernst	No Commission
Lorentz	No Commission
Mourgos	No Commission
Rajs	No Commission
Davies	No Commission
Matos	No Commission
Vargas	No Commission
Zlotkey	.2
Abel	.3
Taylor	.2
■ ■ ■	
Gietz	No Commission

20 rows selected.

12. Create a query that displays the employees’ last names and indicates the amounts of their annual salaries with asterisks. Each asterisk signifies a thousand dollars. Sort the data in descending order of salary. Label the column EMPLOYEES_AND_THEIR_SALARIES.

EMPLOYEE_AND_THEIR_SALARIES	
King	*****
Kochhar	*****
De Haan	*****
Hartstei	*****
Higgins	*****
Abel	*****
■ ■ ■	
Vargas	**

20 rows selected.

Practice 3 - Part Two (continued)

13. Using the DECODE function, write a query that displays the grade of all employees based on the value of the column JOB_ID, as per the following data:

<i>Job</i>	<i>Grade</i>
AD_PRES	A
ST_MAN	B
IT_PROG	C
SA_REP	D
ST_CLERK	E
None of the above	0

JOB_ID	G
AD_PRES	A
AD_VP	0
AD_VP	0
IT_PROG	C
IT_PROG	C
IT_PROG	C
ST_MAN	B
ST_CLERK	E
ST_CLERK	E
ST_CLERK	E
■ ■ ■	
AC_MGR	0
AC_ACCOUNT	0

20 rows selected.

14. Rewrite the statement in the preceding question using the CASE syntax.

4

Displaying Data from Multiple Tables

ORACLE

Copyright © Oracle Corporation, 2001. All rights reserved.

Schedule:	Timing	Topic
	55 minutes	Lecture
	55 minutes	Practice
	110 minutes	Total

Objectives

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

- **Write `SELECT` statements to access data from more than one table using equality and nonequality joins**
- **View data that generally does not meet a join condition by using outer joins**
- **Join a table to itself by using a self join**

ORACLE

4-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Lesson Aim

This lesson covers how to obtain data from more than one table.

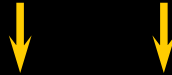
Obtaining Data from Multiple Tables

EMPLOYEES

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
100	King	90
101	Kochhar	90
...		
202	Fay	20
205	Higgins	110
206	Gietz	110

DEPARTMENTS

DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
10	Administration	1700
20	Marketing	1800
50	Shipping	1500
60	IT	1400
80	Sales	2500
90	Executive	1700
110	Accounting	1700
190	Contracting	1700



EMPLOYEE_ID	DEPARTMENT_ID	DEPARTMENT_NAME
200	10	Administration
201	20	Marketing
202	20	Marketing
...		
102	90	Executive
205	110	Accounting
206	110	Accounting

ORACLE

Data from Multiple Tables

Sometimes you need to use **data from more than one table**. In the slide example, the report displays data from two separate tables.

- Employee IDs exist in the EMPLOYEES table.
- Department IDs exist in both the EMPLOYEES and DEPARTMENTS tables.
- Location IDs exist in the DEPARTMENTS table.

To produce the report, you need to link the EMPLOYEES and DEPARTMENTS tables and access data from both of them.

Cartesian Products

- **A Cartesian product is formed when:**
 - A join condition is omitted
 - A join condition is invalid
 - All rows in the first table are joined to all rows in the second table
- **To avoid a Cartesian product, always include a valid join condition in a `WHERE` clause.**

ORACLE

4-4

Copyright © Oracle Corporation, 2001. All rights reserved.

Cartesian Products

When a join condition is invalid or omitted completely, the result is a *Cartesian product*, in which all combinations of rows are displayed. All rows in the first table are joined to all rows in the second table.

A **Cartesian product** tends to generate a large number of rows, and the result is rarely useful. You should always include a valid join condition in a `WHERE` clause, 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
100	King	90
101	Kochhar	90
...		
202	Fay	20
205	Higgins	110
206	Gietz	110

20 rows selected.

DEPARTMENTS (8 rows)

DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
10	Administration	1700
20	Marketing	1800
50	Shipping	1500
60	IT	1400
80	Sales	2500
90	Executive	1700
110	Accounting	1700
190	Contracting	1700

8 rows selected.

**Cartesian
product:
20x8=160 rows**

EMPLOYEE_ID	DEPARTMENT_ID	LOCATION_ID
100	90	1700
101	90	1700
102	90	1700
103	60	1700
104	60	1700
107	60	1700
...		

160 rows selected.

ORACLE

4-5

Copyright © Oracle Corporation, 2001. All rights reserved.

Cartesian Products (continued)

A **Cartesian product** is generated if a join condition is omitted. The example on the slide displays employee last name and department name from the EMPLOYEES and DEPARTMENTS tables. Because no WHERE clause has been 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.

```
SELECT last_name, department_name dept_name
FROM employees, departments;
```

LAST_NAME	DEPT_NAME
King	Administration
Kochhar	Administration
De Haan	Administration

...
160 rows selected.

Instructor Note

Demo: 4_cart.sql

Purpose: To illustrate executing a Cartesian product

Types of Joins

Oracle Proprietary Joins (8i and prior):

- Equijoin
- Non-equijoin
- Outer join
- Self join

SQL: 1999 Compliant Joins:

- Cross joins
- Natural joins
- Using clause
- Full or two sided outer joins
- Arbitrary join conditions for outer joins

ORACLE

4-6

Copyright © Oracle Corporation, 2001. All rights reserved.

Types of Joins

The Oracle9i database offers join syntax that is **SQL: 1999 compliant**. Prior to the 9i release, the join syntax was different from the ANSI standards. The new SQL: 1999 compliant join syntax does not offer any performance benefits over the Oracle proprietary join syntax that existed in prior releases.

Instructor Note

Do not get into details of all the types of joins now. Explain each join one by one, as is done in the following slides.

Joining Tables Using Oracle Syntax

Use a join to query data from more than one table.

```
SELECT    table1.column, table2.column
FROM      table1, table2
WHERE     table1.column1 = table2.column2;
```

- Write the join condition in the WHERE clause.
- Prefix the column name with the table name when the same column name appears in more than one table.

ORACLE

4-7

Copyright © Oracle Corporation, 2001. All rights reserved.

Defining Joins

When data from more than one table in the database is required, a *join condition* is used. Rows in one table can be joined to rows in another table according to common values existing in corresponding columns, that is, usually primary and foreign key columns.

To display data from two or more related tables, write a simple join condition in the WHERE clause.

In the syntax:

`table1.column` denotes the table and column from which data is retrieved
`table1.column1 = table2.column2` is the condition that joins (or relates) the tables together

Guidelines

- When writing a SELECT statement that joins tables, precede the column name with the table name for clarity and to enhance database access.
- If the same column name appears in more than one table, the column name must be prefixed with the table name.
- To join n tables together, you need a minimum of $n-1$ join conditions. For example, to join four tables, a minimum of three joins is required. This rule may not apply if your table has a concatenated primary key, in which case more than one column is required to uniquely identify each row.

For more information, see *Oracle9i SQL Reference*, “SELECT.”

What is an Equijoin?

EMPLOYEES

EMPLOYEE_ID	DEPARTMENT_ID
200	10
201	20
202	20
124	50
141	50
142	50
143	50
144	50
103	60
104	60
107	60
149	80
174	80
176	80

...

DEPARTMENTS

DEPARTMENT_ID	DEPARTMENT_NAME
10	Administration
20	Marketing
20	Marketing
50	Shipping
50	Shipping
50	Shipping
50	Shipping
50	Shipping
50	Shipping
50	Shipping
60	IT
60	IT
60	IT
60	IT
80	Sales
80	Sales
80	Sales

...

Foreign key Primary key

ORACLE

Equijoins

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 on both tables must be equal. Frequently, this type of join involves primary and foreign key complements.

Note: **Equijoins** are also called *simple joins* or *inner joins*.

Instructor Note

Explain the use of a decision matrix for simplifying writing joins. For example, if you want to display the name and department number of all the employees who are in the same department as Goyal, you can start by making the following decision tree:

Columns to Display	Originating Table	Condition
last_name	employees	last_name = 'Goyal'
department_name	departments	employees.department_id = departments.department_id

Now the SQL statement can be easily formulated by looking at the decision matrix. The first column gives the column list in the SELECT statement, the second column gives the tables for the FROM clause, and the third column gives the condition for the WHERE clause.

Retrieving Records with Equijoins

```
SELECT employees.employee_id, employees.last_name,  
       employees.department_id, departments.department_id,  
       departments.location_id  
FROM   employees, departments  
WHERE  employees.department_id = departments.department_id;
```

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_ID	LOCATION_ID
200	Whalen	10	10	1700
201	Hartstein	20	20	1800
202	Fay	20	20	1800
124	Mourgos	50	50	1500
141	Rajs	50	50	1500
142	Davies	50	50	1500
143	Matos	50	50	1500
144	Vargas	50	50	1500

19 rows selected.

ORACLE

4-9

Copyright © Oracle Corporation, 2001. All rights reserved.

Retrieving Records with Equijoins

In the slide example:

- The **SELECT** clause specifies the column names to retrieve:
 - employee last name, employee number, and department number, which are columns in the **EMPLOYEES** table
 - department number, department name, and location ID, which are columns in the **DEPARTMENTS** table
- The **FROM** clause specifies the two tables that the database must access:
 - **EMPLOYEES** table
 - **DEPARTMENTS** table
- The **WHERE** clause specifies how the tables are to be joined:

`EMPLOYEES.DEPARTMENT_ID = DEPARTMENTS.DEPARTMENT_ID`

Because the **DEPARTMENT_ID** column is common to both tables, it must be prefixed by the table name to avoid ambiguity.

Additional Search Conditions Using the AND Operator

EMPLOYEES

LAST_NAME	DEPARTMENT_ID
Whalen	10
Hartstein	20
Fay	20
Mourgos	50
Rajs	50
Davies	50
Matos	50
Vargas	50
Hunold	60
Ernst	60

...

DEPARTMENTS

DEPARTMENT_ID	DEPARTMENT_NAME
10	Administration
20	Marketing
20	Marketing
50	Shipping
50	Shipping
50	Shipping
50	Shipping
50	Shipping
50	Shipping
60	IT
60	IT

...

ORACLE

4-10

Copyright © Oracle Corporation, 2001. All rights reserved.

Additional Search Conditions

In addition to the join, you may have **criteria for your WHERE clause** to restrict the rows under consideration for one or more tables in the join. For example, to display employee Matos' department number and department name, you need an additional condition in the WHERE clause.

```
SELECT last_name, employees.department_id,  
       department_name  
FROM   employees, departments  
WHERE  employees.department_id = departments.department_id  
AND    last_name = 'Matos';
```

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
Matos	50	Shipping

Qualifying Ambiguous Column Names

- Use table prefixes to qualify column names that are in multiple tables.
- Improve performance by using table prefixes.
- Distinguish columns that have identical names but reside in different tables by using column aliases.

ORACLE

4-11

Copyright © Oracle Corporation, 2001. All rights reserved.

Qualifying Ambiguous Column Names

You need to qualify the names of the columns in the WHERE clause with the table name to avoid ambiguity. Without the **table prefixes**, the DEPARTMENT_ID column 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 improves performance, because you tell the Oracle Server exactly where to find the columns.

The requirement to qualify **ambiguous column names** is also applicable to columns that may be ambiguous in other clauses, such as the SELECT clause or the ORDER BY clause.

Instructor Note

Demo: 4_loc.sql

Purpose: To illustrate a SELECT clause with no aliases.

Using Table Aliases

- Simplify queries by using table aliases.
- Improve performance by using table prefixes.

```
SELECT e.employee_id, e.last_name, e.department_id,  
       d.department_id, d.location_id  
FROM   employees e, departments d  
WHERE  e.department_id = d.department_id;
```

ORACLE

4-12

Copyright © Oracle Corporation, 2001. All rights reserved.

Table Aliases

Qualifying column names with table names can be very time consuming, particularly if table names are lengthy. You can use *table aliases* instead of table names. 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.

Notice how **table aliases** are identified in the FROM clause in the example. The table name is specified in full, followed by a space and then the table alias. The EMPLOYEES table has been given an alias of e, and the DEPARTMENTS table has an alias of d.

Guidelines

- Table aliases can be up to 30 characters in length, but shorter is better.
- If a table alias is used for a particular table name in the FROM clause, then that table alias must be substituted for the table name throughout the SELECT statement.
- Table aliases should be meaningful.
- The table alias is valid only for the current SELECT statement.

Joining More than Two Tables

EMPLOYEES

LAST_NAME	DEPARTMENT_ID
King	90
Kochhar	90
De Haan	90
Hunold	60
Ernst	60
Lorentz	60
Mourgos	50
Rajs	50
Davies	50
Matos	50
Vargas	50
Zlotkey	80
Abel	80
Taylor	80

■■■
20 rows selected.

DEPARTMENTS

DEPARTMENT_ID	LOCATION_ID
10	1700
20	1800
50	1500
60	1400
80	2500
90	1700
110	1700
190	1700

8 rows selected.

LOCATIONS

LOCATION_ID	CITY
1400	Southlake
1500	South San Francisco
1700	Seattle
1800	Toronto
2500	Oxford

- To join n tables together, you need a minimum of $n-1$ join conditions. For example, to join three tables, a minimum of two joins is required.

ORACLE

Additional Search Conditions

Sometimes you may need to **join more than two tables**. For example, to display the last name, the department name, and the city for each employee, you have to join the EMPLOYEES, DEPARTMENTS, and LOCATIONS tables.

```
SELECT e.last_name, d.department_name, l.city
FROM   employees e, departments d, locations l
WHERE  e.department_id = d.department_id
AND    d.location_id = l.location_id;
```

LAST_NAME	DEPARTMENT_NAME	CITY
Hunold	IT	Southlake
Ernst	IT	Southlake
Lorentz	IT	Southlake
Mourgos	Shipping	South San Francisco
Rajs	Shipping	South San Francisco
Davies	Shipping	South San Francisco

■■■
19 rows selected.

Non-Equi Joins

EMPLOYEES

LAST_NAME	SALARY
King	24000
Kochhar	17000
De Haan	17000
Hunold	9000
Ernst	6000
Lorentz	4200
Mourgos	5800
Rajs	3500
Davies	3100
Matos	2600
Vargas	2500
Zlotkey	10500
Abel	11000
Taylor	8600

20 rows selected.

JOB_GRADES

GRA	LOWEST_SAL	HIGHEST_SAL
A	1000	2999
B	3000	5999
C	6000	9999
D	10000	14999
E	15000	24999
F	25000	40000

Salary in the EMPLOYEES table must be between lowest salary and highest salary in the JOB_GRADES table.

Non-Equi Joins

A **non-equi join** is a join condition containing something other than an equality operator.

The relationship between the EMPLOYEES table and the JOB_GRADES table has an example of a non-equi join. A relationship between the two tables is that the SALARY column in the EMPLOYEES table must be between the values in the LOWEST_SALARY and HIGHEST_SALARY columns of the JOB_GRADES table. The relationship is obtained using an operator other than equals (=).

Retrieving Records with Non-Equi Joins

```
SELECT e.last_name, e.salary, j.grade_level
FROM   employees e, job_grades j
WHERE  e.salary
      BETWEEN j.lowest_sal AND j.highest_sal;
```

LAST_NAME	SALARY	GRA
Matos	2600	A
Vargas	2500	A
Lorentz	4200	B
Mourgos	5800	B
Rajs	3500	B
Davies	3100	B
Whalen	4400	B
Hunold	9000	C
Ernst	6000	C

20 rows selected.

ORACLE

4-15

Copyright © Oracle Corporation, 2001. All rights reserved.

Non-Equi Joins (continued)

The slide example creates a **non-equi join** 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 grade 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 \leq and \geq can be used, but BETWEEN is the simplest. Remember to specify the low value first and the high value last when using BETWEEN.

Table aliases have been specified in the slide example for performance reasons, not because of possible ambiguity.

Instructor Note

Explain that BETWEEN ... AND ... is actually translated by the Oracle server to a pair of AND conditions ($a \geq \text{lower limit}$) and ($a \leq \text{higher limit}$) and IN (...) is translated by the Oracle server to a set of OR conditions ($a = \text{value1 OR } a = \text{value2 OR } a = \text{value3}$). So using BETWEEN ... AND ... , IN(...) has no performance benefits; the benefit is logical simplicity.

Outer Joins

DEPARTMENTS

DEPARTMENT_NAME	DEPARTMENT_ID
Administration	10
Marketing	20
Shipping	50
IT	60
Sales	80
Executive	90
Accounting	110
Contracting	190

8 rows selected.

EMPLOYEES

DEPARTMENT_ID	LAST_NAME
90	King
90	Kochhar
90	De Haan
60	Hunold
60	Ernst
60	Lorentz
50	Mourgos
50	Rajs
50	Davies
50	Matos
50	Vargas
80	Zlotkey

20 rows selected.



There are no employees in department 190.

ORACLE

Returning Records with No Direct Match with Outer Joins

If a row does not satisfy a join condition, the row will not appear in the query result. For example, in the equijoin condition of EMPLOYEES and DEPARTMENTS tables, employee Grant does not appear because there is no department ID recorded for her in the EMPLOYEES table. Instead of seeing 20 employees in the result set, you see 19 records.

```
SELECT e.last_name, e.department_id, d.department_name
FROM   employees e, departments d
WHERE  e.department_id = d.department_id;
```

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
Whalen	10	Administration
Hartstein	20	Marketing
Fay	20	Marketing
Mourgos	50	Shipping

19 rows selected.

Outer Joins Syntax

- You use an outer join to also see rows that do not meet the join condition.
- The Outer join operator is the plus sign (+).

```
SELECT table1.column, table2.column
FROM   table1, table2
WHERE  table1.column(+) = table2.column;
```

```
SELECT table1.column, table2.column
FROM   table1, table2
WHERE  table1.column = table2.column(+);
```

ORACLE

4-17

Copyright © Oracle Corporation, 2001. All rights reserved.

Using Outer Joins to Return Records with No Direct Match

The missing rows can be returned if an *outer join* operator is used in the join condition. The operator is a plus sign enclosed in parentheses (+), and it is *placed on the “side” of the join that is deficient in information*. This operator has the effect of creating one or more null rows, to which one or more rows from the nondeficient table can be joined.

In the syntax:

<code>table1.column =</code>	is the condition that joins (or relates) the tables together.
<code>table2.column (+)</code>	is the outer join symbol, which can be placed on either side of the WHERE clause condition, but not on both sides. (Place the outer join symbol following the name of the column in the table without the matching rows.)

Instructor Note

Demo: 4_ejoin.sql

Purpose: To illustrate an equijoin leading to an outer join.

Using Outer Joins

```
SELECT e.last_name, e.department_id, d.department_name
FROM   employees e, departments d
WHERE  e.department_id(+) = d.department_id ;
```

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
Whalen	10	Administration
Hartstein	20	Marketing
Fay	20	Marketing
Mourgos	50	Shipping
Rajs	50	Shipping
Davies	50	Shipping
Matos	50	Shipping
...		
Gietz	110	Accounting
		Contracting

20 rows selected.

ORACLE

4-18

Copyright © Oracle Corporation, 2001. All rights reserved.

Using Outer Joins to Return Records with No Direct Match (continued)

The slide example displays employee last names, department ID's and department names. The Contracting department does not have any employees. The empty value is shown in the output shown.

Outer Join Restrictions

- The **outer join** operator can appear on only *one* side of the expression—the side that has information missing. It returns those rows from one table that have no direct match in the other table.
- A condition involving an outer join cannot use the **IN** operator or be linked to another condition by the **OR** operator.

Instructor Note

The **UNION** operator works around the issue of being able to use an outer join operator on one side of the expression. The ANSI full outer join also allows you to have an outer join on both sides of the expression. It is discussed later in this lesson.

Demo: 4_ojoin.sql

Purpose: To illustrate an outer join.

Self Joins

EMPLOYEES (WORKER)

EMPLOYEE_ID	LAST_NAME	MANAGER_ID
100	King	
101	Kochhar	100
102	De Haan	100
103	Hunold	102
104	Ernst	103
107	Lorentz	103
124	Mourgos	100

...

EMPLOYEES (MANAGER)

EMPLOYEE_ID	LAST_NAME
100	King
101	Kochhar
102	De Haan
103	Hunold
104	Ernst
107	Lorentz
124	Mourgos

...



**MANAGER_ID in the WORKER table is equal to
EMPLOYEE_ID in the MANAGER table.**

ORACLE

Joining a Table to Itself

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 Whalen's manager, you need to:

- Find Whalen in the EMPLOYEES table by looking at the LAST_NAME column.
- Find the manager number for Whalen by looking at the MANAGER_ID column. Whalen's manager number is 101.
- Find the name of the manager with EMPLOYEE_ID 101 by looking at the LAST_NAME column. Kochhar's employee number is 101, so Kochhar is Whalen's manager.

In this process, you look in the table twice. The first time you look in the table to find Whalen in the LAST_NAME column and MANAGER_ID value of 101. The second time you look in the EMPLOYEE_ID column to find 101 and the LAST_NAME column to find Kochhar.

Instructor Note

Show the data from the EMPLOYEES table and point out how each manager is also an employee.

Joining a Table to Itself

```
SELECT worker.last_name || ' works for ' || manager.last_name
FROM   employees worker, employees manager
WHERE  worker.manager_id = manager.employee_id ;
```

WORKER.LAST_NAME 'WORKSFOR' MANAGER.LAST_NAME
Kochhar works for King
De Haan works for King
Mourgos works for King
Zlotkey works for King
Hartstein works for King
Whalen works for Kochhar
Higgins works for Kochhar
Hunold works for De Haan
Ernst works for Hunold
...
19 rows selected.

ORACLE

4-20

Copyright © Oracle Corporation, 2001. All rights reserved.

Joining a Table to Itself (continued)

The slide example joins the EMPLOYEES table to itself. To simulate two tables in the FROM clause, there are two aliases, namely w and m, for the same table, EMPLOYEES.

In this example, the WHERE clause contains the join that means “where a worker’s manager number matches the employee number for the manager.”

Instructor Note

Point out the following to the students:

- The column heading in the result of the query on the slide seems meaningless. A meaningful column alias should have been used instead.
- There are only 19 rows in the output, but there are 20 rows in the EMPLOYEES table. This occurs because employee King, who is the president, does not have a manager.

Practice 4, Part One: Overview

This practice covers writing queries to join tables together using Oracle syntax.

ORACLE

4-21

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 4, Part One

This practice is designed to give you a variety of exercises that join tables together using the Oracle syntax shown in the lesson so far.

Complete practice questions 1- 4 at the end of this lesson.

Joining Tables Using SQL: 1999 Syntax

Use a join to query data from more than one table.

```
SELECT    table1.column, table2.column
FROM      table1
[CROSS JOIN table2] |
[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)];
```

ORACLE

4-22

Copyright © Oracle Corporation, 2001. All rights reserved.

Defining Joins

Using the **SQL: 1999 syntax**, you can obtain the same results as were shown in the prior pages.

In the syntax:

<code>table1.column</code>	Denotes the table and column from which data is retrieved
<code>CROSS JOIN</code>	Returns a Cartesian product from the two tables
<code>NATURAL JOIN</code>	Joins two tables based on the same column name
<code>JOIN table</code>	
<code>USING column_name</code>	Performs an equijoin based on the column name
<code>JOIN table ON</code>	
<code>table1.column_name</code> <code>= table2.column_name</code>	Performs an equijoin based on the condition in the ON clause
<code>LEFT/RIGHT/FULL OUTER</code>	

For more information, see *Oracle9i SQL Reference*, “SELECT.”

Creating Cross Joins

- The **CROSS JOIN** clause produces the cross-product of two tables.
- This is the same as a Cartesian product between the two tables.

```
SELECT last_name, department_name
FROM   employees
CROSS JOIN departments ;
```

LAST_NAME	DEPARTMENT_NAME
King	Administration
Kochhar	Administration
De Haan	Administration
Hunold	Administration

160 rows selected.

ORACLE

4-23

Copyright © Oracle Corporation, 2001. All rights reserved.

Creating Cross Joins

The example on the slide gives the same results as the following:

```
SELECT last_name, department_name
FROM   employees, departments;
```

LAST_NAME	DEPARTMENT_NAME
King	Administration
Kochhar	Administration
De Haan	Administration
Hunold	Administration
Ernst	Administration

160 rows selected.

Creating Natural Joins

- The **NATURAL JOIN** clause is based on all columns in the two tables that have the same name.
- 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.

ORACLE

4-24

Copyright © Oracle Corporation, 2001. All rights reserved.

Creating Natural Joins

It was not possible to do a join without explicitly specifying the columns in the corresponding tables in prior releases of Oracle. In Oracle9i it is possible to let the join be completed automatically based on columns in the two tables which have matching data types and names, using the keywords **NATURAL JOIN** keywords.

Note: The join can happen only on columns having the same names and data types in both the tables. If the columns have the same name, but different data types, then the **NATURAL JOIN** syntax causes an error.

Retrieving Records with Natural Joins

```
SELECT department_id, department_name,  
       location_id, city  
FROM   departments  
NATURAL JOIN locations ;
```

DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID	CITY
60	IT	1400	Southlake
50	Shipping	1500	South San Francisco
10	Administration	1700	Seattle
90	Executive	1700	Seattle
110	Accounting	1700	Seattle
190	Contracting	1700	Seattle
20	Marketing	1800	Toronto
80	Sales	2500	Oxford

8 rows selected.

ORACLE

4-25

Copyright © Oracle Corporation, 2001. All rights reserved.

Retrieving Records with Natural Joins

In the example on the slide, the LOCATIONS table is joined to the DEPARTMENT table by the LOCATION_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.

Equijoins

The natural join can also be written as an equijoin:

```
SELECT department_id, department_name,  
       departments.location_id, city  
FROM   departments, locations  
WHERE  departments.location_id = locations.location_id;
```

Natural Joins with a WHERE Clause

Additional restrictions on a natural join are implemented by using a WHERE clause. The example below 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, the `NATURAL JOIN` clause can be modified with the `USING` clause to specify the columns that should be used for an equijoin.
- Use the `USING` clause to match only one column when more than one column matches.
- Do not use a table name or alias in the referenced columns.
- The `NATURAL JOIN` and `USING` clauses are mutually exclusive.

ORACLE

4-26

Copyright © Oracle Corporation, 2001. All rights reserved.

The USING Clause

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. The columns referenced in the `USING` clause should not have a qualifier (table name or alias) anywhere in the SQL statement.

For example, this statement is valid:

```
SELECT l.city, d.department_name
FROM   locations l JOIN departments d USING (location_id)
WHERE  location_id = 1400;
```

This statement is invalid because the `LOCATION_ID` is qualified in the `WHERE` clause:

```
SELECT l.city, d.department_name
FROM   locations l JOIN departments d USING (location_id)
WHERE  d.location_id = 1400;
ORA-25154: column part of USING clause cannot have qualifier
```

The same restriction applies to `NATURAL` joins also. Therefore columns that have the same name in both tables have to be used without any qualifiers.

Retrieving Records with the USING Clause

```
SELECT e.employee_id, e.last_name, d.location_id
FROM   employees e JOIN departments d
      USING (department_id) ;
```

EMPLOYEE_ID	LAST_NAME	LOCATION_ID
200	Whalen	1700
201	Hartstein	1800
202	Fay	1800
124	Mourgos	1500
141	Rajs	1500
142	Davies	1500
143	Matos	1500
144	Vargas	1500
103	Hunold	1400

19 rows selected.

ORACLE

The USING Clause (continued)

The example shown joins the DEPARTMENT_ID column in the EMPLOYEES and DEPARTMENTS tables, and thus shows the location where an employee works.

This can also be written as an **equijoin**:

```
SELECT employee_id, last_name,
       employees.department_id, location_id
FROM   employees, departments
WHERE  employees.department_id = departments.department_id;
```

Creating Joins with the ON Clause

- The join condition for the natural join is basically an equijoin of all columns with the same name.
- To specify arbitrary conditions or specify columns to join, the ON clause is used.
- The join condition is separated from other *search* conditions.
- The ON clause makes code easy to understand.

ORACLE

4-28

Copyright © Oracle Corporation, 2001. All rights reserved.

The ON Condition

Use the **ON clause** to specify a join condition. This lets you 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	LOCATION_ID
200	Whalen	10	10	1700
201	Hartstein	20	20	1800
202	Fay	20	20	1800
124	Mourgos	50	50	1500
141	Rajs	50	50	1500
142	Davies	50	50	1500
143	Matos	50	50	1500

...

19 rows selected.

ORACLE

Creating Joins with the ON Clause

The **ON clause** can also be used as follows to join columns that have different names:

```
SELECT e.last_name emp, m.last_name mgr  
FROM   employees e JOIN employees m  
ON     (e.manager_id = m.employee_id);
```

EMP	MGR
Kochhar	King
De Haan	King
Mourgos	King
Zlotkey	King
Hartstein	King
Whalen	Kochhar

...

19 rows selected.

The preceding example is a selfjoin of the EMPLOYEE table to itself, based on the EMPLOYEE_ID and MANAGER_ID columns.

Creating Three-Way Joins with the ON Clause

```
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
103	Southlake	IT
104	Southlake	IT
107	Southlake	IT
124	South San Francisco	Shipping
141	South San Francisco	Shipping
142	South San Francisco	Shipping
143	South San Francisco	Shipping
144	South San Francisco	Shipping

...

19 rows selected.

ORACLE

4-30

Copyright © Oracle Corporation, 2001. All rights reserved.

Three-Way Joins

A **three-way join** is a join of three tables. In **SQL: 1999 compliant syntax**, joins are performed from left to right so 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.

This can also be written as a three-way equijoin:

```
SELECT employee_id, city, department_name
FROM   employees, departments, locations
WHERE  employees.department_id = departments.department_id
AND    departments.location_id = locations.location_id;
```

Instructor Note

The example shown 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);
```

INNER Versus OUTER Joins

- In SQL: 1999, the join of two tables returning only matched rows is an inner join.
- A join between two tables that returns the results of the inner join as well as unmatched rows left (or right) tables is 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

4-31

Copyright © Oracle Corporation, 2001. All rights reserved.

Joins - Comparing SQL: 1999 to Oracle Syntax

Oracle	SQL: 1999
Equi-Join	Natural/Inner Join
Outer-Join	Left Outer Join
Self-Join	Join ON
Non-Equi-Join	Join USING
Cartesian Product	Cross Join

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
Whalen	10	Administration
Fay	20	Marketing
Hartstein	20	Marketing
...		
De Haan	90	Executive
Kochhar	90	Executive
King	90	Executive
Gietz	110	Accounting
Higgins	110	Accounting
Grant		

20 rows selected.

ORACLE

Example of LEFT OUTER JOIN

This query retrieves all rows in the EMPLOYEES table, which is the **left table** even if there is no match in the DEPARTMENTS table.

This query was completed in earlier releases as follows:

```
SELECT e.last_name, e.department_id, d.department_name
FROM   employees e, departments d
WHERE  d.department_id (+) = e.department_id;
```

RIGHT OUTER JOIN

```
SELECT e.last_name, e.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
King	90	Executive
Kochhar	90	Executive
...		
Whalen	10	Administration
Hartstein	20	Marketing
Fay	20	Marketing
Higgins	110	Accounting
Gietz	110	Accounting
		Contracting

20 rows selected.

ORACLE

Example of RIGHT OUTER JOIN

This query retrieves all rows in the DEPARTMENTS table, which is the **right table** even if there is no match in the EMPLOYEES table.

This query was completed in earlier releases as follows:

```
SELECT e.last_name, e.department_id, d.department_name
FROM   employees e, departments d
WHERE  d.department_id = e.department_id (+);
```

FULL OUTER JOIN

```
SELECT e.last_name, e.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
Whalen	10	Administration
Fay	20	Marketing
...		
De Haan	90	Executive
Kochhar	90	Executive
King	90	Executive
Gietz	110	Accounting
Higgins	110	Accounting
Grant		
		Contracting

21 rows selected.

ORACLE

Example of FULL OUTER JOIN

This query retrieves all rows in the EMPLOYEES table, **even if there is no match** in the DEPARTMENTS table. It also retrieves all rows in the DEPARTMENTS table, **even if there is no match** in the EMPLOYEES table.

Instructor Note

It was not possible to complete this in earlier releases using outer joins. However, you could accomplish the same results using the UNION operator.

```
SELECT e.last_name, e.department_id, d.department_name
FROM   employees e, departments d
WHERE  e.department_id (+) = d.department_id
UNION
SELECT e.last_name, e.department_id, d.department_name
FROM   employees e, departments d
WHERE  e.department_id = d.department_id (+);
```

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 ;
```

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_ID	LOCATION_ID
174	Abel	80	80	2500
176	Taylor	80	80	2500

ORACLE

Applying Additional Conditions

You can apply additional conditions in the WHERE clause. The example shown performs a join on the EMPLOYEES and DEPARTMENTS tables, and, in addition, displays only employees with a manager ID equal to 149.

Summary

In this lesson, you should have learned how to use joins to display data from multiple tables in:

- **Oracle proprietary syntax for versions 8i and earlier**
- **SQL: 1999 compliant syntax for version 9i**

ORACLE

4-36

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary

There are multiple ways to join tables.

Types of Joins

- Equijoins
- Non-equijoins
- Outer joins
- Self joins
- Cross joins
- Natural joins
- Full or outer joins

Cartesian Products

A Cartesian product results in all combinations of rows displayed. This is done by either omitting the WHERE clause or specifying the CROSS JOIN clause.

Table Aliases

- Table aliases speed up database access.
- Table aliases can help to keep SQL code smaller, by conserving memory.

Practice 4, Part Two: Overview

This practice covers the following topics:

- **Joining tables using an equijoin**
- **Performing outer and self joins**
- **Adding conditions**

ORACLE

4-37

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 4, Part Two

This practice is intended to give you practical experience in extracting data from more than one table. Try using both the Oracle proprietary syntax and the SQL: 1999 compliant syntax.

In Part Two, questions 5-8, try writing the join statements using ANSI syntax.

In Part Two, questions 9-11, try writing the join statements using both the Oracle syntax and the ANSI syntax.

Practice 4 - Part One

1. Write a query to display the last name, department number, and department name for all employees.

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
Whalen	10	Administration
Hartstein	20	Marketing
Fay	20	Marketing
Mourgos	50	Shipping
Rajs	50	Shipping
Davies	50	Shipping
Matos	50	Shipping
Vargas	50	Shipping
Hunold	60	IT
Ernst	60	IT
Lorentz	60	IT
Zlotkey	80	Sales
Abel	80	Sales

■ ■ ■

19 rows selected.

2. Create a unique listing of all jobs that are in department 80. Include the location of the department in the output.

JOB_ID	LOCATION_ID
SA_MAN	2500
SA_REP	2500

3. Write a query to display the employee last name, department name, location ID, and city of all employees who earn a commission.

LAST_NAME	DEPARTMENT_NAME	LOCATION_ID	CITY
Zlotkey	Sales	2500	Oxford
Abel	Sales	2500	Oxford
Taylor	Sales	2500	Oxford

Practice 4 - Part One (continued)

4. Display the employee last name and department name for all employees who have an *a* (lowercase) in their last names. Place your SQL statement in a text file named `lab4_4.sql`.

LAST_NAME	DEPARTMENT_NAME
Whalen	Administration
Hartstein	Marketing
Fay	Marketing
Rajs	Shipping
Davies	Shipping
Matos	Shipping
Vargas	Shipping
Taylor	Sales
Kochhar	Executive
De Haan	Executive

10 rows selected.

Practice 4 - Part Two

- Write a query to display the last name, job, department number, and department name for all employees who work in Toronto.

LAST_NAME	JOB_ID	DEPARTMENT_ID	DEPARTMENT_NAME
Hartstein	MK_MAN	20	Marketing
Fay	MK_REP	20	Marketing

- Display the employee last name and employee number along with their manager's last name and manager number. Label the columns Employee, Emp#, Manager, and Mgr#, respectively.

Place your SQL statement in a text file named lab4_6.sql.

Employee	EMP#	Manager	Mgr#
Kochhar	101	King	100
De Haan	102	King	100
Mourgos	124	King	100
Zlotkey	149	King	100
Hartstein	201	King	100
Whalen	200	Kochhar	101
Higgins	205	Kochhar	101
Hunold	103	De Haan	102
Ernst	104	Hunold	103
Lorentz	107	Hunold	103
Rajs	141	Mourgos	124
Davies	142	Mourgos	124
Matos	143	Mourgos	124
Vargas	144	Mourgos	124
Employee	EMP#	Manager	Mgr#
Abel	174	Zlotkey	149
Taylor	176	Zlotkey	149
Grant	178	Zlotkey	149
Fay	202	Hartstein	201
Gietz	206	Higgins	205

19 rows selected.

Practice 4 - Part Two (continued)

7. Modify `lab4_6.sql` to display all employees including King, who has no manager. Order the results by the employee number.
Place your SQL statement in a text file named `lab4_7.sql`. Run the query in `lab4_7.sql`.

Employee	EMP#	Manager	Mgr#
King	100		
Kochhar	101	King	100
De Haan	102	King	100
Hunold	103	De Haan	102
Ernst	104	Hunold	103
Lorentz	107	Hunold	103
Mourgos	124	King	100

■ ■ ■

20 rows selected.

If you have time, complete the following exercises:

8. Create a query that displays employee last names, department numbers, and all the employees who work in the same department as a given employee. Give each column an appropriate label.

DEPARTMENT	EMPLOYEE	COLLEAGUE
20	Fay	Hartstein
20	Hartstein	Fay
50	Davies	Matos
50	Davies	Mourgos
50	Davies	Rajs
50	Davies	Vargas
50	Matos	Davies
50	Matos	Mourgos
50	Matos	Rajs
50	Matos	Vargas
50	Mourgos	Davies
50	Mourgos	Matos
50	Mourgos	Rajs
50	Mourgos	Vargas

■ ■ ■

42 rows selected.

Practice 4 - Part Two (continued)

9. Show the structure of the JOB_GRADES table. Create a query that displays the name, job, department name, salary, and grade for all employees.

Name	Null?	Type
GRADE_LEVEL		VARCHAR2(3)
LOWEST_SAL		NUMBER
HIGHEST_SAL		NUMBER

LAST_NAME	JOB_ID	DEPARTMENT_NAME	SALARY	GRA
Matos	ST_CLERK	Shipping	2600	A
Vargas	ST_CLERK	Shipping	2500	A
Lorentz	IT_PROG	IT	4200	B
Mourgos	ST_MAN	Shipping	5800	B
Rajs	ST_CLERK	Shipping	3500	B
Davies	ST_CLERK	Shipping	3100	B
Whalen	AD_ASST	Administration	4400	B

■ ■ ■

19 rows selected.

If you want an extra challenge, complete the following exercises:

10. Create a query to display the name and hire date of any employee hired after employee Davies.

LAST_NAME	HIRE_DATE
Lorentz	07-FEB-99
Mourgos	16-NOV-99
Matos	15-MAR-98
Vargas	09-JUL-98
Zlotkey	29-JAN-00
Taylor	24-MAR-98
Grant	24-MAY-99
Fay	17-AUG-97

8 rows selected.

Practice 4 - Part Two (continued)

11. Display the names and hire dates for all employees who were hired before their managers, along with their manager's names and hire dates. Label the columns Employee, Emp Hired, Manager, and Mgr Hired, respectively.

LAST_NAME	HIRE_DATE	LAST_NAME	HIRE_DATE
Whalen	17-SEP-87	Kochhar	21-SEP-89
Hunold	03-JAN-90	De Haan	13-JAN-93
Rajs	17-OCT-95	Mourgos	16-NOV-99
Davies	29-JAN-97	Mourgos	16-NOV-99
Matos	15-MAR-98	Mourgos	16-NOV-99
Vargas	09-JUL-98	Mourgos	16-NOV-99
Abel	11-MAY-96	Zlotkey	29-JAN-00
Taylor	24-MAR-98	Zlotkey	29-JAN-00
Grant	24-MAY-99	Zlotkey	29-JAN-00

9 rows selected.

5

Aggregating Data Using Group Functions

ORACLE

Copyright © Oracle Corporation, 2001. All rights reserved.

Schedule:

Timing

35 minutes

40 minutes

75 minutes

Topic

Lecture

Practice

Total

Objectives

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

- **Identify the available group functions**
- **Describe the use of group functions**
- **Group data using the `GROUP BY` clause**
- **Include or exclude grouped rows by using the `HAVING` clause**

ORACLE

5-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Lesson Aim

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.

What Are Group Functions?

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

EMPLOYEES

DEPARTMENT_ID	SALARY
90	24000
90	17000
90	17000
60	9000
60	6000
60	4200
50	5800
50	3500
50	3100
50	2600
50	2500
80	10500
80	11000
80	8600
	7000
10	4400

...
20 rows selected.

The maximum
salary in
the EMPLOYEES
table.

MAX(SALARY)
24000

ORACLE

Group Functions

Unlike single-row functions, **group functions** operate on **sets of rows** to give one result per group. These sets may be the whole table or the table split into groups.

Types of Group Functions

- **AVG**
- **COUNT**
- **MAX**
- **MIN**
- **STDDEV**
- **SUM**
- **VARIANCE**

ORACLE

5-4

Copyright © Oracle Corporation, 2001. All rights reserved.

Group Functions (continued)

Each of the functions accepts an argument. The following table identifies the options that you can use in the syntax:

Function	Description
AVG ([DISTINCT <u>ALL</u>] <i>n</i>)	Average value of <i>n</i> , ignoring null values
COUNT ({ * [DISTINCT <u>ALL</u>] <i>expr</i> })	Number of rows, where <i>expr</i> evaluates to something other than null (count all selected rows using *, including duplicates and rows with nulls)
MAX ([DISTINCT <u>ALL</u>] <i>expr</i>)	Maximum value of <i>expr</i> , ignoring null values
MIN ([DISTINCT <u>ALL</u>] <i>expr</i>)	Minimum value of <i>expr</i> , ignoring null values
STDDEV ([DISTINCT <u>ALL</u>] <i>x</i>)	Standard deviation of <i>n</i> , ignoring null values
SUM ([DISTINCT <u>ALL</u>] <i>n</i>)	Sum values of <i>n</i> , ignoring null values
VARIANCE ([DISTINCT <u>ALL</u>] <i>x</i>)	Variance of <i>n</i> , ignoring null values

Group Functions Syntax

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

ORACLE

5-5

Copyright © Oracle Corporation, 2001. All rights reserved.

Guidelines for Using Group Functions

- **DISTINCT** makes the function consider only nonduplicate values; **ALL** makes it consider every value including duplicates. The default is **ALL** and therefore does not need to be specified.
- 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**, or **COALESCE** functions.
- The Oracle server implicitly sorts the result set in ascending order when using a **GROUP BY clause**. To override this default ordering, **DESC** can be used in an **ORDER BY** clause.

Instructor Note

Stress the use of **DISTINCT** and group functions ignoring null values. **ALL** is the default and is very rarely specified.

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)
8150	11000	6000	32600

ORACLE

Group Functions

You can use **AVG**, **SUM**, **MIN**, and **MAX** functions against columns that can store numeric data. The example on the slide displays the average, highest, lowest, and sum of monthly salaries for all sales representatives.

Using the MIN and MAX Functions

You can use MIN and MAX for any data type.

```
SELECT MIN(hire_date), MAX(hire_date)
FROM   employees;
```

MIN(HIRE_	MAX(HIRE_
17-JUN-87	29-JAN-00

ORACLE

5-7

Copyright © Oracle Corporation, 2001. All rights reserved.

Group Functions (continued)

You can use the MAX and MIN functions for any data type. The slide example displays the most junior and most senior employee.

The following example displays the employee last name that is first and the employee last name that is the last in an alphabetized list of all employees.

```
SELECT MIN(last_name), MAX(last_name)
FROM   employees;
```

MIN(LAST_NAME)	MAX(LAST_NAME)
Abel	Zlotkey

Note: AVG, SUM, VARIANCE, and STDDEV functions can be used only with numeric data types.

Using the COUNT Function

COUNT (*) returns the number of rows in a table.

```
SELECT COUNT ( * )  
FROM   employees  
WHERE  department_id = 50;
```

COUNT(*)
5

ORACLE

5-8

Copyright © Oracle Corporation, 2001. All rights reserved.

The COUNT Function

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 satisfies the condition in the WHERE clause.

In contrast, COUNT (*expr*) returns the number of non-null values in the column identified by *expr*.

COUNT (DISTINCT *expr*) returns the number of unique, non-null values in the column identified by *expr*.

The slide example displays the number of employees in department 50.

Instructor Note

Demo: [5_count1.sql](#), [5_count2.sql](#)

Purpose: To illustrate using the COUNT (*) and COUNT (*expr*) functions

Using the COUNT Function

- **COUNT(*expr*)** returns the number of rows with non-null values for the *expr*.
- Display the number of department values in the **EMPLOYEES** table, excluding the null values.

```
SELECT COUNT(commission_pct)
FROM   employees
WHERE  department_id = 80;
```

COUNT(COMMISSION_PCT)
3

ORACLE

5-9

Copyright © Oracle Corporation, 2001. All rights reserved.

The COUNT Function (continued)

The slide example displays the number of employees in department 80 who can earn a commission.

Example

Display the number of department values in the **EMPLOYEES** table.

```
SELECT COUNT(department_id)
FROM   employees;
```

COUNT(DEPARTMENT_ID)
19

Using the DISTINCT Keyword

- **COUNT(DISTINCT expr)** returns the number of distinct non-null values of the *expr*.
- Display the number of distinct department values in the **EMPLOYEES** table.

```
SELECT COUNT(DISTINCT department_id)
FROM   employees;
```

COUNT(DISTINCTDEPARTMENT_ID)
7

ORACLE

The DISTINCT Keyword

Use the **DISTINCT keyword** to suppress the counting of any duplicate values within a column.

The example on the slide displays the number of distinct department values in the **EMPLOYEES** table.

Group Functions and Null Values

Group functions ignore null values in the column.

```
SELECT AVG(commission_pct)
FROM   employees;
```

AVG(COMMISSION_PCT)
.2125

ORACLE

5-11

Copyright © Oracle Corporation, 2001. All rights reserved.

Group Functions and Null Values

All **group functions ignore null values** in the column. In the slide example, the average is calculated based *only* on the rows in the table where a valid value is stored in the COMMISSION_PCT column. The average is calculated as the total commission paid to all employees divided by the number of employees receiving a commission (four).

Using the NVL Function with Group Functions

The NVL function forces group functions to include null values.

```
SELECT AVG(NVL(commission_pct, 0))  
FROM employees;
```

AVG(NVL(COMMISSION_PCT,0))	
	.0425

ORACLE

5-12

Copyright © Oracle Corporation, 2001. All rights reserved.

Group Functions and Null Values (continued)

The **NVL function** forces group functions to include null values. In the slide example, 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 paid to all employees divided by the total number of employees in the company (20).

Creating Groups of Data

EMPLOYEES

DEPARTMENT_ID	SALARY
10	4400
20	13000
20	6000
50	5800
50	3500
50	3100
50	2500
50	2600
60	9000
60	6000
60	4200
80	10500
80	8600
80	11000
90	24000
90	17000

...

20 rows selected.

4400
9500
3500
6400
10033

The average salary in EMPLOYEES table for each department.

DEPARTMENT_ID	AVG(SALARY)
10	4400
20	9500
50	3500
60	6400
80	10033.3333
90	19333.3333
110	10150
	7000

ORACLE

Groups of Data

Until now, all group functions have treated the table as one large group of information. At times, 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: The GROUP BY Clause Syntax

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

Divide rows in a table into smaller groups by using the GROUP BY clause.

ORACLE

5-14

Copyright © Oracle Corporation, 2001. All rights reserved.

The GROUP BY Clause

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:

<i>group_by_expression</i>	specifies columns whose values determine the basis for grouping rows
----------------------------	--

Guidelines

- If you include a group function in a SELECT clause, you cannot select individual results 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 must include the *columns* in the GROUP BY clause.
- You cannot use a column alias in the GROUP BY clause.
- By default, rows are sorted by ascending order of the columns included in the GROUP BY list. You can override this by using the ORDER BY clause.

Using the GROUP BY Clause

All 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)
10	4400
20	9500
50	3500
60	6400
80	10033.3333
90	19333.3333
110	10150
	7000

8 rows selected.

ORACLE

5-15

Copyright © Oracle Corporation, 2001. All rights reserved.

The GROUP BY Clause (continued)

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 on 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:
 - Department number column in the EMPLOYEES table
 - The average of all the salaries in the group 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. Since 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 being grouped by department number, so the AVG function that is being applied to the salary column will calculate the *average salary for each department*.

Instructor Note

Group results are sorted implicitly, on the grouping column. You can use ORDER BY to specify a different sort order, remembering to use only group functions, or the grouping column.

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)	
	4400
	9500
	3500
	6400
	10033.3333
	19333.3333
	10150
	7000

ORACLE

5-16

Copyright © Oracle Corporation, 2001. All rights reserved.

The GROUP BY Clause (continued)

The GROUP BY column does not have to be in the SELECT clause. For example, the SELECT statement on 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 use the group function in the ORDER BY clause.

```
SELECT  department_id, AVG(salary)
FROM    employees
GROUP BY department_id
```

DEPARTMENT_ID	AVG(SALARY)
50	3500
10	4400
60	6400
...	
90	19333.3333

8 rows selected.

Instructor Note

Demonstrate the query with and without the DEPARTMENT_ID column in the SELECT statement.

Grouping by More Than One Column

EMPLOYEES

DEPARTMENT_ID	JOB_ID	SALARY
90	AD_PRES	24000
90	AD_VP	17000
90	AD_VP	17000
60	IT_PROG	9000
60	IT_PROG	6000
60	IT_PROG	4200
50	ST_MAN	5800
50	ST_CLERK	3500
50	ST_CLERK	3100
50	ST_CLERK	2600
50	ST_CLERK	2500
80	SA_MAN	10500
80	SA_REP	11000
80	SA_REP	8600
...		
20	MK_REP	6000
110	AC_MGR	12000
110	AC_ACCOUNT	8300

20 rows selected.

“Add up the salaries in the EMPLOYEES table for each job, grouped by department.”

DEPARTMENT_ID	JOB_ID	SUM(SALARY)
10	AD_ASST	4400
20	MK_MAN	13000
20	MK_REP	6000
50	ST_CLERK	11700
50	ST_MAN	5800
60	IT_PROG	19200
80	SA_MAN	10500
80	SA_REP	19600
90	AD_PRES	24000
90	AD_VP	34000
110	AC_ACCOUNT	8300
110	AC_MGR	12000
	SA_REP	7000

13 rows selected.

ORACLE

Groups within Groups

Sometimes you need to see results for groups within groups. The slide shows a report that displays the total salary being paid to each job title, within each department.

The EMPLOYEES table is grouped first by department number and, within that grouping, by job title. For example, the four stock clerks in department 50 are grouped together and a single result (total salary) is produced for all stock clerks within the group.

Instructor Note

Demo: 5_order1.sql, 5_order2.sql

Purpose: To illustrate ordering columns that are grouped by DEPARTMENT_ID first and ordering columns that are grouped by JOB_ID first.

Using the GROUP BY Clause on Multiple Columns

```
SELECT    department_id dept_id, job_id, SUM(salary)
FROM      employees
GROUP BY  department_id, job_id ;
```

DEPT_ID	JOB_ID	SUM(SALARY)
10	AD_ASST	4400
20	MK_MAN	13000
20	MK_REP	6000
50	ST_CLERK	11700
50	ST_MAN	5800
60	IT_PROG	19200
80	SA_MAN	10500
80	SA_REP	19600
90	AD_PRES	24000
90	AD_VP	34000
110	AC_ACCOUNT	8300
110	AC_MGR	12000
	SA_REP	7000

13 rows selected.

ORACLE

Groups within Groups (continued)

You can return **summary results for groups** and subgroups by listing more than one GROUP BY column. You can determine the default sort order of the results by the order of the columns in the GROUP BY clause. Here is how the SELECT statement on the slide, containing a GROUP BY clause, is evaluated:

- The SELECT clause specifies the column to be retrieved:
 - Department number in the EMPLOYEES table
 - Job ID in the EMPLOYEES table
 - The sum of all the 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 GROUP BY clause specifies how you must group the rows:
 - First, the rows are grouped by department number.
 - Second, within the department number groups, the rows are grouped by job ID.

So the SUM function is being applied to the salary column for all job IDs within each department number group.

Illegal Queries 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;
```

```
SELECT department_id, COUNT(last_name)
      *
ERROR at line 1:
ORA-00937: not a single-group group function
```

Column missing in the `GROUP BY` clause

ORACLE

5-19

Copyright © Oracle Corporation, 2001. All rights reserved.

Illegal Queries Using Group Functions

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, then the error message “not a single-group group function” appears and an asterisk (*) points to the offending column. You can correct the error on the slide by adding the `GROUP BY` clause.

```
SELECT   department_id, count(last_name)
FROM     employees
GROUP BY department_id;
```

DEPARTMENT_ID	COUNT(LAST_NAME)
10	1
20	2
...	
	1

8 rows selected.

Any column or expression in the `SELECT` list that is not an aggregate function must be in the `GROUP BY` clause.

Instructor Note

Demo: [5_error.sql](#)

Purpose: To illustrate executing a `SELECT` statement with no `GROUP BY` clause

Introduction to Oracle9i: SQL 5-19

Illegal Queries Using Group Functions

- You cannot use the **WHERE** clause to restrict groups.
- You use the **HAVING** 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;
```

```
WHERE     AVG(salary) > 8000
          *
ERROR at line 3:
ORA-00934: group function is not allowed here
```

Cannot use the WHERE clause to restrict groups

ORACLE

5-20

Copyright © Oracle Corporation, 2001. All rights reserved.

Illegal Queries Using Group Functions (continued)

The **WHERE** clause cannot be used to restrict groups. The **SELECT** statement on the slide results in an error because it uses the **WHERE** clause to restrict the display of average salaries of those departments that have an average salary greater than \$8,000.

You can correct the slide error by using the **HAVING clause** to restrict groups.

```
SELECT    department_id, AVG(salary)
FROM      employees
HAVING    AVG(salary) > 8000
GROUP BY department_id;
```

DEPARTMENT_ID	AVG(SALARY)
20	9500
80	10033.3333
90	19333.3333
110	10150

Excluding Group Results

EMPLOYEES

DEPARTMENT_ID	SALARY
90	24000
90	17000
90	17000
60	9000
60	6000
60	4200
50	5800
50	3500
50	3100
50	2600
50	2500
80	10500
80	11000
80	8600
...	...
20	6000
110	12000
110	8300

20 rows selected.

The maximum salary per department when it is greater than \$10,000

DEPARTMENT_ID	MAX(SALARY)
20	13000
80	11000
90	24000
110	12000

ORACLE

Restricting Group Results

In the same way that you use the WHERE clause to restrict the rows that you select, you use the **HAVING clause** to restrict groups. To find the maximum salary of each department, but show only the departments that have a maximum salary of more 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.

Excluding Group Results: The HAVING Clause

Use the **HAVING** clause to restrict groups:

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

5-22

Copyright © Oracle Corporation, 2001. All rights reserved.

The HAVING Clause

You use the **HAVING** clause to specify which groups are to be displayed, and thus, you further restrict 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 that is more logical. Groups are formed and group functions are calculated before the **HAVING** clause is applied to the groups in the **SELECT** list.

Instructor Note

The Oracle server evaluates the clauses in the following order:

- If the statement contains a **WHERE** clause, the server establishes the candidate rows.
- The server identifies the groups specified in the **GROUP BY** clause.
- The **HAVING** clause further restricts result groups that do not meet the group criteria in the **HAVING** clause.

Using the HAVING Clause

```
SELECT    department_id, MAX(salary)
FROM      employees
GROUP BY  department_id
HAVING    MAX(salary)>10000 ;
```

DEPARTMENT_ID	MAX(SALARY)
20	13000
80	11000
90	24000
110	12000

ORACLE

5-23

Copyright © Oracle Corporation, 2001. All rights reserved.

The HAVING Clause (continued)

The slide example displays department numbers and maximum salaries for those departments whose maximum salary is 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 whose maximum salary is greater than \$10,000:

```
SELECT    department_id, AVG(salary)
FROM      employees
GROUP BY  department_id
HAVING    max(salary)>10000 ;
```

DEPARTMENT_ID	AVG(SALARY)
20	9500
80	10033.3333
90	19333.3333
110	10150

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
IT_PROG	19200
AD_PRES	24000
AD_VP	34000

ORACLE

5-24

Copyright © Oracle Corporation, 2001. All rights reserved.

The HAVING Clause (continued)

The slide example displays the job ID and total monthly salary for each job with a total payroll exceeding \$13,000. The example excludes sales representatives and sorts the list by the total monthly salary.

Instructor Note

Demo: 5_job1.sql, 5_job2.sql

Purpose: To illustrate using a WHERE clause to restrict rows by JOB_ID and using a HAVING clause to restrict groups by SUM(SALARY).

Nesting Group Functions

Display the maximum average salary.

```
SELECT  MAX(AVG(salary))  
FROM    employees  
GROUP BY department_id;
```

MAX(AVG(SALARY))
19333.3333

ORACLE

Nesting Group Functions

Group functions can be nested to a depth of two. The slide example displays the maximum average salary.

Summary

In this lesson, you should have learned how to:

- Use the group functions COUNT, MAX, MIN, AVG
- Write queries that use the GROUP BY clause
- Write queries that use the HAVING clause

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

ORACLE

5-26

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary

Seven group functions are available in SQL:

- AVG
- COUNT
- MAX
- MIN
- SUM
- STDDEV
- VARIANCE

You can create subgroups by using the GROUP BY clause. Groups can be excluded using the HAVING clause.

Place the HAVING and GROUP BY clauses after the WHERE clause in a statement. Place the ORDER BY clause last.

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 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.

Practice 5 Overview

This practice covers the following topics:

- **Writing queries that use the group functions**
- **Grouping by rows to achieve more than one result**
- **Excluding groups by using the `HAVING` clause**

ORACLE

5-27

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 5 Overview

At the end of this practice, you should be familiar with using group functions and selecting groups of data.

Paper-Based Questions

For questions 1-3, circle either True or False.

Note: Column aliases are used for the queries.

Instructor Note

Hint for Question #7: Advise the students to think about the `MANAGER_ID` column in `EMPLOYEES` when determining the number of managers, rather than the `JOB_ID` column.

Practice 5

Determine the validity of the following three statements. Circle either True or False.

1. Group functions work across many rows to produce one result per group.
True/False
2. Group functions include nulls in calculations.
True/False
3. The WHERE clause restricts rows prior to inclusion in a group calculation.
True/False
4. Display the highest, lowest, sum, and average salary of all employees. Label the columns Maximum, Minimum, Sum, and Average, respectively. Round your results to the nearest whole number. Place your SQL statement in a text file named lab5_4.sql.

Maximum	Minimum	Sum	Average
24000	2500	175500	8775

5. Modify the query in lab5_4.sql to display the minimum, maximum, sum, and average salary for each job type. Resave lab5_4.sql to lab5_5.sql. Run the statement in lab5_5.sql.

JOB_ID	Maximum	Minimum	Sum	Average
AC_ACCOUNT	8300	8300	8300	8300
AC_MGR	12000	12000	12000	12000
AD_ASST	4400	4400	4400	4400
AD PRES	24000	24000	24000	24000
AD_VP	17000	17000	34000	17000
IT_PROG	9000	4200	19200	6400
MK_MAN	13000	13000	13000	13000
MK_REP	6000	6000	6000	6000
SA_MAN	10500	10500	10500	10500
SA_REP	11000	7000	26600	8867
ST_CLERK	3500	2500	11700	2925
ST_MAN	5800	5800	5800	5800

12 rows selected.

Practice 5 (continued)

6. Write a query to display the number of people with the same job.

JOB_ID	COUNT(*)
AC_ACCOUNT	1
AC_MGR	1
AD_ASST	1
AD PRES	1
AD_VP	2
IT_PROG	3
MK_MAN	1
MK_REP	1
SA_MAN	1
SA_REP	3
ST_CLERK	4
ST_MAN	1

12 rows selected.

7. Determine the number of managers without listing them. Label the column Number of Managers. *Hint: Use the MANAGER_ID column to determine the number of managers.*

Number of Managers
8

8. Write a query that displays the difference between the highest and lowest salaries. Label the column DIFFERENCE.

DIFFERENCE
21500

If you have time, complete the following exercises:

9. Display the manager number and the salary of the lowest paid employee for that manager. Exclude anyone whose manager is not known. Exclude any groups where the minimum salary is \$6,000 or less. Sort the output in descending order of salary.

MANAGER_ID	MIN(SALARY)
102	9000
205	8300
149	7000

Practice 5 (continued)

10. Write a query to display each department's name, location, number of employees, and the average salary for all employees in that department. Label the columns Name, Location, Number of People, and Salary, respectively. Round the average salary to two decimal places.

Name	Location	Number of People	Salary
Accounting	1700	2	10150
Administration	1700	1	4400
Executive	1700	3	19333.33
IT	1400	3	6400
Marketing	1800	2	9500
Sales	2500	3	10033.33
Shipping	1500	5	3500

7 rows selected.

If you want an extra challenge, complete the following exercises:

11. Create a query that will display the total number of employees and, of that total, the number of employees hired in 1995, 1996, 1997, and 1998. Create appropriate column headings.

TOTAL	1995	1996	1997	1998
20	1	2	2	3

12. Create a matrix query to display the job, the salary for that job based on department number, and the total salary for that job, for departments 20, 50, 80, and 90, giving each column an appropriate heading.

Job	Dept 20	Dept 50	Dept 80	Dept 90	Total
AC_ACCOUNT					8300
AC_MGR					12000
AD_ASST					4400
AD_PRES				24000	24000
AD_VP				34000	34000
IT_PROG					19200
MK_MAN	13000				13000
MK_REP	6000				6000
SA_MAN			10500		10500
SA_REP			19600		26600
ST_CLERK		11700			11700
ST_MAN		5800			5800

12 rows selected.

6

Subqueries

ORACLE

Copyright © Oracle Corporation, 2001. All rights reserved.

Schedule:

Timing

25 minutes

30 minutes

55 minutes

Topic

Lecture

Practice

Total

Objectives

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

- Describe the types of problem that subqueries can solve
- Define subqueries
- List the types of subqueries
- Write single-row and multiple-row subqueries

ORACLE

6-2

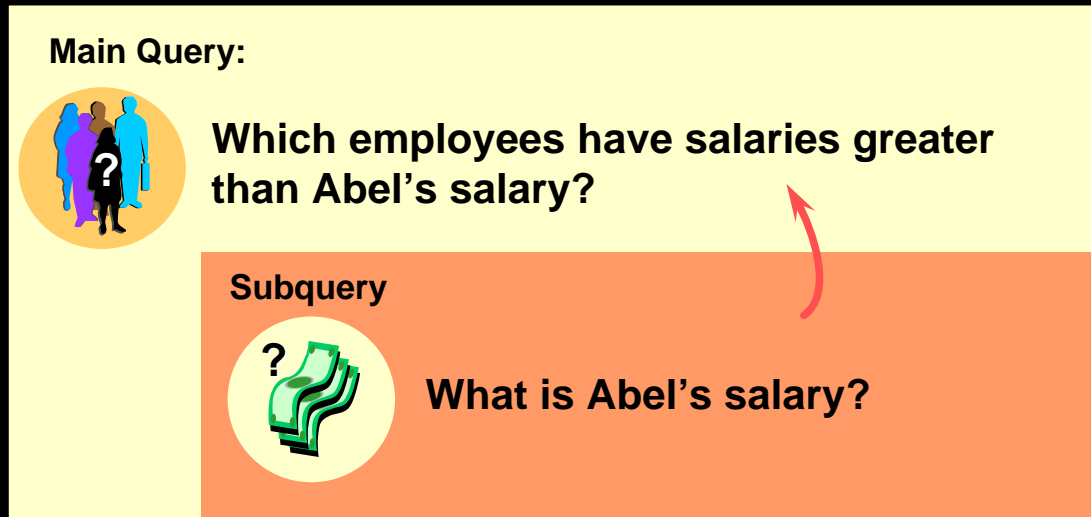
Copyright © Oracle Corporation, 2001. All rights reserved.

Lesson Aim

In this lesson, you learn about 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 covers **single-row subqueries** and **multiple-row subqueries**.

Using a Subquery to Solve a Problem

Who has a salary greater than Abel's?



ORACLE

6-3

Copyright © Oracle Corporation, 2001. All rights reserved.

Using a Subquery to Solve a Problem

Suppose you want to write a query to find out who earns a salary greater than Abel's salary.

To solve this problem, you need *two* queries: one to find what Abel earns, and a second query to find who earns more than that amount.

You can solve this problem by combining the two queries, placing one query *inside* the other query.

The **inner query** or the **subquery** returns a value that is used by the outer query or the main query.

Using a subquery is equivalent to performing two sequential queries and using the result of the first query as the search value in the second query.

Instructor Note

This lesson concentrates on noncorrelated subqueries.

Subquery Syntax

```
SELECT  select_list
FROM    table
WHERE   expr operator
        (SELECT      select_list
         FROM         table);
```

- The subquery (inner query) executes once before the main query.
- The result of the subquery is used by the main query (outer query).

ORACLE

6-4

Copyright © Oracle Corporation, 2001. All rights reserved.

Subqueries

A subquery is a `SELECT` statement that is embedded in a 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 `WHERE` clause
- The `HAVING` clause
- The `FROM` clause

In the syntax:

operator includes a comparison condition such as `>`, `=`, or `IN`

Note: Comparison conditions fall into two classes: single-row operators (`>`, `=`, `>=`, `<`, `<>`, `<=`) and multiple-row operators (`IN`, `ANY`, `ALL`).

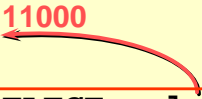
The subquery is often referred to as a **nested `SELECT`**, **sub-`SELECT`**, or **inner `SELECT`** statement. The subquery generally executes first, and its output is used to complete the query condition for the main or outer query.

Instructor Note

Additionally, subqueries can be placed in the `CREATE VIEW` statement, `CREATE TABLE` statement, `UPDATE` statement, `INTO` clause of an `INSERT` statement, and `SET` clause of an `UPDATE` statement.

Using a Subquery

```
SELECT last_name
FROM   employees 11000
WHERE  salary >
      (SELECT salary
       FROM   employees
       WHERE  last_name = 'Abel');
```



LAST_NAME
King
Kochhar
De Haan
Hartstein
Higgins

ORACLE

6-5

Copyright © Oracle Corporation, 2001. All rights reserved.

Using a Subquery

In the slide, the inner query determines the salary of employee Abel. The **outer query** takes the result of the **inner query** and uses this result to display all the employees who earn more than this amount.

Instructor Note

Execute the subquery (inner query) on its own first to show the value that the subquery returns. Then execute the outer query using the result returned by the inner query. Finally, execute the entire query (containing the subquery), and show that the result is the same.

Introduction to Oracle9i: SQL 6-5

Guidelines for Using Subqueries

- Enclose subqueries in parentheses.
- Place subqueries on the right side of the comparison condition.
- The `ORDER BY` clause in the subquery is not needed unless you are performing Top-N analysis.
- Use single-row operators with single-row subqueries and use multiple-row operators with multiple-row subqueries.

ORACLE

6-6

Copyright © Oracle Corporation, 2001. All rights reserved.

Guidelines for Using Subqueries

- A subquery must be enclosed in parentheses.
- Place the subquery on the right side of the comparison condition for readability.
- Prior to release Oracle8i, subqueries could not contain an `ORDER BY` clause. Only one `ORDER BY` clause can be used for a `SELECT` statement, and if specified it must be the last clause in the main `SELECT` statement. Starting with release Oracle8i, an `ORDER BY` clause can be used and is required in the subquery to perform Top-N analysis.
- Two classes of comparison conditions are used in subqueries: single-row operators and multiple-row operators.

Instructor Note

A subquery can execute multiple times in correlated subqueries. Students may ask how many subqueries can be written. The Oracle server imposes no limit on the number of subqueries; the limit is related to the buffer size that the query uses.

Types of Subqueries

- **Single-row subquery**



- **Multiple-row subquery**



ORACLE

6-7

Copyright © Oracle Corporation, 2001. All rights reserved.

Types of Subqueries

- **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:** Queries that return more than one column from the inner SELECT statement.

Instructor Note

Multiple column subqueries are also available.

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

6-8

Copyright © Oracle Corporation, 2001. All rights reserved.

Single-Row Subqueries

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.



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);
```

LAST_NAME	JOB_ID
Rajs	ST_CLERK
Davies	ST_CLERK
Matos	ST_CLERK
Vargas	ST_CLERK

Executing Single-Row Subqueries

```
SELECT last_name, job_id, salary
FROM   employees
WHERE  job_id = 
          (SELECT job_id
           FROM   employees
           WHERE  employee_id = 141)
AND    salary > 
          (SELECT salary
           FROM   employees
           WHERE  employee_id = 143);
```

LAST_NAME	JOB_ID	SALARY
Rajs	ST_CLERK	3500
Davies	ST_CLERK	3100

ORACLE

6-9

Copyright © Oracle Corporation, 2001. All rights reserved.

Executing Single-Row Subqueries

A `SELECT` statement can be considered as a query block. The example on the slide displays employees whose job ID is the same as that of employee 141 and whose salary is greater than that of employee 143.

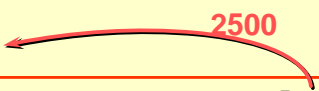
The example consists of three query blocks: the outer query and two inner queries. The inner query blocks are executed first, producing the query results `ST_CLERK` and `2600`, respectively. The outer query block is then processed and uses the values returned by the inner queries to complete its search conditions.

Both inner queries return single values (`ST_CLERK` and `2600`, 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);
```



LAST_NAME	JOB_ID	SALARY
Vargas	ST_CLERK	2500

ORACLE

6-10

Copyright © Oracle Corporation, 2001. All rights reserved.

Using Group Functions in a Subquery

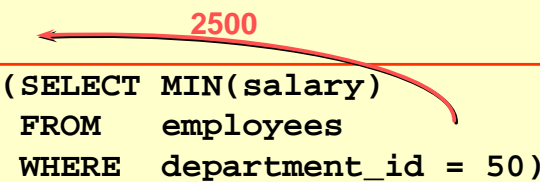
You can display data from a main query by using a **group function in a subquery** to return a single row. The subquery is in parentheses and is placed after the comparison condition.

The example on 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.

The HAVING Clause with Subqueries

- The Oracle server executes 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) > (SELECT MIN(salary)
                        FROM      employees
                        WHERE     department_id = 50);
```



ORACLE

6-11

Copyright © Oracle Corporation, 2001. All rights reserved.

The HAVING Clause with Subqueries

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 on the slide displays all the departments that have a minimum salary greater than that of department 50.

DEPARTMENT_ID	MIN(SALARY)
10	4400
20	6000
...	
	7000

7 rows selected.

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);
```

```
ERROR at line 4:
ORA-01427: single-row subquery returns more than
one row
```

Single-row operator with multiple-row subquery

ORACLE

6-12

Copyright © Oracle Corporation, 2001. All rights reserved.

Errors with Subqueries

One common error with subqueries is more than one row returned for a single-row subquery.

In the SQL statement on the slide, the subquery contains a `GROUP BY` clause, which implies that the subquery will return multiple rows, one for each group it finds. In this case, the result of the subquery will be 4400, 6000, 2500, 4200, 7000, 17000, and 8300.

The outer query takes the results of the subquery (4400, 6000, 2500, 4200, 7000, 17000, 8300) and uses these results in its `WHERE` clause. The `WHERE` clause contains an equal (`=`) operator, a single-row comparison operator expecting 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`.

Will this Statement Return Rows?

```
SELECT last_name, job_id
FROM   employees
WHERE  job_id =
      (SELECT job_id
       FROM   employees
       WHERE  last_name = 'Haas');
```

no rows selected

Subquery returns no values

ORACLE

6-13

Copyright © Oracle Corporation, 2001. All rights reserved.

Problems with Subqueries

A common problem with subqueries is **no rows being returned by the inner query**.

In the SQL statement on 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 selects no rows when executed.

There is no employee named Haas. So 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.

Multiple-Row Subqueries

- **Return more than one row**
- **Use multiple-row comparison operators**

Operator	Meaning
IN	Equal to any member in the list
ANY	Compare value to each value returned by the subquery
ALL	Compare value to every value returned by the subquery

ORACLE

6-14

Copyright © Oracle Corporation, 2001. All rights reserved.

Multiple-Row Subqueries

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.

```
SELECT last_name, salary, department_id
FROM   employees
WHERE  salary IN (SELECT   MIN(salary)
                   FROM     employees
                   GROUP BY department_id);
```

Example

Find 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 returned by the inner query to complete its search condition. In fact, the main query would appear to the Oracle server as follows:

```
SELECT last_name, salary, department_id
FROM   employees
WHERE  salary IN (2500, 4200, 4400, 6000, 7000, 8300, 8600, 17000);
```

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
124	Mourgos	ST_MAN	5800
141	Rajs	ST_CLERK	3500
142	Davies	ST_CLERK	3100
143	Matos	ST_CLERK	2600
144	Vargas	ST_CLERK	2500

10 rows selected.

ORACLE

6-15

Copyright © Oracle Corporation, 2001. All rights reserved.

Multiple-Row Subqueries (continued)

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.

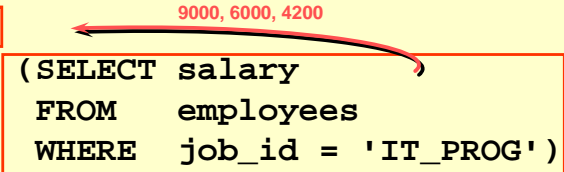
<ALL means less than the maximum. >ALL means more than the minimum.

Instructor Note

When using SOME or ANY, you often use the DISTINCT keyword to prevent rows from being selected several times.

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';
```



EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
141	Rajs	ST_CLERK	3500
142	Davies	ST_CLERK	3100
143	Matos	ST_CLERK	2600
144	Vargas	ST_CLERK	2500

ORACLE

6-16

Copyright © Oracle Corporation, 2001. All rights reserved.

Multiple-Row Subqueries (continued)

The **ALL operator** compares a value to *every* value returned by a subquery. The slide example 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.

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);

no rows selected
```

ORACLE

6-17

Copyright © Oracle Corporation, 2001. All rights reserved.

Returning Nulls in the Resulting Set of a Subquery

The SQL statement on 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 hence 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);
```

Introduction to Oracle9i: SQL 6-17

Summary

In this lesson, you should have learned how to:

- Identify when a subquery can help solve a question
- Write subqueries when a query is based on unknown values

```
SELECT  select_list
FROM    table
WHERE   expr operator
        (SELECT select_list
         FROM   table);
```

ORACLE

6-18

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary

In this lesson, you should have learned how to use subqueries. A subquery is a `SELECT` statement that is embedded in a clause of another SQL statement. Subqueries are useful when a query is based on a search criteria 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, and the `WHERE` or `HAVING` clause uses the results
- Can contain group functions

Practice 6 Overview

This practice covers the following topics:

- **Creating subqueries to query values based on unknown criteria**
- **Using subqueries to find out which values exist in one set of data and not in another**

ORACLE

6-19

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 6

In this practice, you write complex queries using nested `SELECT` statements.

Paper-Based Questions

You may want to create the inner query first for these questions. Make sure that it runs and produces the data that you anticipate before coding the outer query.

Practice 6

1. Write a query to display the last name and hire date of any employee in the same department as Zlotkey. Exclude Zlotkey.

LAST_NAME	HIRE_DATE
Abel	11-MAY-96
Taylor	24-MAR-98

2. Create a query to display the employee numbers and last names of all employees who earn more than the average salary. Sort the results in ascending order of salary.

EMPLOYEE_ID	LAST_NAME	SALARY
103	Hunold	9000
149	Zlotkey	10500
174	Abel	11000
205	Higgins	12000
201	Hartstein	13000
101	Kochhar	17000
102	De Haan	17000
100	King	24000

8 rows selected.

3. Write a query that displays the employee numbers and last names of all employees who work in a department with any employee whose last name contains a *u*. Place your SQL statement in a text file named lab6_3.sql. Run your query.

EMPLOYEE_ID	LAST_NAME
124	Mourgos
141	Rajs
142	Davies
143	Matos
144	Vargas
103	Hunold
104	Ernst
107	Lorentz

8 rows selected.

Practice 6 (continued)

4. Display the last name, department number, and job ID of all employees whose department location ID is 1700.

LAST_NAME	DEPARTMENT_ID	JOB_ID
Whalen	10	AD_ASST
King	90	AD PRES
Kochhar	90	AD_VP
De Haan	90	AD_VP
Higgins	110	AC_MGR
Gietz	110	AC_ACCOUNT

6 rows selected.

5. Display the last name and salary of every employee who reports to King.

LAST_NAME	SALARY
Kochhar	17000
De Haan	17000
Mourgos	5800
Zlotkey	10500
Hartstein	13000

6. Display the department number, last name, and job ID for every employee in the Executive department.

DEPARTMENT_ID	LAST_NAME	JOB_ID
90	King	AD PRES
90	Kochhar	AD_VP
90	De Haan	AD_VP

If you have time, complete the following exercises:

7. Modify the query in lab6_3.sql to display the employee numbers, last names, and salaries of all employees who earn more than the average salary and who work in a department with any employee with a *u* in their name. Resave lab6_3.sql to lab6_7.sql. Run the statement in lab6_7.sql.

EMPLOYEE_ID	LAST_NAME	SALARY
103	Hunold	9000

7

Producing Readable Output with *i*SQL*Plus

ORACLE

Copyright © Oracle Corporation, 2001. All rights reserved.

Schedule:	Timing	Topic
	35 minutes	Lecture
	35 minutes	Practice
	70 minutes	Total

Objectives

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

- **Produce queries that require a substitution variable**
- **Customize the *iSQL*Plus* environment**
- **Produce more readable output**
- **Create and execute script files**

ORACLE

7-2

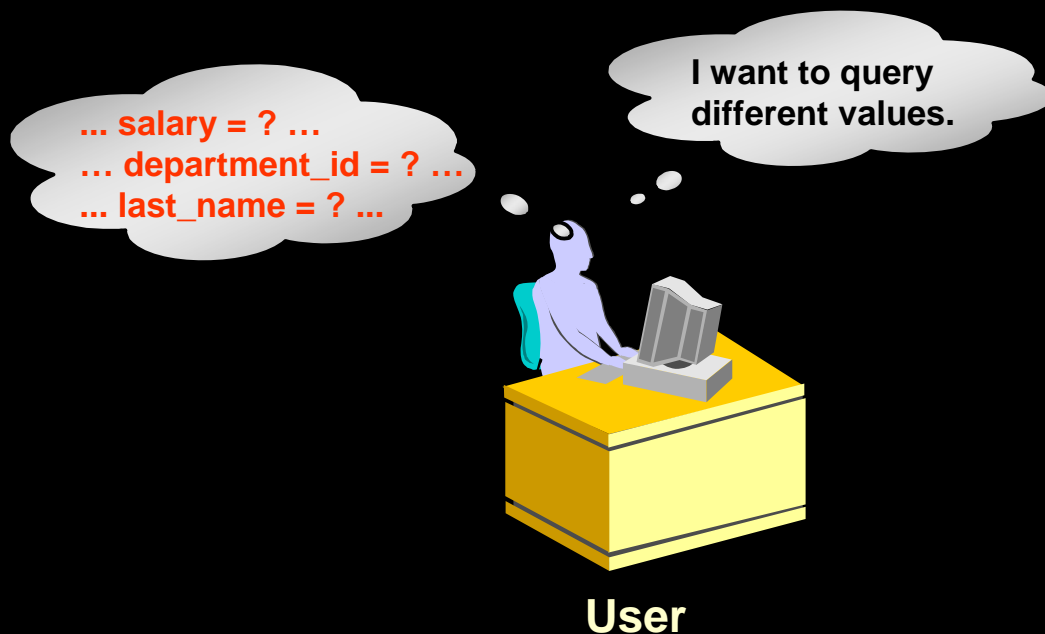
Copyright © Oracle Corporation, 2001. All rights reserved.

Lesson Aim

In this lesson, you will learn how to include *iSQL*Plus* commands to produce more readable SQL output.

You can create a command file containing a WHERE clause to restrict the rows displayed. To change the condition each time the command file is run, you use substitution variables. *Substitution variables* can replace values in the WHERE clause, a text string, and even a column or a table name.

Substitution Variables



ORACLE

7-3

Copyright © Oracle Corporation, 2001. All rights reserved.

Substitution Variables

The examples so far have been hard-coded. In a finished application, the user would trigger the report, and the report would run without further prompting. The range of data would be predetermined by the fixed WHERE clause in the *iSQL*Plus script file*.

Using *iSQL*Plus*, you can create reports that prompt the user 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 the values are temporarily stored. When the statement is run, the value is substituted.

Instructor Note

Demo: `7_varno.sql`, `7_varyes.sql`

Purpose: To illustrate returning all rows and using a case-insensitive query with substitution variables. With *iSQL*Plus* 9.0.1.0.1, there is a bug when using `&` substitution and wildcards (%) for character values.

This bug has been reported. The `7_varyes.sql` will produce an error in *iSQL*Plus*, but the concept is important for students continuing classes using other products (such as Forms, Reports). You may want to demonstrate `7_varyes.sql` in the *SQL*Plus* environment as an option.

Substitution Variables

Use *iSQL*Plus* substitution variables to:

- **Temporarily store values**
 - Single ampersand (&)
 - Double ampersand (&&)
 - DEFINE command
- **Pass variable values between SQL statements**
- **Dynamically alter headers and footers**

ORACLE

7-4

Copyright © Oracle Corporation, 2001. All rights reserved.

Substitution Variables

In *iSQL*Plus*, you can use **single ampersand (&) substitution** variables to temporarily store values. You can predefine variables in *iSQL*Plus* by using the `DEFINE` command. `DEFINE` creates and assigns a value to a variable.

Examples of Restricted Ranges of Data

- Reporting figures only for the current quarter or 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 be used to achieve other goals. For example:

- Dynamically altering headers and footers
- Obtaining input values from a file rather than from a person
- Passing values from one SQL statement to another

*iSQL*Plus* does not support validation checks (except for data type) on user input.

Instructor Note

A substitution variable can be used anywhere in SQL and *iSQL*Plus* commands, except as the first word entered at the command prompt.

Using the & 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 ;
```

The screenshot shows the iSQL*Plus web interface. At the top, there are links for Password, Log Out, and Help. Below the header, the text 'Define Substitution Variables' is displayed. A text input field contains the variable name 'employee_num'. To the right of the input field are two buttons: 'Submit for Execution' and 'Cancel'.

ORACLE

7-5

Copyright © Oracle Corporation, 2001. All rights reserved.

Single-Ampersand Substitution Variable

When running a report, users often want to restrict the data returned dynamically. *iSQL*Plus* provides this flexibility by means of user variables. Use an ampersand (&) to identify each variable in your SQL statement. You do not need to define the value of each variable.

Notation	Description
<i>&user_variable</i>	Indicates a variable in a SQL statement; if the variable does not exist, <i>iSQL*Plus</i> prompts the user for a value (<i>iSQL*Plus</i> discards a new variable once it is used.)

The example on the slide creates an *iSQL*Plus* substitution variable for an employee number. When the statement is executed, *iSQL*Plus* prompts the user for an employee number and then displays the employee number, last name, salary, and department number for that employee.

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

Using the & Substitution Variable

ORACLE[®] iSQL*Plus

Define Substitution Variables

"employee_num"

Submit for Execution Cancel

old 3: WHERE employee_id = &employee_num
new 3: WHERE employee_id = 101

EMPLOYEE_ID	LAST_NAME	SALARY	DEPARTMENT_ID
101	Kochhar	17000	90

ORACLE[®]

7-6

Copyright © Oracle Corporation, 2001. All rights reserved.

Single-Ampersand Substitution Variable

When *iSQL*Plus* detects that the SQL statement contains an `&`, you are prompted to enter a value for the substitution variable named in the SQL statement. Once you enter a value and click the **Submit for Execution** button, the results are displayed in the output area of your *iSQL*Plus* session.

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' ;
```

Define Substitution Variables

"job_title" IT_PROG

Submit for Execution

Cancel

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

ORACLE

7-7

Copyright © Oracle Corporation, 2001. All rights reserved.

Specifying Character and Date Values with Substitution Variables

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 variable in 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 iSQL*Plus substitution variable.

Note: You can also use functions such as UPPER and LOWER with the ampersand. Use UPPER('&job_title') so that the user does not have to enter the job title in uppercase.

Specifying Column Names, Expressions, and Text

Use substitution variables to supplement the following:

- WHERE conditions
- ORDER BY clauses
- Column expressions
- Table names
- Entire SELECT statements

ORACLE

7-8

Copyright © Oracle Corporation, 2001. All rights reserved.

Specifying Column Names, Expressions, and Text

Not only can you use the substitution variables in the WHERE clause of a SQL statement, but these variables can also be used to substitute for column names, expressions, or text.

Example

Display the employee number and any other column and any condition of employees.

```
SELECT  employee_id, &column_name
FROM    employees
WHERE   &condition;
```

"column_name"

"condition"

EMPLOYEE_ID	JOB_ID
200	AD_ASST

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

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

Specifying Column Names, Expressions, and Text

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

Define Substitution Variables

"column_name"
"condition"
"order_column"

EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
102	De Haan	AD_VP	17000
100	King	AD_PRE	24000
101	Kochhar	AD_VP	17000

ORACLE

7-9

Copyright © Oracle Corporation, 2001. All rights reserved.

Specifying Column Names, Expressions, and Text (continued)

The slide example displays the employee number, name, job title, and any other column specified by the user at run time, from the EMPLOYEES table. You can also specify the condition for retrieval of rows and the column name by which the resultant data has to be ordered.

Instructor Note

Demo: 7_expr.sql

Purpose: To illustrate changing column names and conditions by using substitution variables

Defining Substitution Variables

- You can predefine variables using the *iSQL*Plus* **DEFINE** command.
`DEFINE variable = value` creates a user variable with the CHAR data type.
- If you need to predefine a variable that includes spaces, you must enclose the value within single quotation marks when using the **DEFINE** command.
- A defined variable is available for the session

ORACLE

7-10

Copyright © Oracle Corporation, 2001. All rights reserved.

Defining Substitution Variables

You can predefine user variables before executing a **SELECT** statement. *iSQL*Plus* provides the **DEFINE** command for defining and setting substitution variables:

Command	Description
<code>DEFINE variable = value</code>	Creates a user variable with the CHAR data and assigns a value to it
<code>DEFINE variable</code>	Displays the variable, its value, and its data type
<code>DEFINE</code>	Displays all user variables with their values and data types

Instructor Note

Mention that *iSQL*Plus* commands can continue onto multiple lines and that they require the continuation character, the hyphen.

DEFINE and UNDEFINE Commands

- A variable remains defined until you either:
 - Use the UNDEFINE command to clear it
 - Exit *iSQL*Plus*
- You can verify your changes with the DEFINE command.

```
DEFINE job_title = IT_PROG
DEFINE job_title
DEFINE JOB_TITLE          = "IT_PROG" (CHAR)
```

```
UNDEFINE job_title
DEFINE job_title
SP2-0135: symbol job_title is UNDEFINED
```

ORACLE

7-11

Copyright © Oracle Corporation, 2001. All rights reserved.

The DEFINE and UNDEFINE Commands

Variables are defined until you either:

- Issue the UNDEFINE command on a variable
- Exit *iSQL*Plus*

When you undefine variables, you can verify your changes with the DEFINE command. When you exit *iSQL*Plus*, variables defined during that session are lost.

Using the DEFINE Command with & Substitution Variable

- Create the substitution variable using the **DEFINE** command.

```
DEFINE employee_num = 200
```

- Use a variable prefixed with an ampersand (&) to substitute the value in the SQL statement.

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

EMPLOYEE_ID	LAST_NAME	SALARY	DEPARTMENT_ID
200	Whalen	4400	10

ORACLE

7-12

Copyright © Oracle Corporation, 2001. All rights reserved.

Using the DEFINE Command

The example on the slide creates an *iSQL*Plus* substitution variable for an employee number by using the **DEFINE** command, and at run time displays the employee number, name, salary, and department number for that employee.

Because the variable is created using the *iSQL*Plus* **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 *iSQL*Plus* session.

Using the && Substitution Variable

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

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

Define Substitution Variables

"column_name"

Submit for Execution

Cancel

EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
200	Whalen	AD_ASST	10
201	Hartstein	MK_MAN	20

...

20 rows selected.

ORACLE

7-13

Copyright © Oracle Corporation, 2001. All rights reserved.

Double-Ampersand Substitution Variable

You can use the **double-ampersand (&&)** substitution variable if you want to reuse the variable value without prompting the user each time. The user will see the prompt for the value only once. In the example on the slide, the user is asked to give the value for variable *column_name* only once. The value supplied by the user (*department_id*) is used both for display and ordering of data.

*iSQL*Plus* stores the value supplied by using the `DEFINE` command; it will use it again whenever you reference the variable name. Once a user variable is in place, you need to use the `UNDEFINE` command to delete it.

Using the VERIFY Command

Use the **VERIFY** command to toggle the display of the substitution variable, before and after *iSQL*Plus* replaces substitution variables with values.

```
SET VERIFY ON
```

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

"employee_num" 200

```
old 3: WHERE employee_id = &employee_num  
new 3: WHERE employee_id = 200
```

ORACLE

7-14

Copyright © Oracle Corporation, 2001. All rights reserved.

The VERIFY Command

To confirm the changes in the SQL statement, use the *iSQL*Plus* **VERIFY** command. Setting **SET VERIFY ON** forces *iSQL*Plus* to display the text of a command before and after it replaces substitution variables with values.

The example on the slide displays the old as well as the new value of the `EMPLOYEE_ID` column.

Customizing the *iSQL*Plus* Environment

- Use **SET** commands to control current session.

```
SET system_variable value
```

- Verify what you have set by using the **SHOW** command.

```
SET ECHO ON
```

```
SHOW ECHO  
echo ON
```

ORACLE

7-15

Copyright © Oracle Corporation, 2001. All rights reserved.

Customizing the *iSQL*Plus* Environment

You can control the environment in which *iSQL*Plus* is currently operating by using the **SET** commands.

Syntax

```
SET system_variable value
```

In the syntax:

<i>system_variable</i>	is a variable that controls one aspect of the session environment
<i>value</i>	is a value for the system variable

You can verify what you have set by using the **SHOW** command. The **SHOW** command on the slide checks whether **ECHO** had been set on or off.

To see all **SET** variable values, use the **SHOW ALL** command.

For more information, see *iSQL*Plus User's Guide and Reference*, "Command Reference."

SET Command Variables

- **ARRAYSIZE** {20 | *n*}
- **FEEDBACK** {6 | *n* | OFF | ON}
- **HEADING** {OFF | ON}
- **LONG** {80 | *n*} | ON | *text*}

SET HEADING OFF

SHOW HEADING
HEADING OFF

ORACLE

7-16

Copyright © Oracle Corporation, 2001. All rights reserved.

SET Command Variables

SET Variable and Values	Description
ARRAY[SIZE] { <u>20</u> <i>n</i> }	Sets the database data fetch size
FEED[BACK] { <u>6</u> <i>n</i> OFF ON}	Displays the number of records returned by a query when the query selects at least <i>n</i> records
HEA[DING] {OFF <u>ON</u> }	Determines whether column headings are displayed in reports
LONG { <u>80</u> <i>n</i> }	Sets the maximum width for displaying LONG values

Note: The value *n* represents a numeric value. The underlined values indicate default values. If you enter no value with the variable, *iSQL*Plus* assumes the default value.

iSQL*Plus Format Commands

- **COLUMN** [*column option*]
- **TITLE** [*text* | OFF | ON]
- **BTITLE** [*text* | OFF | ON]
- **BREAK** [ON *report_element*]

ORACLE

7-17

Copyright © Oracle Corporation, 2001. All rights reserved.

Obtaining More Readable Reports

You can control the report features by using the following commands:

Command	Description
COL[UMN] [<i>column option</i>]	Controls column formats
TTI[TLE] [<i>text</i> OFF ON]	Specifies a header to appear at the top of each page of the report
BTI[TLE] [<i>text</i> OFF ON]	Specifies a footer to appear at the bottom of each page of the report
BRE[AK] [ON <i>report_element</i>]	Suppresses duplicate values and divides rows of data into sections by using line breaks

Guidelines

- All format commands remain in effect until the end of the iSQL*Plus session or until the format setting is overwritten or cleared.
- Remember to reset your iSQL*Plus settings to the default values after every report.
- There is no command for setting an iSQL*Plus variable to its default value; you must know the specific value or log out and log in again.
- If you give an alias to your column, you must reference the alias name, not the column name.

The COLUMN Command

Controls display of a column:

```
COL[UMN] [{column|alias} [option]]
```

- **CLE[AR]:** Clears any column formats
- **HEA[DING] *text*:** Sets the column heading
- **FOR[MAT] *format*:** Changes the display of the column using a format model
- **NOPRINT | PRINT**
- **NULL**

ORACLE

7-18

Copyright © Oracle Corporation, 2001. All rights reserved.

COLUMN Command Options

Option	Description
CLE[AR]	Clears any column formats
HEA[DING] <i>text</i>	Sets the column heading (a vertical line () forces a line feed in the heading if you do not use justification.)
FOR[MAT] <i>format</i>	Changes the display of the column data
NOPRI[NT]	Hides the column
NUL[L] <i>text</i>	Specifies text to be displayed for null values
PRI[NT]	Shows the column

Using the COLUMN Command

- Create column headings.

```
COLUMN last_name HEADING 'Employee|Name'  
COLUMN salary JUSTIFY LEFT FORMAT $99,990.00  
COLUMN manager FORMAT 999999999 NULL 'No manager'
```

- Display the current setting for the LAST_NAME column.

```
COLUMN last_name
```

- Clear settings for the LAST_NAME column.

```
COLUMN last_name CLEAR
```

ORACLE

7-19

Copyright © Oracle Corporation, 2001. All rights reserved.

Displaying or Clearing Settings

To show or clear the current **COLUMN command** settings, use the following commands:

Command	Description
COL[UMN] <i>column</i>	Displays the current settings for the specified column
COL[UMN]	Displays the current settings for all columns
COL[UMN] <i>column</i> CLE[AR]	Clears the settings for the specified column
CLE[AR] COL[UMN]	Clears the settings for all columns

COLUMN Format Models

Element	Description	Example	Result
9	Single zero-suppression digit	999999	1234
0	Enforces leading zero	099999	001234
\$	Floating dollar sign	\$9999	\$1234
L	Local currency	L9999	L1234
.	Position of decimal point	9999.99	1234.00
,	Thousand separator	9,999	1,234

ORACLE

7-20

Copyright © Oracle Corporation, 2001. All rights reserved.

COLUMN Format Models

The slide displays sample **COLUMN format models**.

The Oracle server displays a string of pound signs (#) in place of a whole number whose digits exceed the number of digits provided in the format model. It also displays pound signs in place of a value whose format model is alphanumeric but whose actual value is numeric.

Using the BREAK Command

Use the **BREAK** command to suppress duplicates.

```
BREAK ON job_id
```

ORACLE

7-21

Copyright © Oracle Corporation, 2001. All rights reserved.

The BREAK Command

Use the **BREAK command** to divide rows into sections and suppress duplicate values. To ensure that the BREAK command works effectively, use the ORDER BY clause to order the columns that you are breaking on.

Syntax

```
BREAK on column[|alias|row]
```

In the syntax:

column[alias row]	suppresses the display of duplicate values for a given column
--------------------	---

Clear all BREAK settings by using the **CLEAR command**:

```
CLEAR BREAK
```

Using the TTITLE and BTITLE Commands

- Display headers and footers.

```
TTI[TLE] [text|OFF|ON]
```

- Set the report header.

```
TTITLE 'Salary|Report'
```

- Set the report footer.

```
BTITLE 'Confidential'
```

ORACLE

7-22

Copyright © Oracle Corporation, 2001. All rights reserved.

The TTITLE and BTITLE Commands

Use the **TTITLE command** to format page headers and the **BTITLE command** for footers. Footers appear at the bottom of the page.

The syntax for BTITLE and TTITLE is identical. Only the syntax for TTITLE is shown. You can use the vertical bar (|) to split the text of the title across several lines.

Syntax

```
TTI[TLE] | BTI[TLE] [text|OFF|ON]
```

In the syntax:

text represents the title text (enter single quotes if the text is more than one word).

OFF | ON toggles the title either off or on. It is not visible when turned off.

The TTITLE example on the slide sets the report header to display Salary centered on one line and Report centered below it. The BTITLE example sets the report footer to display Confidential. TTITLE automatically puts the date and a page number on the report.

The TTITLE and BTITLE Commands (continued)

Note: The slide gives an abridged syntax for TTITLE and BTITLE. Various options for TTITLE and BTITLE are covered in another SQL course.

Instructor Note

SQL*Plus 3.3 introduced the commands REPHEADER and REPFOOTER. REPHEADER places and formats a specified report header at the top of each report or lists the current REPHEADER definition. REPFOOTER places and formats a specified report footer at the bottom of each report or lists the current REPFOOTER definition.

Creating a Script File to Run a Report

1. Create and test the SQL `SELECT` statement.
2. Save the `SELECT` statement into a script file.
3. Load the script file into an editor.
4. Add formatting commands before the `SELECT` statement.
5. Verify that the termination character follows the `SELECT` statement.

ORACLE

7-24

Copyright © Oracle Corporation, 2001. All rights reserved.

Creating a Script File to Run a Report

You can either enter each of the *iSQL*Plus* commands at the SQL prompt or put all the commands, including the `SELECT` statement, in a **command (or script) file**. A typical script consists of at least one `SELECT` statement and several *iSQL*Plus* commands.

How to Create a Script File

1. Create the SQL `SELECT` statement at the SQL prompt. Ensure that the data required for the report is accurate before you save the statement to a file and apply formatting commands. Ensure that the relevant `ORDER BY` clause is included if you intend to use breaks.
2. Save the `SELECT` statement to a script file.
3. Edit the script file to enter the *iSQL*Plus* commands.
4. Add the required formatting commands before the `SELECT` statement. Be certain not to place *iSQL*Plus* commands within the `SELECT` statement.
5. Verify that the `SELECT` statement is followed by a run character, either a semicolon (;) or a slash (/).

Creating a Script File to Run a Report

6. Clear formatting commands after the `SELECT` statement.
7. Save the script file.
8. Load the script file into the *iSQL*Plus* text window, and click the Execute button.

ORACLE

7-25

Copyright © Oracle Corporation, 2001. All rights reserved.

How to Create a Script File (continued)

6. Add the format-clearing *iSQL*Plus* commands after the run character. Alternatively, you can store all the format-clearing commands in a reset file.
7. Save the script file with your changes.
8. Load the script file into the *iSQL*Plus* text window, and click the Execute button.

Guidelines

- You can include blank lines between *iSQL*Plus* commands in a script.
- If you have a lengthy *iSQL*Plus* or *SQL*Plus* command, you can continue it on the next line by ending the current line with a hyphen (-).
- You can abbreviate *iSQL*Plus* commands.
- Include reset commands at the end of the file to restore the original *iSQL*Plus* environment.

Note: `REM` represents a remark or comment in *iSQL*Plus*.

Sample Report

Fri Sep 28	Employee Report	page 1
Job Category	Employee	Salary
AC_ACCOUNT	Gietz	\$8,300.00
AC_MGR	Higgins	\$12,000.00
AD_ASST	Whalen	\$4,400.00
IT_PROG	Ernst	\$6,000.00
	Hunold	\$9,000.00
	Lorentz	\$4,200.00
MK_MAN	Hartstein	\$13,000.00
MK_REP	Fay	\$6,000.00
SA_MAN	Zlotkey	\$10,500.00
SA_REP	Abel	\$11,000.00
	Grant	\$7,000.00
	Taylor	\$8,600.00
Confidential		

...

ORACLE

7-26

Copyright © Oracle Corporation, 2001. All rights reserved.

Example

Create a **script file** to create a report that displays the job ID, last name, and salary for every employee whose salary is less than \$15,000. Add a centered, two-line header that reads “Employee Report” and a centered footer that reads “Confidential.” Rename the job title column to read “Job Category” split over two lines. Rename the employee name column to read “Employee.” Rename the salary column to read “Salary” and format it as \$2,500.00.

Example (continued)

```
SET FEEDBACK OFF
TTITLE 'Employee|Report'
BTITLE 'Confidential'
BREAK ON job_id
COLUMN job_id HEADING 'Job|Category'
COLUMN last_name HEADING 'Employee'
COLUMN salary HEADING 'Salary' FORMAT $99,999.99
REM ** Insert SELECT statement
SELECT  job_id, last_name, salary
FROM    employees
WHERE   salary < 15000
ORDER BY job_id, last_name
/
REM clear all formatting commands ...
SET FEEDBACK ON
COLUMN job_id CLEAR
COLUMN last_name CLEAR
COLUMN salary CLEAR
CLEAR BREAK
...
```

Summary

In this lesson, you should have learned how to:

- **Use *i*SQL*Plus substitution variables to store values temporarily**
- **Use `SET` commands to control the current *i*SQL*Plus environment**
- **Use the `COLUMN` command to control the display of a column**
- **Use the `BREAK` command to suppress duplicates and divide rows into sections**
- **Use the `TTITLE` and `BTITLE` commands to display headers and footers**

ORACLE

7-28

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary

In this lesson, you should have learned about substitution variables and how useful they are for running reports. They give you the flexibility to replace values in a `WHERE` clause, column names, and expressions. You can customize reports by writing script files with:

- Single ampersand substitution variables
- Double ampersand substitution variables
- The `DEFINE` command
- The `UNDEFINE` command
- Substitution variables in the command line

You can create a more readable report by using the following commands:

- `COLUMN`
- `TTITLE`
- `BTITLE`
- `BREAK`

Practice 7 Overview

This practice covers the following topics:

- **Creating a query to display values using substitution variables**
- **Starting a command file containing variables**

ORACLE

7-29

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 7 Overview

This practice gives you the opportunity to create files that can be run interactively by using substitution variables to create run-time selection criteria.

Practice 7

Determine whether the following two statements are true or false:

1. The following statement is valid:

```
DEFINE & p_val = 100
```

True/False

2. The DEFINE command is a SQL command.

True/False

3. Write a script to display the employee last name, job, and hire date for all employees who started between a given range. Concatenate the name and job together, separated by a space and comma, and label the column Employees. In a separate SQL script file, use the DEFINE command to provide the two ranges. Use the format MM/DD/YYYY. Save the script files as lab7_3a.sql and lab7_3b.sql.

```
DEFINE low_date = 01/01/1998
DEFINE high_date = 01/01/1999
```

EMPLOYEES	HIRE_DATE
Matos, ST_CLERK	15-MAR-98
Vargas, ST_CLERK	09-JUL-98
Taylor, SA_REP	24-MAR-98

4. Write a script to display the last names, job IDs, and department names for every employee in a given location. The search condition should allow for case-insensitive searches of the department location. Save the script file as lab7_4.sql.

EMPLOYEE NAME	JOB_ID	DEPARTMENT NAME
Whalen	AD_ASST	Administration
King	AD_PRES	Executive
Kochhar	AD_VP	Executive
De Haan	AD_VP	Executive
Higgins	AC_MGR	Accounting
Gietz	AC_ACCOUNT	Accounting

6 rows selected.

Practice 7 (continued)

5. Modify the code in `lab7_4.sql` to create a report containing the department name, employee last name, hire date, salary, and annual salary for each employee in a given location. Label the columns `DEPARTMENT NAME`, `EMPLOYEE NAME`, `START DATE`, `SALARY`, and `ANNUAL SALARY`, placing the labels on multiple lines. Resave the script as `lab7_5.sql`, and execute the commands in the script.

DEPARTMENT NAME	EMPLOYEE NAME	START DATE	SALARY	ANNUAL SALARY
Accounting	Higgins	07-JUN-94	\$12,000.00	\$144,000.00
	Gietz	07-JUN-94	\$8,300.00	\$99,600.00
Administration	Whalen	17-SEP-87	\$4,400.00	\$52,800.00
Executive	King	17-JUN-87	\$24,000.00	\$288,000.00
	Kochhar	21-SEP-89	\$17,000.00	\$204,000.00
	De Haan	13-JAN-93	\$17,000.00	\$204,000.00

8

Manipulating Data

ORACLE®

Copyright © Oracle Corporation, 2001. All rights reserved.

Schedule:	Timing	Topic
	60 minutes	Lecture
	30 minutes	Practice
	90 minutes	Total

Objectives

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

- Describe each DML statement
- Insert rows into a table
- Update rows in a table
- Delete rows from a table
- Merge rows in a table
- Control transactions

ORACLE

8-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Lesson Aim

In this lesson, you learn how 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 transactions with the **COMMIT**, **SAVEPOINT**, and **ROLLBACK** statements.

Data Manipulation Language

- A DML statement is executed when you:
 - Add new rows to a table
 - Modify existing rows in a table
 - Remove existing rows from a table
- A *transaction* consists of a collection of DML statements that form a logical unit of work.

ORACLE

8-3

Copyright © Oracle Corporation, 2001. All rights reserved.

Data Manipulation Language

Data manipulation language (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 form 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: decrease the savings account, increase the checking account, and record the transaction in the transaction journal. The Oracle server must guarantee that all 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.

Instructor Note

DML statements can be issued directly in *iSQL*Plus*, performed automatically by tools such as Oracle Forms Services, or programmed with tools such as the 3GL precompilers.

Every table has INSERT, UPDATE, and DELETE privileges associated with it. These privileges are automatically granted to the creator of the table, but in general they must be explicitly granted to other users.

Starting with Oracle 7.2, you can place a subquery in the place of the table name in an UPDATE statement, essentially the same way you use a view.

Adding a New Row to a Table

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		1700

...insert a new row
into the
DEPARTMENTS
table...

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		1700
70	Public Relations	100	1700

New
row

ORACLE

Adding a New Row to a Table

The slide graphic illustrates adding a new department to the DEPARTMENTS table.

The INSERT Statement Syntax

- Add new rows to a table by using the INSERT statement.

```
INSERT INTO  table [(column [, column...])]  
VALUES      (value [, value...]);
```

- Only one row is inserted at a time with this syntax.

ORACLE

8-5

Copyright © Oracle Corporation, 2001. All rights reserved.

Adding a New Row to a Table (continued)

You can add new rows to a table by issuing the **INSERT statement**.

In the syntax:

<i>table</i>	is the name of the table
<i>column</i>	is the name of the column in the table to populate
<i>value</i>	is the corresponding value for the column

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 row created.
```

- Enclose character and date values within single quotation marks.

ORACLE

8-6

Copyright © Oracle Corporation, 2001. All rights reserved.

Adding a New Row to a Table (continued)

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.

```
DESCRIBE departments
```

Name	Null?	Type
DEPARTMENT_ID	NOT NULL	NUMBER(4)
DEPARTMENT_NAME	NOT NULL	VARCHAR2(30)
MANAGER_ID		NUMBER(6)
LOCATION_ID		NUMBER(4)

For clarity, use the column list in the INSERT clause.

Enclose character and date values within single quotation marks; it is not recommended to enclose numeric values within single quotation marks.

Number values should not be enclosed in single quotes, because implicit conversion may take place for numeric values assigned to NUMBER data type columns if single quotes are included.

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 row created.
```

- **Explicit method: Specify the NULL keyword in the VALUES clause.**

```
INSERT INTO departments  
VALUES      (100, 'Finance', NULL, NULL);  
1 row created.
```

ORACLE

8-7

Copyright © Oracle Corporation, 2001. All rights reserved.

Methods for Inserting Null Values

Method	Description
Implicit	Omit the column from the column list.
Explicit	Specify the NULL keyword in the VALUES list, specify the empty string (' ') in the VALUES list for character strings and dates.

Be sure that you can use null values in the targeted column by verifying the Null? status with the *iSQL*Plus* **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.

Common errors that can occur during user input:

- Mandatory value missing for a NOT NULL column
- Duplicate value violates uniqueness constraint
- Foreign key constraint violated
- CHECK constraint violated
- Data type mismatch
- Value too wide to fit in column

Inserting Special Values

The **SYSDATE** function records the current date and time.

```
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',
        SYSDATE, 'AC_ACCOUNT', 6900,
        NULL, 205, 100);
```

1 row created.

ORACLE

8-8

Copyright © Oracle Corporation, 2001. All rights reserved.

Inserting Special Values by Using SQL Functions

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 **SYSDATE** function for current date and time.

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;
```

EMPLOYEE_ID	LAST_NAME	JOB_ID	HIRE_DATE	COMMISSION_PCT
113	Popp	AC_ACCOUNT	27-SEP-01	

Inserting Specific Date Values

- Add a new employee.

```
INSERT INTO employees
VALUES      (114,
             'Den', 'Rapehealy',
             'DRAPHEAL', '515.127.4561',
             TO_DATE('FEB 3, 1999', 'MON DD, YYYY'),
             'AC_ACCOUNT', 11000, NULL, 100, 30);
```

1 row created.

- Verify your addition.

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_P
114	Den	Rapehealy	DRAPHEAL	515.127.4561	03-FEB-99	AC_ACCOUNT	11000	

ORACLE

8-9

Copyright © Oracle Corporation, 2001. All rights reserved.

Inserting Specific Date and Time Values

The DD-MON-YY format is usually used to insert a date value. With this format, recall that the century defaults to the current century. Because the date also contains time information, the default time is midnight (00:00:00).

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 on the slide records information for employee Rapehealy in the EMPLOYEES table. It sets the HIRE_DATE column to be February 3, 1999. If you use the following statement instead of the one shown on the slide, the year of the hire_date is interpreted as 2099.

```
INSERT INTO employees
VALUES      (114,
             'Den', 'Rapehealy',
             'DRAPHEAL', '515.127.4561',
             '03-FEB-99',
             'AC_ACCOUNT', 11000, NULL, 100, 30);
```

If the RR format is used, the system provides the correct century automatically, even if it is not the current one.

Instructor Note

The default date format in Oracle9i is DD-MON-RR. Prior to release 8.16, the default format was DD-MON-YY.

Creating a Script

- Use & 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);
```

Define Substitution Variables

"department_id" 40
"department_name" Human Resources
"location" 2500

Submit for Execution

Cancel

1 row created.

ORACLE

8-10

Copyright © Oracle Corporation, 2001. All rights reserved.

Creating a Script to Manipulate Data

You can save commands with substitution variables to a file and execute the commands in the file. The example above records information for a department in the DEPARTMENTS table.

Run the script file and you are prompted for input for the & substitution variables. The values you input are then substituted into the statement. This allows you to run the same script file over and over, but supply a different set of values each time you run it.

Instructor Note

Be sure to mention the following points about the script:

- Do not prefix the *iSQL*Plus* substitution parameter with the ampersand in the DEFINE command.
- Use a dash to continue an *iSQL*Plus* command on the next line.

Copying Rows from Another Table

- Write your **INSERT** statement with a subquery.

```
INSERT INTO sales_reps(id, name, salary, commission_pct)
SELECT employee_id, last_name, salary, commission_pct
FROM employees
WHERE job_id LIKE '%REP%';
```

4 rows created.

- Do not use the **VALUES** clause.
- Match the number of columns in the **INSERT** clause to those in the subquery.

ORACLE

8-11

Copyright © Oracle Corporation, 2001. All rights reserved.

Copying Rows from Another Table

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, you use a subquery.

Syntax

```
INSERT INTO table [ column (, column) ] subquery;
```

In the syntax:

<i>table</i>	is the table name
<i>column</i>	is the name of the column in the table to populate
<i>subquery</i>	is the subquery that returns rows into the table

The number of columns and their data types in the column list of the **INSERT** clause must match the number of values and their data types in the subquery. To create a copy of the rows of a table, use **SELECT *** in the subquery.

```
INSERT INTO copy_emp
SELECT *
FROM employees;
```

For more information, see *Oracle9i SQL Reference*, “**SELECT**,” subqueries section.

Instructor Note


Please run the script `8_cretabs.sql` to create the `COPY_EMP` and `SALES_REPS` tables before demonstrating the code examples. Do not get into too many details on copying rows from another table.

Changing Data in a Table

EMPLOYEES

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	HIRE_DATE	JOB_ID	SALARY	DEPARTMENT_ID	COMMISSION_P
100	Steven	King	SKING	17-JUN-87	AD_PRES	24000	90	
101	Neena	Kochhar	NKOCHHAR	21-SEP-89	AD_VP	17000	90	
102	Lex	De Haan	LDEHAAN	13-JAN-93	AD_VP	17000	90	
103	Alexander	Hunold	AHUNOLD	03-JAN-90	IT_PROG	9000	60	
104	Bruce	Ernst	BERNST	21-MAY-91	IT_PROG	6000	60	
107	Diana	Lorentz	DLORENTZ	07-FEB-99	IT_PROG	4200	60	
124	Kevin	Mourgos	KMOURGOS	16-NOV-99	ST_MAN	5800	50	

Update rows in the **EMPLOYEES** table.



EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	HIRE_DATE	JOB_ID	SALARY	DEPARTMENT_ID	COMMISSION_P
100	Steven	King	SKING	17-JUN-87	AD_PRES	24000	90	
101	Neena	Kochhar	NKOCHHAR	21-SEP-89	AD_VP	17000	90	
102	Lex	De Haan	LDEHAAN	13-JAN-93	AD_VP	17000	90	
103	Alexander	Hunold	AHUNOLD	03-JAN-90	IT_PROG	9000	30	
104	Bruce	Ernst	BERNST	21-MAY-91	IT_PROG	6000	30	
107	Diana	Lorentz	DLORENTZ	07-FEB-99	IT_PROG	4200	30	
124	Kevin	Mourgos	KMOURGOS	16-NOV-99	ST_MAN	5800	50	

ORACLE

Changing Data in a Table

The slide graphic illustrates changing the department number for employees in department 60 to department 30.

The UPDATE Statement Syntax

- Modify existing rows with the UPDATE statement.

```
UPDATE      table
SET         column = value [, column = value, ...]
[WHERE      condition];
```

- Update more than one row at a time, if required.

ORACLE

8-13

Copyright © Oracle Corporation, 2001. All rights reserved.

Updating Rows

You can modify existing rows by using the **UPDATE statement**.

In the syntax:

<i>table</i>	is the name of the table
<i>column</i>	is the name of the column in the table to populate
<i>value</i>	is the corresponding value or subquery for the column
<i>condition</i>	identifies the rows to be updated and 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 *Oracle9i SQL Reference*, “UPDATE.”

Note: In general, use the primary key to identify a single row. Using other columns can unexpectedly cause several rows to be updated. For example, identifying a single row in the EMPLOYEES table by name is dangerous, because more than one employee may have the same name.

Updating Rows in a Table

- Specific row or rows are modified if you specify the WHERE clause.

```
UPDATE employees
SET    department_id = 70
WHERE  employee_id = 113;
1 row updated.
```

- All rows in the table are modified if you omit the WHERE clause.

```
UPDATE    copy_emp
SET       department_id = 110;
22 rows updated.
```

ORACLE

8-14

Copyright © Oracle Corporation, 2001. All rights reserved.

Updating Rows (continued)

The **UPDATE statement** modifies specific rows if the WHERE clause is specified. The slide example transfers employee 113 (Popp) to department 70.

If you omit the WHERE clause, all the rows in the table are modified.

```
SELECT last_name, department_id
FROM    copy_emp;
```

LAST_NAME	DEPARTMENT_ID
King	110
Kochhar	110
De Haan	110
Hunold	110
Ernst	110
Lorentz	110

■ ■ ■

22 rows selected.

Note: The COPY_EMP table has the same data as the EMPLOYEES table.

Updating Two Columns with a Subquery

Update employee 114's job and salary to match that of employee 205.

```
UPDATE employees
SET   job_id = (SELECT job_id
                FROM   employees
                WHERE  employee_id = 205),
      salary = (SELECT salary
                FROM   employees
                WHERE  employee_id = 205)
WHERE employee_id = 114;
1 row updated.
```

ORACLE

8-15

Copyright © Oracle Corporation, 2001. All rights reserved.

Updating Two Columns with a Subquery

You can update multiple columns in the **SET clause** of an UPDATE statement by writing multiple subqueries.

Syntax

```
UPDATE table
SET   column =
      (SELECT column
       FROM table
       WHERE condition)
[ ,
  column =
      (SELECT column
       FROM table
       WHERE condition) ]
[WHERE condition] ;
```

Note: If no rows are updated, a message “0 rows updated.” is returned.

Updating Rows Based on Another Table

Use subqueries in UPDATE statements to update rows 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 row updated.
```

ORACLE

8-16

Copyright © Oracle Corporation, 2001. All rights reserved.

Updating Rows Based on Another Table

You can use **subqueries in UPDATE statements** to update rows in a table. The example on 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.

Updating Rows: Integrity Constraint Error

```
UPDATE employees
SET    department_id = 55
WHERE  department_id = 110;
```

```
UPDATE employees
      *
ERROR at line 1:
ORA-02291: integrity constraint (HR.EMP_DEPT_FK)
violated - parent key not found
```

Department number 55 does not exist

ORACLE

8-17

Copyright © Oracle Corporation, 2001. All rights reserved.

Integrity Constraint Error

If you attempt to update a record with a value that is tied to an **integrity constraint**, an error is returned. In the example on the slide, department number 55 does not exist in the parent table, DEPARTMENTS, and so you receive the *parent key* violation ORA-02291.

Note: Integrity constraints ensure that the data adheres to a predefined set of rules. A subsequent lesson covers integrity constraints in greater depth.

Instructor Note

Explain integrity constraints, and review the concepts of primary key and foreign key.

Removing a Row from a Table

DEPARTMENTS

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
30	Purchasing		
100	Finance		
50	Shipping	124	1500
60	IT	103	1400

Delete a row from the DEPARTMENTS table.

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
30	Purchasing		
50	Shipping	124	1500
60	IT	103	1400

ORACLE

8-18

Copyright © Oracle Corporation, 2001. All rights reserved.

Removing a Row from a Table

The slide graphic removes the Finance department from the DEPARTMENTS table (assuming that there are no constraints defined on the DEPARTMENTS table).

Instructor Note

After all the rows have been eliminated with the DELETE statement, only the data structure of the table remains. A more efficient method of emptying a table is with 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 for the following reasons:

- The TRUNCATE statement is a data definition language (DDL) statement and generates no rollback information. It is covered in a subsequent lesson.
- Truncating a table does not fire the delete triggers of the table.
- If the table is the parent of a referential integrity constraint, you cannot truncate the table. Disable the constraint before issuing the TRUNCATE statement.

The DELETE Statement

You can remove existing rows from a table by using the DELETE statement.

```
DELETE [FROM]   table
[WHERE          condition];
```

ORACLE

8-19

Copyright © Oracle Corporation, 2001. All rights reserved.

Deleting Rows

You can remove existing rows by using the **DELETE statement**.

In the syntax:

<i>table</i>	is the table name
<i>condition</i>	identifies the rows to be deleted and is composed of column names, expressions, constants, subqueries, and comparison operators

Note: If no rows are deleted, a message “0 rows deleted.” is returned:

For more information, see *Oracle9i SQL Reference*, “DELETE.”

Instructor Note

The DELETE statement does not ask for confirmation. However, the delete operation is not made permanent until the data transaction is committed. Therefore, you can undo the operation with the ROLLBACK statement if you make a mistake.

Deleting Rows from a Table

- Specific rows are deleted if you specify the WHERE clause.

```
DELETE FROM departments
WHERE department_name = 'Finance';
1 row deleted.
```

- All rows in the table are deleted if you omit the WHERE clause.

```
DELETE FROM copy_emp;
22 rows deleted.
```

ORACLE

8-20

Copyright © Oracle Corporation, 2001. All rights reserved.

Deleting Rows (continued)

You can delete specific rows by specifying the WHERE clause in the **DELETE statement**. The slide example 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';

no rows selected.
```

If you omit the WHERE clause, all rows in the table are deleted. The second example on the slide deletes all the rows from the COPY_EMP table, because no WHERE clause has been specified.

Example

Remove rows identified in the WHERE clause.

```
DELETE FROM employees
WHERE employee_id = 114;
```

1 row deleted.

```
DELETE FROM departments
WHERE department_id IN (30, 40);
```

2 rows deleted.

Deleting Rows Based on Another Table

Use subqueries in **DELETE** statements to remove rows from a table based on values from another table.

```
DELETE FROM employees
WHERE department_id =
    (SELECT department_id
     FROM departments
     WHERE department_name LIKE '%Public%');

1 row deleted.
```

ORACLE

8-21

Copyright © Oracle Corporation, 2001. All rights reserved.

Deleting Rows Based on Another Table

You can use **subqueries to delete rows** from a table based on values from another table. The example on the slide deletes all the employees who are 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.

Deleting Rows: Integrity Constraint Error

```
DELETE FROM departments
WHERE      department_id = 60;
```

```
DELETE FROM departments
      *
ERROR at line 1:
ORA-02292: integrity constraint (HR.EMP_DEPT_FK)
violated - child record found
```

You cannot delete a row that contains a primary key that is used as a foreign key in another table.

ORACLE

8-22

Copyright © Oracle Corporation, 2001. All rights reserved.

Integrity Constraint Error

If you attempt to **delete a record with a value that is tied to an integrity constraint**, an error is returned. The example on the slide tries to delete department number 60 from the DEPARTMENTS table, but it results in an error because department number is used as a foreign key in the EMPLOYEES table. If the parent record that you attempt to delete has child records, then 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;
```

1 row deleted.

Instructor Note

If referential integrity constraints are in use, you may receive an Oracle server error message when you attempt to delete a row. However, if the referential integrity constraint contains the ON DELETE CASCADE option, then the selected row and its children are deleted from their respective tables.

Using a Subquery in an INSERT Statement

```
INSERT INTO
    (SELECT employee_id, last_name,
            email, hire_date, job_id, salary,
            department_id
     FROM   employees
     WHERE  department_id = 50)
VALUES (99999, 'Taylor', 'DTAYLOR',
        TO_DATE('07-JUN-99', 'DD-MON-RR'),
        'ST_CLERK', 5000, 50);

1 row created.
```

ORACLE

8-23

Copyright © Oracle Corporation, 2001. All rights reserved.

Using a Subquery in an INSERT Statement

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 could not put in a duplicate employee ID, nor leave out a value for a mandatory not null column.

Using a Subquery in an INSERT Statement

```
SELECT employee_id, last_name, email, hire_date,  
       job_id, salary, department_id  
FROM   employees  
WHERE  department_id = 50;
```

EMPLOYEE_ID	LAST_NAME	EMAIL	HIRE_DATE	JOB_ID	SALARY	DEPARTMENT_ID
124	Mourgos	KMOURGOS	16-NOV-99	ST_MAN	5800	50
141	Rajs	TRAJS	17-OCT-95	ST_CLERK	3500	50
142	Davies	CDAVIES	29-JAN-97	ST_CLERK	3100	50
143	Matos	RMATOS	15-MAR-98	ST_CLERK	2600	50
144	Vargas	PVARGAS	09-JUL-98	ST_CLERK	2500	50
99999	Taylor	DTAYLOR	07-JUN-99	ST_CLERK	5000	50

6 rows selected.

ORACLE

Using a Subquery in an INSERT Statement

The example shows the results of the subquery that was used to identify the table for the INSERT statement.

Using the WITH CHECK OPTION Keyword on DML Statements

- A subquery is used to identify the table and columns of the DML statement.
- The WITH CHECK OPTION keyword prohibits you from changing rows that are not in the subquery.

```
INSERT INTO (SELECT employee_id, last_name, email,
                hire_date, job_id, salary
            FROM employees
            WHERE department_id = 50 WITH CHECK OPTION)
VALUES (99998, 'Smith', 'JSMITH',
        TO_DATE('07-JUN-99', 'DD-MON-RR'),
        'ST_CLERK', 5000);
INSERT INTO
    *
ERROR at line 1:
ORA-01402: view WITH CHECK OPTION where-clause violation
```

ORACLE

The WITH CHECK OPTION Keyword

Specify **WITH CHECK OPTION** to indicate that, if the subquery is used in place of a table in an INSERT, UPDATE, or DELETE statement, no changes that would produce rows that are not included in the subquery are permitted to that table.

In the example shown, the WITH CHECK OPTION keyword is used. The subquery identifies rows that are in department 50, but the department ID is not in the SELECT list, and a value is not provided for it in the VALUES list. Inserting this row would result in a department ID of null, which is not in the subquery.

Overview of the Explicit Default Feature

- With the explicit default feature, you can use the **DEFAULT** keyword as a column value where the column default is desired.
- The addition of this feature is for compliance with the SQL: 1999 Standard.
- This allows the user to control where and when the default value should be applied to data.
- Explicit defaults can be used in **INSERT** and **UPDATE** statements.

ORACLE

8-26

Copyright © Oracle Corporation, 2001. All rights reserved.

Explicit Defaults

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.

Using Explicit Default Values

- **DEFAULT with INSERT:**

```
INSERT INTO departments
  (department_id, department_name, manager_id)
VALUES (300, 'Engineering', DEFAULT);
```

- **DEFAULT with UPDATE:**

```
UPDATE departments
SET manager_id = DEFAULT WHERE department_id = 10;
```

ORACLE

8-27

Copyright © Oracle Corporation, 2001. All rights reserved.

Using Explicit Default Values

Specify **DEFAULT** to set the column to the value previously specified as the default value for the column. If no default value for the corresponding column has been specified, Oracle sets the column to null.

In the first example shown, the **INSERT** statement uses a default value for the **MANAGER_ID** column. If there is no default value defined for the column, a null value is inserted instead.

The second example uses the **UPDATE** statement to set the **MANAGER_ID** column to a default value for department 10. If no default value is defined for the column, it changes the value to null.

Note: When creating a table, you can specify a default value for a column. This is discussed in the “Creating and Managing Tables” lesson.

The MERGE Statement

- Provides the ability to conditionally update or insert data into a database table
- Performs an UPDATE if the row exists, and an INSERT if it is a new row:
 - Avoids separate updates
 - Increases performance and ease of use
 - Is useful in data warehousing applications

ORACLE

8-28

Copyright © Oracle Corporation, 2001. All rights reserved.

MERGE Statements

SQL has been extended to include the **MERGE statement**. Using this statement, you can update or insert a row conditionally into a table, thus avoiding multiple UPDATE statements. The decision whether to update or insert into the target table is based on a condition in the ON clause.

Since the MERGE command combines the INSERT and UPDATE commands, you need both INSERT and UPDATE privileges on the target table and the SELECT privilege on the source 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.

The MERGE Statement Syntax

You can conditionally insert or update 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 = col_val1,
      col2 = col2_val
  WHEN NOT MATCHED THEN
    INSERT (column_list)
    VALUES (column_values);
```

ORACLE

8-29

Copyright © Oracle Corporation, 2001. All rights reserved.

Merging Rows

You can update existing rows and insert new rows conditionally by using 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 upon which the MERGE operation either updates or inserts
WHEN MATCHED	instructs the server how to respond to the results of the join condition
WHEN NOT MATCHED	

For more information, see *Oracle9i SQL Reference*, “MERGE.”

Merging Rows

Insert or update rows in the COPY_EMP table to match the EMPLOYEES table.

```
MERGE INTO copy_emp c
  USING 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.department_id  = e.department_id
  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);
```

ORACLE

8-30

Copyright © Oracle Corporation, 2001. All rights reserved.

Example of Merging Rows

```
MERGE INTO copy_emp c
  USING 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,
      c.commission_pct  = e.commission_pct,
      c.manager_id      = e.manager_id,
      c.department_id   = e.department_id
  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 example shown matches the EMPLOYEE_ID in the COPY_EMP table to the EMPLOYEE_ID in the EMPLOYEES table. If a match is found, the row in the COPY_EMP table is updated to match the row in the EMPLOYEES table. If the row is not found, it is inserted into the COPY_EMP table.

Merging Rows

```
SELECT *  
FROM COPY_EMP;  
  
no rows selected
```

```
MERGE INTO copy_emp c  
  USING employees e  
  ON (c.employee_id = e.employee_id)  
WHEN MATCHED THEN  
  UPDATE SET  
    ...  
WHEN NOT MATCHED THEN  
  INSERT VALUES...;
```

```
SELECT *  
FROM COPY_EMP;  
  
20 rows selected.
```

ORACLE

8-31

Copyright © Oracle Corporation, 2001. All rights reserved.

Example of Merging Rows

The condition `c.employee_id = e.employee_id` is evaluated. Because the `COPY_EMP` table is empty, 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_EMP` table.

If rows existed in the `COPY_EMP` table and employee IDs matched in both tables (the `COPY_EMP` and `EMPLOYEES` tables), the existing rows in the `COPY_EMP` table would be updated to match the `EMPLOYEES` table.

Instructor Note

In a data warehousing environment, you may have a large fact table and a smaller dimension table with rows that need to be inserted into the large fact table conditionally. The `MERGE` statement is useful in this situation.

You may want to have a break here.

Database Transactions

A database transaction consists of one of the following:

- **DML statements which constitute one consistent change to the data**
- **One DDL statement**
- **One DCL statement**

ORACLE

8-32

Copyright © Oracle Corporation, 2001. All rights reserved.

Database Transactions

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 make up one consistent change to the data. For example, a transfer of funds between two accounts should include the debit to one account and the credit to another account in the same amount. Both actions should either fail or succeed together; the credit should not be committed without the debit.

Transaction Types

Type	Description
Data manipulation language (DML)	Consists of any number of DML statements that the Oracle server treats as a single entity or a logical unit of work
Data definition language (DDL)	Consists of only one DDL statement
Data control language (DCL)	Consists of only one DCL statement

Database Transactions

- **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 *iSQL*Plus*
 - The system crashes

ORACLE

8-33

Copyright © Oracle Corporation, 2001. All rights reserved.

When Does a 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 *iSQL*Plus*
- A 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.

Instructor Note

Please run the script `8_cretest.sql` to create the test table and insert data into the table.

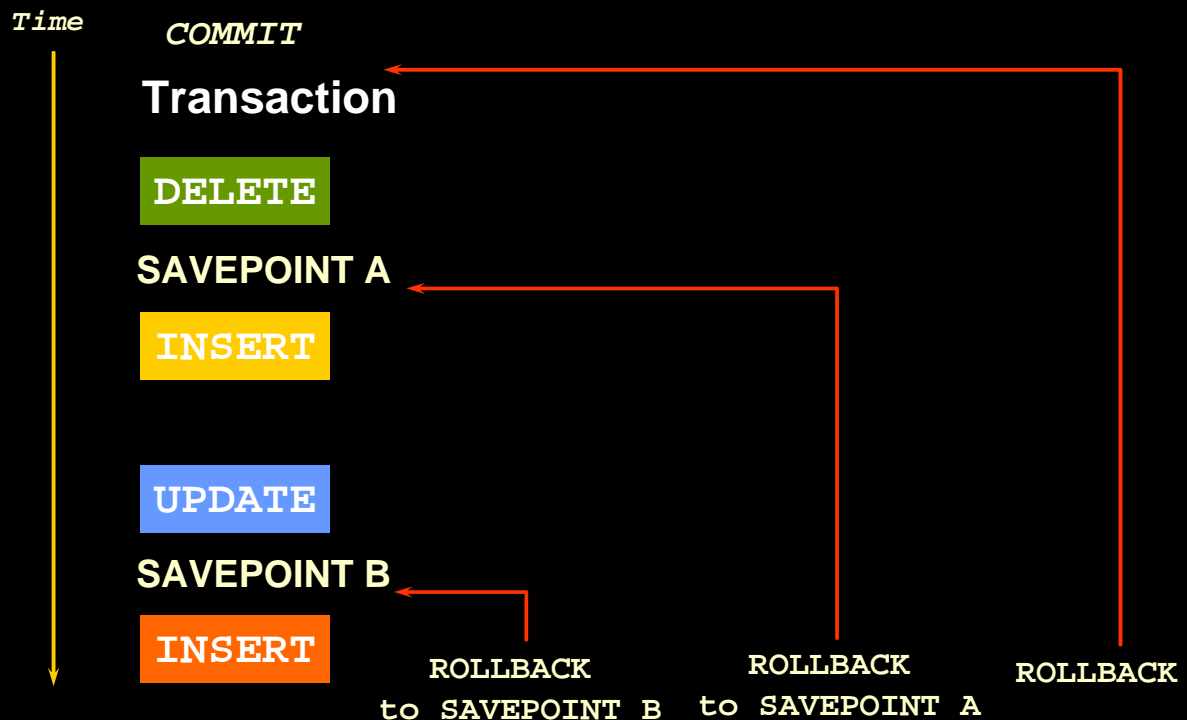
Advantages of COMMIT and ROLLBACK Statements

With COMMIT and ROLLBACK statements, you can:

- **Ensure data consistency**
- **Preview data changes before making changes permanent**
- **Group logically related operations**

ORACLE

Controlling Transactions



ORACLE

8-35

Copyright © Oracle Corporation, 2001. All rights reserved.

Explicit Transaction Control Statements

You can control the logic of transactions by using the **COMMIT**, **SAVEPOINT**, and **ROLLBACK** statements.

Statement	Description
COMMIT	Ends the current transaction by making all pending data changes permanent
SAVEPOINT <i>name</i>	Marks a savepoint within the current transaction
ROLLBACK	ROLLBACK ends the current transaction by discarding all pending data changes
ROLLBACK TO SAVEPOINT <i>name</i>	ROLLBACK TO SAVEPOINT rolls back the current transaction to the specified savepoint, thereby discarding any changes and or savepoints 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. As savepoints are logical, there is no way to list the savepoints you have created.

Note: SAVEPOINT is not ANSI standard SQL.

Instructor Note

Savepoints are not schema objects and cannot be referenced in the data dictionary.

Rolling Back Changes to a Marker

- Create a marker in a current transaction by using the **SAVEPOINT** statement.
- Roll back to that marker by using the **ROLLBACK TO SAVEPOINT** statement.

```
UPDATE...  
SAVEPOINT update_done;  
Savepoint created.  
INSERT...  
ROLLBACK TO update_done;  
Rollback complete.
```

ORACLE

8-36

Copyright © Oracle Corporation, 2001. All rights reserved.

Rolling Back Changes to a Savepoint

You can create a marker in the current transaction by using the **SAVEPOINT statement** which divides the transaction into smaller sections. You can then discard pending changes up to that marker by using the **ROLLBACK TO SAVEPOINT** statement.

If you create a second savepoint with the same name as an earlier savepoint, the earlier savepoint is deleted.

Instructor Note

Savepoints are especially useful in PL/SQL and 3GL programs in which recent changes can be undone conditionally based on run-time conditions.

Implicit Transaction Processing

- **An automatic commit occurs under the following circumstances:**
 - DDL statement is issued
 - DCL statement is issued
 - Normal exit from *iSQL*Plus*, without explicitly issuing **COMMIT** or **ROLLBACK** statements
- **An automatic rollback occurs under an abnormal termination of *iSQL*Plus* or a system failure.**

ORACLE

8-37

Copyright © Oracle Corporation, 2001. All rights reserved.

Implicit Transaction Processing

Status	Circumstances
Automatic commit	DDL statement or DCL statement is issued. <i>iSQL*Plus</i> exited normally, without explicitly issuing COMMIT or ROLLBACK commands.
Automatic rollback	Abnormal termination of <i>iSQL*Plus</i> or system failure.

Note: A third command is available in *iSQL*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. Also, the **COMMIT** statement is issued when a DDL statement is issued or when you exit from *iSQL*Plus*.

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 their state at the time of the last commit. In this way, the Oracle server protects the integrity of the tables.

From *iSQL*Plus*, a normal exit from the session is accomplished by clicking the Exit button. With *SQL*Plus*, a normal exit is accomplished by typing the command **EXIT** at the prompt. Closing the window is interpreted as an abnormal exit.

State of the Data Before COMMIT or ROLLBACK

- The previous state of the data can be recovered.
- The current user can review the results of the DML operations by using the `SELECT` statement.
- Other users *cannot* view the results of the DML statements by the current user.
- The affected rows are *locked*; other users cannot change the data within the affected rows.

ORACLE

8-38

Copyright © Oracle Corporation, 2001. All rights reserved.

Committing Changes

Every data change made during the transaction is temporary until the transaction is committed.

State of the data before `COMMIT` or `ROLLBACK` statements are issued:

- Data manipulation operations primarily affect the database buffer; therefore, the previous state of the data can be recovered.
- The current user can review the results of the data manipulation operations by querying the tables.
- Other users cannot view the results of the data manipulation operations made by the current user. The Oracle server institutes read consistency to ensure that each user sees data as it existed at the last commit.
- The affected rows are locked; other users cannot change the data in the affected rows.

Instructor Note

With the Oracle server, data changes can actually be written to the database files before transactions are committed, but they are still only temporary.

If a number of users are making changes simultaneously to the same table, then each user sees only his or her changes until other users commit their changes.

By default, the Oracle server has *row-level locking*. It is possible to alter the default locking mechanism.

State of the Data after COMMIT

- Data changes are made permanent in the database.
- The previous state of the data is permanently lost.
- All users can view the results.
- Locks on the affected rows are released; those rows are available for other users to manipulate.
- All savepoints are erased.

ORACLE

8-39

Copyright © Oracle Corporation, 2001. All rights reserved.

Committing Changes (continued)

Make all pending changes permanent by using the **COMMIT statement**. Following a COMMIT statement:

- Data changes are written to the database.
- The previous state of the data is permanently lost.
- All users can view the results of the transaction.
- The locks on the affected rows are released; the rows are now available for other users to perform new data changes.
- All savepoints are erased.

Committing Data

- **Make the changes.**

```
DELETE FROM employees
WHERE employee_id = 99999;
1 row deleted.

INSERT INTO departments
VALUES (290, 'Corporate Tax', NULL, 1700);
1 row inserted.
```

- **Commit the changes.**

```
COMMIT;
Commit complete.
```

ORACLE

8-40

Copyright © Oracle Corporation, 2001. All rights reserved.

Committing Changes (continued)

The slide example deletes a row from the EMPLOYEES table and inserts a new row into the DEPARTMENTS table. It then makes the change permanent by issuing the **COMMIT** statement.

Example

Remove departments 290 and 300 in the DEPARTMENTS table, and update a row in the COPY_EMP table. Make the data change permanent.

```
DELETE FROM departments
WHERE department_id IN (290, 300);

2 rows deleted.
```

```
UPDATE copy_emp
SET department_id = 80
WHERE employee_id = 206;

1 row updated.
```

```
COMMIT;
```

```
Commit Complete.
```

Instructor Note

Use this example to explain how COMMIT ensures that two related operations occur together or not at all. In this case, COMMIT prevents empty departments from being created.

State of the Data After ROLLBACK

Discard all pending changes by using the **ROLLBACK** statement:

- Data changes are undone.
- Previous state of the data is restored.
- Locks on the affected rows are released.

```
DELETE FROM copy_emp;  
22 rows deleted.  
ROLLBACK;  
Rollback complete.
```

ORACLE

8-41

Copyright © Oracle Corporation, 2001. All rights reserved.

Rolling Back Changes

Discard all pending changes by using the **ROLLBACK statement**. Following a ROLLBACK statement:

- Data changes are undone.
- The previous state of the data is restored.
- The locks on the affected rows are released.

Example

While attempting to remove a record from the TEST table, you can accidentally empty the table. You can correct the mistake, reissue the proper statement, and make the data change permanent.

```
DELETE FROM test;  
25,000 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.
```

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

8-42

Copyright © Oracle Corporation, 2001. All rights reserved.

Statement-Level Rollbacks

Part of a transaction can be discarded by 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**, but 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.

Oracle issues an implicit commit before and after any data definition language (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.

Instructor Note

The Oracle server implements locks on data to provide data concurrency in the database. Those locks are released when certain events occur (such as a system failure) or when the transaction is completed. Implicit locks on the database are obtained when a DML statement is successfully executed. The Oracle Server locks data at the lowest level possible by default .
Manually acquire locks on the database tables by executing a LOCK TABLE statement or the SELECT statement with the FOR UPDATE clause.

Starting with Oracle9i, the DBA has the choice of managing undo segments or having Oracle automatically manage undo data in an undo tablespace.

Please read the Instructor Note on page 8-52.

For more information on locking, refer to *Oracle9i Concepts*, “Data Concurrency and Consistency.”

Read Consistency

- Read consistency guarantees a consistent view of the data at all times.
- Changes made by one user do not conflict with 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.

ORACLE

8-43

Copyright © Oracle Corporation, 2001. All rights reserved.

Read Consistency

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 way.
- Changes made by one writer do not disrupt or conflict with changes another writer is making.

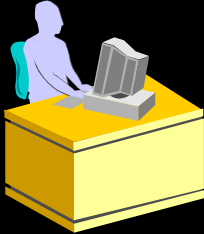
The purpose of read consistency is to ensure that each user sees data as it existed at the last commit, before a DML operation started.

Instructor Note

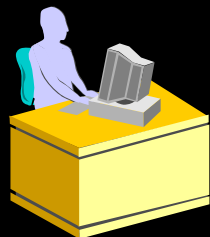
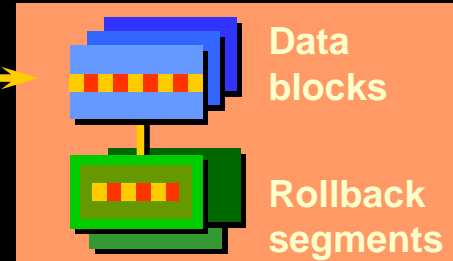
Please read the Instructor note on page 8-53.

Implementation of Read Consistency

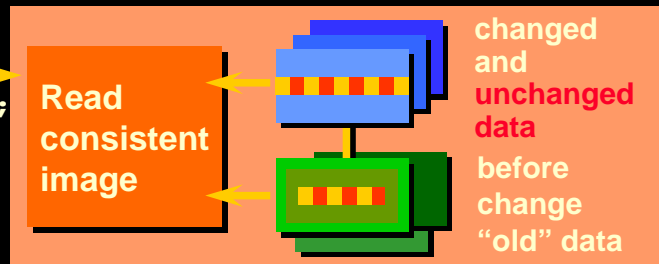
User A



```
UPDATE employees  
SET salary = 7000  
WHERE last_name = 'Goyal';
```



```
SELECT *  
FROM userA.employees;
```



User B

ORACLE

8-44

Copyright © Oracle Corporation, 2001. All rights reserved.

Implementation of Read Consistency

Read consistency is an automatic implementation. It keeps a partial copy of the database in undo segments.

When an insert, update, or delete operation is made to the database, the Oracle server takes a copy of the data before it is changed and writes it to a *undo segment*.

All readers, except the one who issued the change, still see the database as it existed before the changes started; they view the rollback segment's "snapshot" of the data.

Before 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 executing a `SELECT` statement. 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.

Instructor Note

When you commit a transaction, the Oracle server releases the rollback information but does not immediately destroy it. The information remains in the undo segment to create read-consistent views of pertinent data for queries that started before the transaction committed.

Starting with Oracle9i, the DBA has the choice of managing undo segments or having Oracle automatically manage undo data in an undo tablespace. This is discussed in the DBA courses.

Locking

In an Oracle database, locks:

- **Prevent destructive interaction between concurrent transactions**
- **Require no user action**
- **Automatically use the lowest level of restrictiveness**
- **Are held for the duration of the transaction**
- **Are of two types: explicit locking and implicit locking**

ORACLE®

8-45

Copyright © Oracle Corporation, 2001. All rights reserved.

What Are Locks?

Locks are mechanisms that prevent destructive interaction between transactions accessing the same resource, either a user object (such as tables or rows) or a system object not visible to users (such as shared data structures and data dictionary rows).

How the Oracle Database Locks Data

Oracle locking is performed automatically and requires no user action. Implicit locking occurs for SQL statements as necessary, depending on the action requested. Implicit locking occurs for all SQL statements except `SELECT`.

The users can also lock data manually, which is called explicit locking.

Instructor Note

[See the Instructor Note on page 8-52.](#)

Implicit Locking

- **Two lock modes:**
 - **Exclusive:** Locks out other users
 - **Share:** Allows other users to access
- **High level of data concurrency:**
 - **DML:** Table share, row exclusive
 - **Queries:** No locks required
 - **DDL:** Protects object definitions
- **Locks held until commit or rollback**

ORACLE

8-46

Copyright © Oracle Corporation, 2001. All rights reserved.

DML Locking

When performing data manipulation language (DML) operations, the Oracle server provides data concurrency through DML locking. DML locks occur at two levels:

- A **share lock** is automatically obtained at the table level during DML operations. With share lock mode, several transactions can acquire share locks on the same resource.
- An **exclusive lock** is acquired automatically for each row modified by a DML statement. Exclusive locks prevent the row from being changed by other transactions until the transaction is committed or rolled back. This lock ensures that no other user can modify the same row at the same time and overwrite changes not yet committed by another user.
- DDL locks occur when you modify a database object such as a table.

Instructor Note

A `SELECT . . . FOR UPDATE` statement also implements a lock. This is covered in the Oracle9i PL/SQL course.

Summary

In this lesson, you should have learned how to use DML statements and control transactions.

Statement	Description
INSERT	Adds a new row to the table
UPDATE	Modifies existing rows in the table
DELETE	Removes existing rows from the table
MERGE	Conditionally inserts or updates data in a table
COMMIT	Makes all pending changes permanent
SAVEPOINT	Is used to rollback to the savepoint marker
ROLLBACK	Discards all pending data changes

ORACLE

8-47

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary

In this lesson, you should have learned how to manipulate data in the Oracle database by using the INSERT, UPDATE, and DELETE statements. Control data changes by using the COMMIT, SAVEPOINT, and ROLLBACK statements.

The Oracle server guarantees a consistent view of data at all times.

Locking can be implicit or explicit.

Practice 8 Overview

This practice covers the following topics:

- **Inserting rows into the tables**
- **Updating and deleting rows in the table**
- **Controlling transactions**

ORACLE

8-48

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 8 Overview

In this practice, you add rows to the MY_EMPLOYEE table, update and delete data from the table, and control your transactions.

Practice 8

Insert data into the MY_EMPLOYEE table.

1. Run the statement in the lab8_1.sql script to build the MY_EMPLOYEE table to be used for the lab.
2. Describe the structure of the MY_EMPLOYEE table to identify the column names.

Name	Null?	Type
ID	NOT NULL	NUMBER(4)
LAST_NAME		VARCHAR2(25)
FIRST_NAME		VARCHAR2(25)
USERID		VARCHAR2(8)
SALARY		NUMBER(9,2)

3. Add the first row of data to the MY_EMPLOYEE table from the following sample data. Do not list the columns in the INSERT clause.

ID	LAST_NAME	FIRST_NAME	USERID	SALARY
1	Patel	Ralph	rpatel	895
2	Dancs	Betty	bdancs	860
3	Biri	Ben	bbiri	1100
4	Newman	Chad	cnewman	750
5	Ropeburn	Audrey	aropebur	1550

4. Populate the MY_EMPLOYEE table with the second row of sample data from the preceding list. This time, list the columns explicitly in the INSERT clause.
5. Confirm your addition to the table.

ID	LAST_NAME	FIRST_NAME	USERID	SALARY
1	Patel	Ralph	rpatel	895
2	Dancs	Betty	bdancs	860

Practice 8 (continued)

- Write an insert statement in a text file named `loademp.sql` to load rows into the `MY_EMPLOYEE` table. Concatenate the first letter of the first name and the first seven characters of the last name to produce the user ID.
- Populate the table with the next two rows of sample data by running the insert statement in the script that you created.
- Confirm your additions to the table.

ID	LAST_NAME	FIRST_NAME	USERID	SALARY
1	Patel	Ralph	rpatel	895
2	Dancs	Betty	bdancs	860
3	Biri	Ben	bbiri	1100
4	Newman	Chad	cnewman	750

- Make the data additions permanent.

Update and delete data in the `MY_EMPLOYEE` table.

- Change the last name of employee 3 to Drexler.
- Change the salary to 1000 for all employees with a salary less than 900.
- Verify your changes to the table.

ID	LAST_NAME	FIRST_NAME	USERID	SALARY
1	Patel	Ralph	rpatel	1000
2	Dancs	Betty	bdancs	1000
3	Drexler	Ben	bbiri	1100
4	Newman	Chad	cnewman	1000

- Delete Betty Dancs from the `MY_EMPLOYEE` table.
- Confirm your changes to the table.

ID	LAST_NAME	FIRST_NAME	USERID	SALARY
1	Patel	Ralph	rpatel	1000
3	Drexler	Ben	bbiri	1100
4	Newman	Chad	cnewman	1000

Practice 8 (continued)

15. Commit all pending changes.

Control data transaction to the MY_EMPLOYEE table.

16. Populate the table with the last row of sample data by modifying the statements in the script that you created in step 6. Run the statements in the script.
17. Confirm your addition to the table.

ID	LAST_NAME	FIRST_NAME	USERID	SALARY
1	Patel	Ralph	rpatel	1000
3	Drexler	Ben	bbiri	1100
4	Newman	Chad	cnewman	1000
5	Ropeburn	Audrey	aropebur	1550

18. Mark an intermediate point in the processing of the transaction.
19. Empty the entire table.
20. Confirm that the table is empty.
21. Discard the most recent DELETE operation without discarding the earlier INSERT operation.
22. Confirm that the new row is still intact.

ID	LAST_NAME	FIRST_NAME	USERID	SALARY
1	Patel	Ralph	rpatel	1000
3	Drexler	Ben	bbiri	1100
4	Newman	Chad	cnewman	1000
5	Ropeburn	Audrey	aropebur	1550

23. Make the data addition permanent.

Instructor Note (for page 8-42 - 8-45)

Demo: 8_select.sql

Purpose: To illustrate the concept that a reader does not lock another reader

Login to iSQL*Plus using the teach/oracle account.

Login to iSQL*Plus using an unused oraxx/oracle account.

Run the 8_select.sql script from the teach/oracle account. (This script selects all records from the DEPARTMENTS table).

Run the 8_select.sql script in the oraxx/oracle account. (This script selects all records from the DEPARTMENTS table).

In both the logins, the script executes successfully. This demonstrates the concept: *a reader does not lock another reader.*

Demo: 8_grant.sql, 8_update.sql, 8_select.sql

Purpose: To illustrate that a writer does not lock a reader

Run the 8_grant.sql script in the teach/oracle account. (This script grants SELECT and UPDATE privileges on the DEPARTMENTS table to the oraxx account).

Run the 8_update.sql script in the teach/oracle account. (This script updates the DEPARTMENTS table, changing location of the department ID 20 to location 1500. The update places a lock on the DEPARTMENTS table).

Run the 8_select.sql script in the teach/oracle account. (This script selects all records from the DEPARTMENTS table. Observe that the location of department ID 20 is changed to location ID 1500).

Run the 8_select.sql script in the oraxx/oracle account. (This script selects all records from the DEPARTMENTS table).

Observe that the script executes successfully in the oraxx/oracle account, but the location for department ID 20 still has the location ID of 1800. This demonstrates the concept: *a writer does not lock a reader.*

Demo: 8_update.sql, 8_rollback.sql, 8_select.sql

Purpose: To illustrate that a writer locks another writer

Run the 8_update.sql script in the oraxx/oracle account. (The script does not execute because the DEPARTMENTS table is locked by the teach/oracle account.)

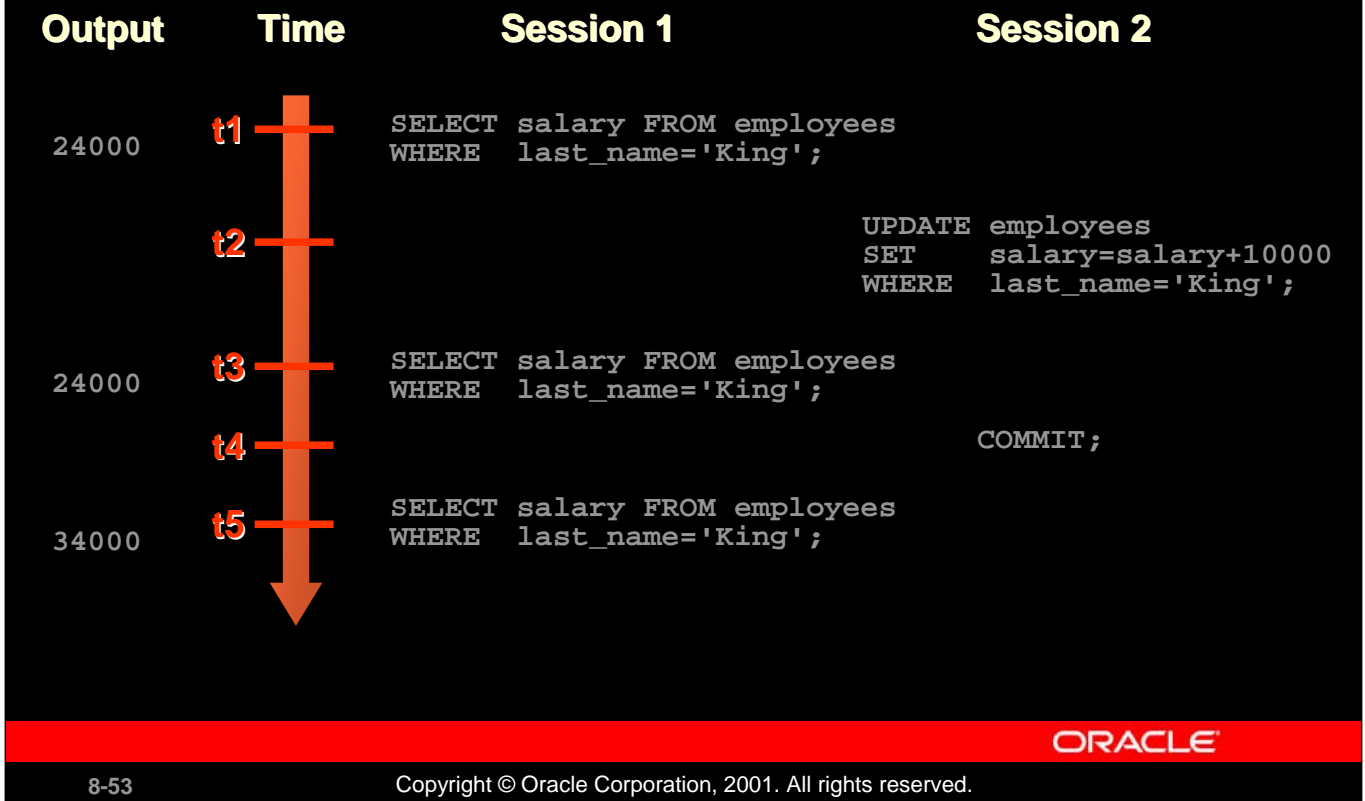
Switch to the teach/oracle account and run the 8_rollback.sql script. (This script rolls back the transaction, thereby releasing the lock on the DEPARTMENTS table.)

Switch to the oraxx/oracle account. You see that the 8_update.sql script has executed successfully because the lock on the DEPARTMENTS table has been released

Run the 8_select.sql script in the oraxx/oracle account. (This script selects all records from the DEPARTMENTS table. Observe that the location of department ID 20 is changed to location ID 1500.)

This demonstrates the concept: *a writer locks a writer.*

Read Consistency Example



ORACLE

8-53

Copyright © Oracle Corporation, 2001. All rights reserved.

Instructor Note (for page 8-43)

Read Consistency Example

For the duration of a SQL statement, read consistency guarantees that the selected data is consistent to the time point when the processing of the statement started.

Oracle server keeps noncommitted data in data blocks of undo segments (before images). As long as the changes are not committed, all users see the original data. The Oracle server uses data of both table segments and undo segments to generate a read-consistent view on the data.

In the example in the slide, the update of session 2 is not visible to process 1 until session 2 has committed the update (from t4 on). For the select statement at time point t3, the salary of King must be read from a data block of a undo segment that belongs to the transaction of session 2.

9

Creating and Managing Tables

ORACLE

Copyright © Oracle Corporation, 2001. All rights reserved.

Schedule:

Timing

30 minutes

20 minutes

50 minutes

Topic

Lecture

Practice

Total

Objectives

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

- **Describe the main database objects**
- **Create tables**
- **Describe the data types that can be used when specifying column definition**
- **Alter table definitions**
- **Drop, rename, and truncate tables**

ORACLE

9-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Lesson Aim

In this lesson, you learn about tables, the main database objects, and their relationships to each other. You also learn how to **create**, alter, and drop tables.

Database Objects

Object	Description
Table	Basic unit of storage; composed of rows and columns
View	Logically represents subsets of data from one or more tables
Sequence	Numeric value generator
Index	Improves the performance of some queries
Synonym	Gives alternative names to objects

ORACLE

9-3

Copyright © Oracle Corporation, 2001. All rights reserved.

Database Objects

An 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:** Subset of data from one or more tables
- **Sequence:** Numeric value generator
- **Index:** Improves the performance of some queries
- **Synonym:** Gives alternative names to objects

Oracle9i Table Structures

- Tables can be created at any time, even while users are using the database.
- You do not need to specify the size of any 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.
- Table structure can be modified online.

Note: More database objects are available but are not covered in this course.

Instructor Note

Tables can have up to 1,000 columns and must conform to standard database object-naming conventions. Column definitions can be omitted when using the AS subquery clause. Tables are created without data unless a query is specified. Rows are usually added by using INSERT statements.

Naming Rules

Table names and column names:

- Must begin with a letter
- Must be 1–30 characters long
- Must contain only A–Z, a–z, 0–9, _, \$, and #
- Must not duplicate the name of another object owned by the same user
- Must not be an Oracle server reserved word

ORACLE

9-4

Copyright © Oracle Corporation, 2001. All rights reserved.

Naming Rules

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.

Naming Guidelines

Use descriptive names for tables and other database objects.

Note: Names are case insensitive. For example, EMPLOYEES is treated as the same name as eMPLOYEES or eMpLOYEES.

For more information, see *Oracle9i SQL Reference*, “Object Names and Qualifiers.”

The CREATE TABLE Statement

- You must have:
 - CREATE TABLE privilege
 - A storage area

```
CREATE TABLE [schema.]table
              (column datatype [DEFAULT expr][, ...]);
```

- You specify:
 - Table name
 - Column name, column data type, and column size

ORACLE

9-5

Copyright © Oracle Corporation, 2001. All rights reserved.

The CREATE TABLE Statement

Create tables to store data by executing the SQL **CREATE TABLE statement**. This statement is one of the **data definition language (DDL) statements**, that are covered in subsequent lessons. DDL statements are a subset of SQL statements used to create, modify, or remove Oracle9i **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 uses **data control language (DCL) statements**, which are covered in a later lesson, to grant privileges to users.

In the syntax:

<i>schema</i>	is the same as the owner's name
<i>table</i>	is the name of the table
DEFAULT <i>expr</i>	specifies a default value if a value is omitted in the INSERT statement
<i>column</i>	is the name of the column
<i>datatype</i>	is the column's data type and length

Instructor Note

Please read the Instructor note on page 9-37

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.**

ORACLE

9-6

Copyright © Oracle Corporation, 2001. All rights reserved.

Referencing Another User's Tables

A *schema* is a collection of objects. Schema objects are the logical structures that directly refer to the data in a database. Schema objects 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 is a schema named USER_B, and USER_B has an EMPLOYEES table, then specify the following to retrieve data from that table:

```
SELECT *  
FROM   user_b.employees;
```


The DEFAULT Option

- Specify a default value for a column during an insert.

```
... 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.

ORACLE

9-7

Copyright © Oracle Corporation, 2001. All rights reserved.

The DEFAULT Option

A column can be given a default value by using the **DEFAULT option**. This option prevents null values from entering the columns if 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** and **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.

Note: CURRVAL and NEXTVAL are explained later.

Instructor Note

Here is an example for a pseudocolumn. For each row returned by a query, the ROWNUM pseudocolumn returns a number indicating the order in which Oracle server selects the row from a table or set of joined rows. The first row selected has a ROWNUM of 1, the second has 2, and so on.

The default value works with the DEFAULT keyword for INSERT and UPDATE statements discussed in the “Manipulating Data” lesson.

Creating Tables

- Create the table.

```
CREATE TABLE dept
      (deptno  NUMBER(2),
       dname   VARCHAR2(14),
       loc     VARCHAR2(13));
```

Table created.

- Confirm table creation.

```
DESCRIBE dept
```

Name	Null?	Type
DEPTNO		NUMBER(2)
DNAME		VARCHAR2(14)
LOC		VARCHAR2(13)

ORACLE

9-8

Copyright © Oracle Corporation, 2001. All rights reserved.

Creating Tables

The example on the slide creates the DEPT table, with three columns: DEPTNO, DNAME, and LOC. It further confirms the creation of the table by issuing the DESCRIBE command.

Since creating a table is a **DDL statement**, an automatic **commit** takes place when this statement is executed.

Instructor Note

Explain that additional syntax for CREATE TABLE could include constraints and so on. For more information on the CREATE TABLE syntax, refer to: *Oracle9i SQL Reference*, “CREATE TABLE.”

Tables in the Oracle Database

- **User Tables:**
 - Are a collection of tables created and maintained by the user
 - Contain user information
- **Data Dictionary:**
 - Is a collection of tables created and maintained by the Oracle Server
 - Contain database information

ORACLE

9-9

Copyright © Oracle Corporation, 2001. All rights reserved.

Tables in the Oracle Database

User tables are tables created by the user, 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.

All data dictionary tables are owned by the **SYS user**. The base tables are rarely accessed by the user because the information in them is not easy to understand. Therefore, users typically access data dictionary views because the information is presented in a format that is easier to understand.

Information stored in the data dictionary includes names of the Oracle server users, privileges granted to users, database object names, table constraints, and auditing information.

There are four categories of data dictionary views; each category has a distinct prefix that reflects its intended use.

Prefix	Description
USER_	These views contain information about objects owned by the user
ALL_	These views contain information about all of the tables (object tables and relational tables) accessible to the user.
DBA_	These views are restricted views, which can be accessed only by people who have been assigned the DBA role.
V\$	These views are dynamic performance views, database server performance, memory, and locking.

Querying the Data Dictionary

- See the names of tables owned by the user.

```
SELECT table_name  
FROM user_tables ;
```

- View distinct object types owned by the user.

```
SELECT DISTINCT object_type  
FROM user_objects ;
```

- View tables, views, synonyms, and sequences owned by the user.

```
SELECT *  
FROM user_catalog ;
```

ORACLE

9-10

Copyright © Oracle Corporation, 2001. All rights reserved.

Querying the Data Dictionary

You can query the data dictionary tables to view various database objects owned by you. The data dictionary tables frequently used are these:

- USER_TABLES
- USER_OBJECTS
- USER_CATALOG

Note: USER_CATALOG has a synonym called CAT. You can use this synonym instead of USER_CATALOG in SQL statements.

```
SELECT *  
FROM CAT ;
```

Data Types

Data Type	Description
<code>VARCHAR2(size)</code>	Variable-length character data
<code>CHAR(size)</code>	Fixed-length character data
<code>NUMBER(p,s)</code>	Variable-length numeric data
<code>DATE</code>	Date and time values
<code>LONG</code>	Variable-length character data up to 2 gigabytes
<code>CLOB</code>	Character data up to 4 gigabytes
<code>RAW</code> and <code>LONG RAW</code>	Raw binary data
<code>BLOB</code>	Binary data up to 4 gigabytes
<code>BFILE</code>	Binary data stored in an external file; up to 4 gigabytes
<code>ROWID</code>	A 64 base number system representing the unique address of a row in its table.

ORACLE

9-11

Copyright © Oracle Corporation, 2001. All rights reserved.

Data Types

Data type	Description
<code>VARCHAR2(size)</code>	Variable-length character data (a maximum <i>size</i> must be specified: Minimum <i>size</i> is 1; maximum <i>size</i> is 4000)
<code>CHAR [(size)]</code>	Fixed-length character data of length <i>size</i> bytes (default and minimum <i>size</i> is 1; maximum <i>size</i> is 2000)
<code>NUMBER [(p,s)]</code>	Number having precision <i>p</i> and scale <i>s</i> (The precision is the total number of decimal digits, and the scale is the number of digits to the right of the decimal point; the precision can range from 1 to 38 and the scale can range from -84 to 127)
<code>DATE</code>	Date and time values to the nearest second between January 1, 4712 B.C., and December 31, 9999 A.D.
<code>LONG</code>	Variable-length character data up to 2 gigabytes
<code>CLOB</code>	Character data up to 4 gigabytes

Data Types (continued)

Data type	Description
RAW(<i>size</i>)	Raw binary data of length <i>size</i> (a maximum <i>size</i> must be specified. maximum <i>size</i> is 2000)
LONG RAW	Raw binary data of variable length up to 2 gigabytes
BLOB	Binary data up to 4 gigabytes
BFILE	Binary data stored in an external file; up to 4 gigabytes
ROWID	A 64 base number system representing the unique address of a row in its table.

- 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 may want to use a CLOB column rather than a LONG column.

Instructor Note

Oracle8 introduced large object (LOB) data types that can store large and unstructured data such as text, image, video, and spatial data, up to 4 gigabytes in size. In Oracle9i, LONG columns can be easily migrated to LOB columns. Refer students to *Oracle9i Migration Release 9.0.1 Guide*.

Instructor Note (for page 9-13)

The date and time data types shown on the next page are new to release Oracle9i.

DateTime Data Types

Datetime enhancements with Oracle9i:

- New Datetime data types have been introduced.
- New data type storage is available.
- Enhancements have been made to time zones and local time zone.

Data Type	Description
TIMESTAMP	Date with fractional seconds
INTERVAL YEAR TO MONTH	Stored as an interval of years and months
INTERVAL DAY TO SECOND	Stored as an interval of days to hours minutes and seconds

ORACLE

9-13

Copyright © Oracle Corporation, 2001. All rights reserved.

Other DateTime Data Types

Data Type	Description
TIMESTAMP	Allows the time to be stored as a date with fractional seconds. There are several variations of the data type.
INTERVAL YEAR TO MONTH	Allows time to be stored as an interval of years and months. Used to represent the difference between two datetime values, where the only significant portions are the year and month.
INTERVAL DAY TO SECOND	Allows time to be stored as an interval of days to hours, minutes, and seconds. Useful in representing the precise difference between two datetime values.

DateTime Data Types

- The **TIMESTAMP data** type is an extension of the **DATE** data type.
- It stores the year, month, and day of the **DATE** data type, plus hour, minute, and second values as well as the fractional second value.
- The **TIMESTAMP** data type is specified as follows:

```
TIMESTAMP[(fractional_seconds_precision)]
```

ORACLE

9-14

Copyright © Oracle Corporation, 2001. All rights reserved.

DateTime Data Types

The `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 to 9. The default is 6.

Example

```
CREATE TABLE new_employees
(employee_id NUMBER,
 first_name VARCHAR2(15),
 last_name VARCHAR2(15),
 ...
 start_date TIMESTAMP(7),
 ...);
```

In the preceding example, we are creating a table `NEW_EMPLOYEES` with a column `start_date` with a data type of `TIMESTAMP`. The precision of '7' indicates the fractional seconds precision which if not specified defaults to '6'.

Assume that two rows are inserted into the `NEW_EMPLOYEES` table. The output shows the differences in the display. (A `DATE` data type defaults to display the format of `DD-MON-RR`):

```
SELECT start_date
FROM   new_employees;

17-JUN-87 12.00.00.000000 AM
21-SEP-89 12.00.00.000000 AM
```


TIMESTAMP WITH TIME ZONE Data Type

- **TIMESTAMP WITH TIME ZONE** is a variant of **TIMESTAMP** that includes a time zone displacement in its value.
- The time zone displacement is the difference, in hours and minutes, between local time and UTC.

```
TIMESTAMP[(fractional_seconds_precision)]  
WITH TIME ZONE
```

ORACLE

9-15

Copyright © Oracle Corporation, 2001. All rights reserved.

Datetime Data Types

UTC stands for **Coordinated Universal Time**—formerly Greenwich Mean Time. Two **TIMESTAMP WITH TIME ZONE** values are considered identical if they represent the same instant in UTC, regardless of the **TIME ZONE** offsets stored in the data.

Because **TIMESTAMP WITH TIME ZONE** can also store time zone information, it is particularly suited for recording date information that must be gathered or coordinated across geographic regions.

For example,

```
TIMESTAMP '1999-04-15 8:00:00 -8:00'
```

is the same as

```
TIMESTAMP '1999-04-15 11:00:00 -5:00'
```

That is, 8:00 a.m. Pacific Standard Time is the same as 11:00 a.m. Eastern Standard Time.

This can also be specified as

```
TIMESTAMP '1999-04-15 8:00:00 US/Pacific'
```

Note: **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 to 9. The default is 6.

TIMESTAMP WITH LOCAL TIME Data Type

- **TIMESTAMP WITH LOCAL TIME ZONE** is another variant of **TIMESTAMP** that includes a time zone displacement in its value.
- Data stored in the database is normalized to the database time zone.
- The time zone displacement is not stored as part of the column data; Oracle returns the data in the users' local session time zone.
- **TIMESTAMP WITH LOCAL TIME ZONE** data type is specified as follows:

```
TIMESTAMP[(fractional_seconds_precision)]  
WITH LOCAL TIME ZONE
```

ORACLE

9-16

Copyright © Oracle Corporation, 2001. All rights reserved.

DateTime Data Types

Unlike **TIMESTAMP WITH TIME ZONE**, you can specify columns of type **TIMESTAMP WITH LOCAL TIME ZONE** as part of a primary or unique key. The time zone displacement is the difference (in hours and minutes) between local time and UTC. There is no literal for **TIMESTAMP WITH LOCAL TIME ZONE**.

Note: `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 to 9. The default is 6.

Example

```
CREATE TABLE time_example  
  (order_date TIMESTAMP WITH LOCAL TIME ZONE);  
  
INSERT INTO time_example VALUES('15-NOV-00 09:34:28 AM');  
  
SELECT *  
FROM   time_example;  
  
order_date  
-----  
15-NOV-00 09.34.28.000000 AM
```

The **TIMESTAMP WITH LOCAL TIME ZONE** type is appropriate for two-tier applications where you want to display dates and times using the time zone of the client system.

INTERVAL YEAR TO MONTH Data Type

- **INTERVAL YEAR TO MONTH stores a period of time using the YEAR and MONTH datetime fields.**

```
INTERVAL YEAR [(year_precision)] TO MONTH
```

```
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 digits.

ORACLE

9-17

Copyright © Oracle Corporation, 2001. All rights reserved.

INTERVAL YEAR TO MONTH Data Type

INTERVAL YEAR TO MONTH stores a period of time using the YEAR and MONTH datetime fields. Use INTERVAL YEAR TO MONTH to represent the difference between two datetime values, where the only significant portions are the year and month. For example, you might use this value to set a reminder for a date 120 months in the future, or check whether 6 months have elapsed since a particular date.

Specify INTERVAL YEAR TO MONTH as follows:

```
INTERVAL YEAR [(year_precision)] TO MONTH
```

In the syntax:

year_precision is the number of digits in the YEAR datetime field. The default value of year_precision is 2.

Example

```
CREATE TABLE time_example2
(loan_duration INTERVAL YEAR (3) TO MONTH);

INSERT INTO time_example2 (loan_duration)
VALUES (INTERVAL '120' MONTH(3));

SELECT TO_CHAR( sysdate+loan_duration, 'dd-mon-yyyy')
FROM    time_example2;           --today's date is 26-Sep-2001
```

TO_CHAR(SYS

26-sep-2011

INTERVAL DAY TO SECOND Data Type

- **INTERVAL DAY TO SECOND stores a period of time in terms of days, hours, minutes, and seconds.**

```
INTERVAL DAY [(day_precision)]  
TO SECOND [(fractional_seconds_precision)]
```

```
INTERVAL '4 5:12:10.222' DAY TO SECOND(3)  
Indicates 4 days, 5 hours, 12 minutes, 10 seconds,  
and 222 thousandths of a second. INTERVAL '123' YEAR(3).
```

```
INTERVAL '7' DAY  
Indicates 7 days.
```

```
INTERVAL '180' DAY(3)  
Indicates 180 days.
```

ORACLE

9-18

Copyright © Oracle Corporation, 2001. All rights reserved.

INTERVAL DAY TO SECOND Data Type

INTERVAL DAY TO SECOND stores a period of time in terms of days, hours, minutes, and seconds. Use INTERVAL DAY TO SECOND to represent the precise difference between two datetime values. For example, you might use this value to set a reminder for a time 36 hours in the future, or to record the time between the start and end of a race. To represent long spans of time, including multiple years, with high precision, you can use a large value for the days portion.

Specify INTERVAL DAY TO SECOND as follows:

```
INTERVAL DAY [(day_precision)]  
TO SECOND [(fractional_seconds_precision)]
```

In the syntax:

<code>day_precision</code>	is the number of digits in the DAY datetime field. Accepted values are 0 to 9. The default is 2.
<code>fractional_seconds_precision</code>	is the number of digits in the fractional part of the SECOND datetime field. Accepted values are 0 to 9. The default is 6.

INTERVAL DAY TO SECOND Data Type

- **INTERVAL DAY TO SECOND** stores a period of time in terms of days, hours, minutes, and seconds.

```
INTERVAL '4 5:12:10.222' DAY TO SECOND(3)  
Indicates 4 days, 5 hours, 12 minutes, 10 seconds,  
and 222 thousandths of a second.
```

```
INTERVAL '4 5:12' DAY TO MINUTE  
Indicates 4 days, 5 hours and 12 minutes.
```

```
INTERVAL '400 5' DAY(3) TO HOUR  
Indicates 400 days 5 hours.
```

```
INTERVAL '11:12:10.2222222' HOUR TO SECOND(7)  
indicates 11 hours, 12 minutes, and 10.2222222 seconds.
```

ORACLE

9-19

Copyright © Oracle Corporation, 2001. All rights reserved.

INTERVAL DAY TO SECOND Data Type

Example

```
CREATE TABLE time_example3  
(day_duration INTERVAL DAY (3) TO SECOND);  
  
INSERT INTO time_example3 (day_duration)  
VALUES (INTERVAL '180' DAY(3));  
  
SELECT sysdate + day_duration "Half Year"  
FROM   time_example3;           --today's date is 26-Sep-2001
```

Half Year
25-MAR-02

Creating a Table by Using a Subquery Syntax

- Create a table and insert rows by combining the **CREATE TABLE** statement and the **AS subquery** option.

```
CREATE TABLE table  
      [(column, column...)]  
AS subquery;
```

- Match the number of specified columns to the number of subquery columns.
- Define columns with column names and default values.

ORACLE

9-20

Copyright © Oracle Corporation, 2001. All rights reserved.

Creating a Table from Rows in Another Table

A second method for creating a table is to apply the **AS subquery** clause, which both creates the table and inserts rows returned from the subquery.

In the syntax:

<i>table</i>	is the name of the table
<i>column</i>	is the name of the column, default value, and integrity constraint
<i>subquery</i>	is the SELECT statement that defines the set of rows to be inserted into the new table

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.
- The integrity rules are not passed onto the new table, only the column data type definitions.

Creating a Table by Using a Subquery

```
CREATE TABLE dept80
AS
  SELECT  employee_id, last_name,
          salary*12 ANNSAL,
          hire_date
  FROM    employees
  WHERE   department_id = 80;
```

Table created.

```
DESCRIBE dept80
```

Name	Null?	Type
EMPLOYEE_ID		NUMBER(6)
LAST_NAME	NOT NULL	VARCHAR2(25)
ANNSAL		NUMBER
HIRE_DATE	NOT NULL	DATE

ORACLE

9-21

Copyright © Oracle Corporation, 2001. All rights reserved.

Creating a Table from Rows in Another Table (continued)

The slide example 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 column definitions by using the *iSQL*Plus* DESCRIBE command.

Be sure to give a column alias when selecting an expression. The expression SALARY*12 is given the alias ANNSAL. Without the alias, this error is generated:

ERROR at line 3:

ORA-00998: must name this expression with a column alias

Instructor Note

To create a table with the same structure as an existing table, but without the data from the existing table, use a subquery with a WHERE clause that always evaluates as false. For example:

```
CREATE TABLE COPY_TABLE AS
( SELECT *
  FROM employees
  WHERE 1 = 2 );
```

The ALTER TABLE Statement

Use the ALTER TABLE statement to:

- Add a new column
- Modify an existing column
- Define a default value for the new column
- Drop a column

ORACLE

9-22

Copyright © Oracle Corporation, 2001. All rights reserved.

The ALTER TABLE Statement

After you create a table, you may need to change the table structure because: you omitted a column, your column definition needs to be changed, or you need to remove columns. You can do this by using the **ALTER TABLE statement**.

The ALTER TABLE Statement

Use the ALTER TABLE statement to add, modify, or drop columns.

```
ALTER TABLE table
ADD          (column datatype [DEFAULT expr]
             [, column datatype]...);
```

```
ALTER TABLE table
MODIFY       (column datatype [DEFAULT expr]
             [, column datatype]...);
```

```
ALTER TABLE table
DROP         (column);
```

ORACLE

9-23

Copyright © Oracle Corporation, 2001. All rights reserved.

The ALTER TABLE Statement (continued)

You can add, modify, and drop columns to a table by using the ALTER TABLE statement.

In the syntax:

<i>table</i>	is the name of the table
ADD MODIFY DROP	is the type of modification
<i>column</i>	is the name of the new column
<i>datatype</i>	is the data type and length of the new column
DEFAULT <i>expr</i>	specifies the default value for a new column

Note: The slide gives the abridged syntax for ALTER TABLE. More about ALTER TABLE is covered in a subsequent lesson.

Instructor Note

In Oracle8i and later, there are new options for the ALTER TABLE command, including the ability to drop a column from a table, which are covered later in this lesson.

Adding a Column

DEPT80

EMPLOYEE_ID	LAST_NAME	ANNSAL	HIRE_DATE
149	Zlotkey	126000	29-JAN-00
174	Abel	132000	11-MAY-96
176	Taylor	103200	24-MAR-98

New column

JOB_ID

“Add a new column to the DEPT80 table.”

DEPT80

EMPLOYEE_ID	LAST_NAME	ANNSAL	HIRE_DATE	JOB_ID
149	Zlotkey	126000	29-JAN-00	
174	Abel	132000	11-MAY-96	
176	Taylor	103200	24-MAR-98	

ORACLE

Adding a Column

The graphic adds the JOB_ID column to the DEPT80 table. Notice that the new column becomes the last column in the table.

Adding a Column

- You use the **ADD** clause to add columns.

```
ALTER TABLE dept80
ADD      (job_id VARCHAR2(9));
Table altered.
```

- The new column becomes the last column.

EMPLOYEE_ID	LAST_NAME	ANNSAL	HIRE_DATE	JOB_ID
149	Zlotkey	126000	29-JAN-00	
174	Abel	132000	11-MAY-96	
176	Taylor	103200	24-MAR-98	

ORACLE

9-25

Copyright © Oracle Corporation, 2001. All rights reserved.

Guidelines for Adding a Column

- You can add or **modify columns**.
- You cannot specify where the column is to appear. The new column becomes the last column.

The example on 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, then the new column is initially null for all the rows.

Modifying a Column

- You can change a column's data type, size, and default value.

```
ALTER TABLE dept80
MODIFY      (last_name VARCHAR2(30));
Table altered.
```

- A change to the default value affects only subsequent insertions to the table.

ORACLE

9-26

Copyright © Oracle Corporation, 2001. All rights reserved.

Modifying a Column

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 numeric or character columns.
- You can decrease the width of a column only if the column contains only null values or if the table has no rows.
- You can change the data type only if the column contains null values.
- 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 COLUMN** clause to drop columns you no longer need from the table.

```
ALTER TABLE dept80
DROP COLUMN job_id;
Table altered.
```

ORACLE

9-27

Copyright © Oracle Corporation, 2001. All rights reserved.

Dropping a Column

You can drop a column from a table by using the **ALTER TABLE** statement with the **DROP COLUMN clause**. This is a feature available in Oracle8i and later.

Guidelines

- The column may or may not contain data.
- Using the **ALTER TABLE** 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.
- Once a column is dropped, it cannot be recovered.

Instructor Note

When a column is dropped from a table, any other columns in that table that are marked with the **SET UNUSED** option are dropped too.

The SET UNUSED Option

- 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.

```
ALTER TABLE    table
SET    UNUSED   (column);
OR
ALTER TABLE    table
SET    UNUSED   COLUMN column;
```

```
ALTER TABLE table
DROP  UNUSED COLUMNS;
```

ORACLE

9-28

Copyright © Oracle Corporation, 2001. All rights reserved.

The SET UNUSED Option

The **SET UNUSED** option marks one or more columns as unused so that they can be dropped when the demand on system resources is lower. This is a feature available in Oracle8i and later. Specifying this clause does not actually remove the target columns from each row in the table (that is, it does not restore the disk space used by these columns). Therefore, the response time is faster than if you executed the **DROP** clause. Unused columns are treated as if they were dropped, even though their column data remains in the table's rows. After a column has been marked as unused, you have no access to that column. A **SELECT *** query will not retrieve data from unused columns. In addition, the names and types of columns marked unused will not be displayed during a **DESCRIBE**, and you can add to the table a new column with the same name as an unused column. **SET UNUSED** information is stored in the **USER_UNUSED_COL_TABS** dictionary view.

The DROP UNUSED COLUMNS Option

DROP UNUSED COLUMNS removes from the table all columns currently marked as unused. You can use this statement when you want to reclaim the extra disk space from 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);
Table altered.
```

```
ALTER TABLE dept80
DROP  UNUSED COLUMNS;
Table altered.
```

Dropping a Table

- All data and structure in the table is deleted.
- Any pending transactions are committed.
- All indexes are dropped.
- You *cannot* roll back the DROP TABLE statement.

```
DROP TABLE dept80;  
Table dropped.
```

ORACLE

9-29

Copyright © Oracle Corporation, 2001. All rights reserved.

Dropping a Table

The **DROP TABLE** statement removes the definition of an Oracle 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
```

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: The DROP TABLE statement, once executed, is irreversible. The Oracle server does not question the action when you issue the DROP TABLE statement. If you own that table or have a high-level privilege, then the table is immediately removed. As with all DDL statements, DROP TABLE is committed automatically.

Changing the Name of an Object

- To change the name of a table, view, sequence, or synonym, you execute the **RENAME** statement.

```
RENAME dept TO detail_dept;  
Table renamed.
```

- You must be the owner of the object.

ORACLE

9-30

Copyright © Oracle Corporation, 2001. All rights reserved.

Renaming a Table

Additional DDL statements include the **RENAME statement**, which is used to rename a table, view, sequence, or a synonym.

Syntax

```
RENAME      old_name    TO  new_name ;
```

In the syntax:

old_name is the old name of the table, view, sequence, or synonym.

new_name is the new name of the table, view, sequence, or synonym.

You must be the owner of the object that you rename.

Truncating a Table

- **The TRUNCATE TABLE statement:**
 - Removes all rows from a table
 - Releases the storage space used by that table

```
TRUNCATE TABLE detail_dept;  
Table truncated.
```

- You cannot roll back row removal when using TRUNCATE.
- Alternatively, you can remove rows by using the DELETE statement.

ORACLE

9-31

Copyright © Oracle Corporation, 2001. All rights reserved.

Truncating a Table

Another DDL statement is the **TRUNCATE TABLE statement**, which is used to remove all rows from a table and to release the storage space used by that table. When using the TRUNCATE TABLE statement, you cannot roll back row removal.

Syntax

```
TRUNCATE TABLE table;
```

In the syntax:

table is the name of the table

You must be the owner of the table or have **DELETE TABLE system privileges** to truncate a table.

The DELETE statement can also remove all rows from a table, but it does not release storage space. The TRUNCATE command is faster. Removing rows with the TRUNCATE statement is faster than removing them with the DELETE statement for the following reasons:

- The TRUNCATE statement is a data definition language (DDL) statement and generates no rollback information.
- Truncating a table does not fire the delete triggers of the table.
- If the table is the parent of a referential integrity constraint, you cannot truncate the table. Disable the constraint before issuing the TRUNCATE statement.

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 created.
```

- Comments can be viewed through the data dictionary views:
 - ALL_COL_COMMENTS
 - USER_COL_COMMENTS
 - ALL_TAB_COMMENTS
 - USER_TAB_COMMENTS

ORACLE

9-32

Copyright © Oracle Corporation, 2001. All rights reserved.

Adding a Comment to a Table

You can add a comment of up to 2,000 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:

<i>table</i>	is the name of the table
<i>column</i>	is the name of the column in a table
<i>text</i>	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 ' ';
```

Summary

In this lesson, you should have learned how to use DDL statements to create, alter, drop, and rename tables.

Statement	Description
CREATE TABLE	Creates a table
ALTER TABLE	Modifies table structures
DROP TABLE	Removes the rows and table structure
RENAME	Changes the name of a table, view, sequence, or synonym
TRUNCATE	Removes all rows from a table and releases the storage space
COMMENT	Adds comments to a table or view

ORACLE

9-33

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary

In this lesson, you should have learned how to use DDL commands to create, alter, drop, and rename tables. You also learned how to truncate a table and add comments to a table.

CREATE TABLE

- Create a table.
- Create a table based on another table by using a subquery.

ALTER TABLE

- Modify table structures.
- Change column widths, change column data types, and add columns.

DROP TABLE

- Remove rows and a table structure.
- Once executed, this statement cannot be rolled back.

RENAME

- Rename a table, view, sequence, or synonym.

TRUNCATE

- Remove all rows from a table and release the storage space used by the table.
- The DELETE statement removes only rows.

COMMENT

- Add a comment to a table or a column.
- Query the data dictionary to view the comment.

Practice 9 Overview

This practice covers the following topics:

- **Creating new tables**
- **Creating a new table by using the `CREATE TABLE AS` syntax**
- **Modifying column definitions**
- **Verifying that the tables exist**
- **Adding comments to tables**
- **Dropping tables**
- **Altering tables**

ORACLE

9-34

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 9 Overview

Create new tables by using the `CREATE TABLE` statement. Confirm that the new table was added to the database. Create the syntax in the command file, and then execute the command file to create the table.

Instructor Note

Explain what a table instance chart is. Tell students how to interpret a table instance chart. Explain that they need to look out for the entries in the Column Name, Data Type, and Length fields. The other entries are optional, and if these entries exist, they are constraints that need to be incorporated as a part of the table definition.

Point out to students that the practices are based on the tables that they are creating exclusively for this lesson. They need to be careful not to alter the other tables in the schema.

Practice 9

1. Create the DEPT table based on the following table instance chart. Place the syntax in a script called lab9_1.sql, then execute the statement in the script to create the table. Confirm that the table is created.

Column Name	ID	NAME
Key Type		
Nulls/Unique		
FK Table		
FK Column		
Data type	NUMBER	VARCHAR2
Length	7	25

Name	Null?	Type
ID		NUMBER(7)
NAME		VARCHAR2(25)

2. Populate the DEPT table with data from the DEPARTMENTS table. Include only columns that you need.
3. Create the EMP table based on the following table instance chart. Place the syntax in a script called lab9_3.sql, and then execute the statement in the script to create the table. Confirm that the table is created.

Column Name	ID	LAST_NAME	FIRST_NAME	DEPT_ID
Key Type				
Nulls/Unique				
FK Table				
FK Column				
Data type	NUMBER	VARCHAR2	VARCHAR2	NUMBER
Length	7	25	25	7

Name	Null?	Type
ID		NUMBER(7)
LAST_NAME		VARCHAR2(25)
FIRST_NAME		VARCHAR2(25)
DEPT_ID		NUMBER(7)

Practice 9 (continued)

4. Modify the EMP table to allow for longer employee last names. Confirm your modification.

Name	Null?	Type
ID		NUMBER(7)
LAST_NAME		VARCHAR2(50)
FIRST_NAME		VARCHAR2(25)
DEPT_ID		NUMBER(7)

5. Confirm that both the DEPT and EMP tables are stored in the data dictionary. (*Hint: USER_TABLES*)

TABLE_NAME
DEPT
EMP

6. Create the EMPLOYEES2 table based on the structure of the EMPLOYEES table. Include only the EMPLOYEE_ID, FIRST_NAME, LAST_NAME, SALARY, and DEPARTMENT_ID columns. Name the columns in your new table ID, FIRST_NAME, LAST_NAME, SALARY, and DEPT_ID, respectively.
7. Drop the EMP table.
8. Rename the EMPLOYEES2 table as EMP.
9. Add a comment to the DEPT and EMP table definitions describing the tables. Confirm your additions in the data dictionary.
10. Drop the FIRST_NAME column from the EMP table. Confirm your modification by checking the description of the table.
11. In the EMP table, mark the DEPT_ID column in the EMP table as UNUSED. Confirm your modification by checking the description of the table.
12. Drop all the UNUSED columns from the EMP table. Confirm your modification by checking the description of the table.

Instructor Note (for page 9-5)

There is an option, `CREATE GLOBAL TEMPORARY TABLE`, that identifies a table as temporary and visible to all sessions. The data in a temporary table is visible only to the session that inserts data into the table.

A temporary table has a definition that persists like the definitions of regular tables, but it contains either session-specific or transaction-specific data. You specify whether the data is session- or transaction-specific with the `ON COMMIT` keywords. Temporary tables use temporary segments. Unlike permanent tables, temporary tables and their indexes do not automatically allocate a segment when they are created. Instead, segments are allocated when the first `INSERT` (or `CREATE TABLE AS SELECT`) statement is performed. This means that if a `SELECT`, `UPDATE`, or `DELETE` statement is performed before the first `INSERT`, then the table appears to be empty. You can perform DDL commands (`ALTER TABLE`, `DROP TABLE`, `CREATE INDEX`, and so on) on a temporary table only when no session is currently bound to it. A session gets bound to a temporary table when an `INSERT` is performed on it. The session gets unbound by a `TRUNCATE`, at session termination, or by doing a `COMMIT` or `ABORT` for a transaction-specific temporary table. Temporary segments are deallocated at the end of the transaction for transaction-specific temporary tables and at the end of the session for session-specific temporary tables.

For more information on temporary tables and `CREATE TABLE`, refer to *Oracle9i Concepts*, “Temporary Tables.”

10

Including Constraints

ORACLE

Copyright © Oracle Corporation, 2001. All rights reserved.

Schedule:

Timing

45 minutes

25 minutes

70 minutes

Topic

Lecture

Practice

Total

Objectives

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

- **Describe constraints**
- **Create and maintain constraints**

ORACLE

10-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Lesson Aim

In this lesson, you learn how to implement business rules by including **integrity constraints**.

What are Constraints?

- **Constraints enforce rules at the table level.**
- **Constraints prevent the deletion of a table if there are dependencies.**
- **The following constraint types are valid:**
 - NOT NULL
 - UNIQUE
 - PRIMARY KEY
 - FOREIGN KEY
 - CHECK

ORACLE

10-3

Copyright © Oracle Corporation, 2001. All rights reserved.

Constraints

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 deletion of a table if there are dependencies from other tables
- Provide rules for Oracle tools, such as Oracle Developer

Data Integrity Constraints

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
FOREIGN KEY	Establishes and enforces a foreign key relationship between the column and a column of the referenced table
CHECK	Specifies a condition that must be true

For more information, see *Oracle9i SQL Reference*, “CONSTRAINT.”

Constraint Guidelines

- Name a constraint or the Oracle server generates a name by using the `SYS_Cn` format.
- Create a constraint either:
 - At the same time as the table is created, or
 - After the table has been created
- Define a constraint at the column or table level.
- View a constraint in the data dictionary.

ORACLE

10-4

Copyright © Oracle Corporation, 2001. All rights reserved.

Constraint Guidelines

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. 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 table has been created.

You can view the constraints defined for a specific table by looking at the `USER_CONSTRAINTS` data dictionary table.

Defining Constraints

```
CREATE TABLE [schema.]table
    (column datatype [DEFAULT expr]
    [column_constraint],
    ...
    [table_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

10-5

Copyright © Oracle Corporation, 2001. All rights reserved.

Defining Constraints

The slide gives the syntax for **defining constraints** while creating a table.

In the syntax:

<i>schema</i>	is the same as the owner's name
<i>table</i>	is the name of the table
DEFAULT <i>expr</i>	specifies a default value to use if a value is omitted in the INSERT statement
<i>column</i>	is the name of the column
<i>datatype</i>	is the column's data type and length
<i>column_constraint</i>	is an integrity constraint as part of the column definition
<i>table_constraint</i>	is an integrity constraint as part of the table definition

For more information, see *Oracle9i SQL Reference*, "CREATE TABLE."

Defining Constraints

- **Column constraint level**

```
column [CONSTRAINT constraint_name] constraint_type,
```

- **Table constraint level**

```
column, ...  
[CONSTRAINT constraint_name] constraint_type  
(column, ...),
```

ORACLE

10-6

Copyright © Oracle Corporation, 2001. All rights reserved.

Defining Constraints (continued)

Constraints are usually created at the same time as the table. Constraints can be added to a table after its creation and also temporarily disabled.

Constraints can be defined at one of two levels.

Constraint Level	Description
Column	References a single column and is defined within a specification for the owning column; can define any type of integrity constraint
Table	References one or more columns and is defined separately from the definitions of the columns in the table; can define any constraints except NOT NULL

In the syntax:

<i>constraint_name</i>	is the name of the constraint
<i>constraint_type</i>	is the type of the constraint

Instructor Note

Explain that the column level and the table level refer to location in the syntax.

The NOT NULL Constraint

Ensures that null values are not permitted for the column:

EMPLOYEE_ID	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	DEPARTMENT_ID
100	King	SKING	515.123.4567	17-JUN-87	AD_PRES	24000	90
101	Kochhar	NKOCHHAR	515.123.4568	21-SEP-89	AD_VP	17000	90
102	De Haan	LDEHAAN	515.123.4569	13-JAN-93	AD_VP	17000	90
103	Hunold	AHUNOLD	590.423.4567	03-JAN-90	IT_PROG	9000	60
104	Ernst	BERNST	590.423.4568	21-MAY-91	IT_PROG	6000	60
178	Grant	KGRANT	011.44.1644.429263	24-MAY-99	SA_REP	7000	
200	Whalen	JWHALEN	515.123.4444	17-SEP-87	AD_ASST	4400	10

20 rows selected.

↑
NOT NULL constraint
(No row can contain
a null value for
this column.)

↑
**NOT NULL
constraint**

↑
**Absence of NOT NULL
constraint**
(Any row can contain
null for this column.)

ORACLE

The NOT NULL Constraint

The **NOT NULL constraint** ensures that the column contains no null values. Columns without the NOT NULL constraint can contain null values by default.

The NOT NULL Constraint

Is defined at the column level:

```
CREATE TABLE employees(  
    employee_id    NUMBER(6),  
    last_name      VARCHAR2(25) NOT NULL,  
    salary         NUMBER(8,2),  
    commission_pct NUMBER(2,2),  
    hire_date      DATE  
                  CONSTRAINT emp_hire_date_nn  
                  NOT NULL,  
    ...
```

System
named

User
named

ORACLE

10-8

Copyright © Oracle Corporation, 2001. All rights reserved.

The NOT NULL Constraint (continued)

The NOT NULL constraint can be specified only at the **column level**, not at the **table level**.

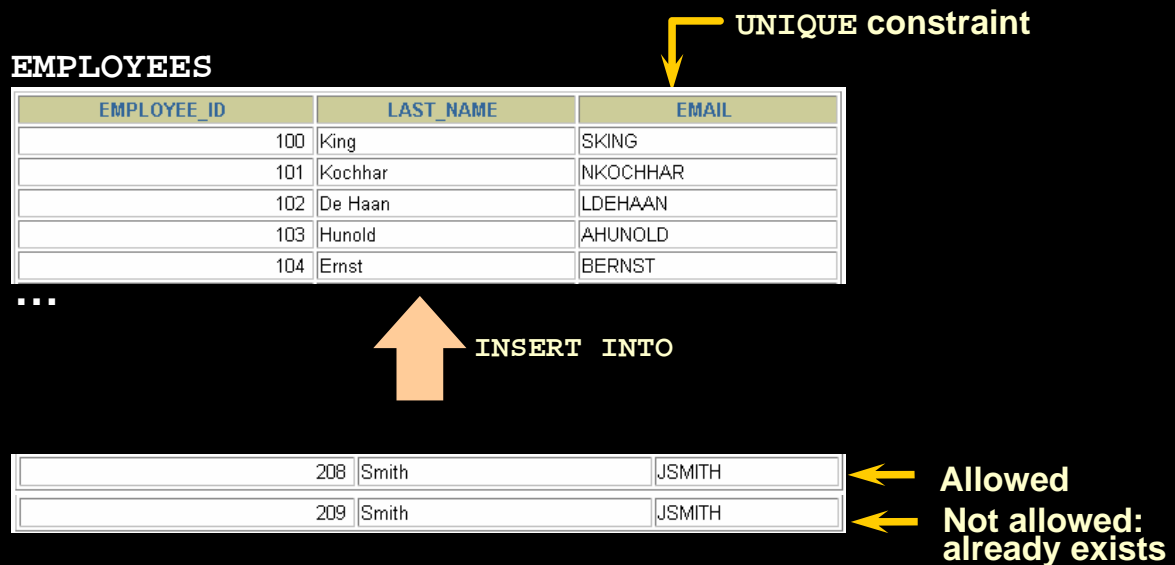
The slide example applies the NOT NULL constraint to the LAST_NAME and HIRE_DATE columns of the EMPLOYEES table. Because these constraints are unnamed, the Oracle server creates names for them.

You can specify the name of the constraint when you specify the constraint:

```
... last_name VARCHAR2(25)  
    CONSTRAINT emp_last_name_nn NOT NULL...
```

Note: The constraint examples described in this lesson may not be present in the sample tables provided with the course. If desired, these constraints can be added to the tables.

The UNIQUE Constraint



ORACLE

10-9

Copyright © Oracle Corporation, 2001. All rights reserved.

The UNIQUE Constraint

A **UNIQUE key integrity constraint** requires that every value in a column or set of columns (key) be unique—that is, no two rows of a table can have duplicate values in a specified column or 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 allow 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 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 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.

Instructor Note

Explain to students that since the JSMITH e-mail ID already exists after the first insertion, the second entry is not allowed.

The 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));
```

ORACLE

10-10

Copyright © Oracle Corporation, 2001. All rights reserved.

The UNIQUE Constraint (continued)

UNIQUE constraints can be defined at the column or table level. A **composite unique key** is created by using the table level definition.

The example on 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.

The PRIMARY KEY Constraint

DEPARTMENTS

PRIMARY KEY

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

...

Not allowed
(Null value)

INSERT INTO

	Public Accounting		1400
50	Finance	124	1500

Not allowed
(50 already exists)

ORACLE

10-11

Copyright © Oracle Corporation, 2001. All rights reserved.

The PRIMARY KEY Constraint

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 set of columns that uniquely identifies each row in a table. This constraint enforces uniqueness of the column or column combination and ensures that no column that is part of the primary key can contain a null value.

The PRIMARY KEY Constraint

Defined at either the table level or the column level:

```
CREATE TABLE departments(  
    department_id      NUMBER(4),  
    department_name     VARCHAR2(30)  
        CONSTRAINT dept_name_nn NOT NULL,  
    manager_id         NUMBER(6),  
    location_id        NUMBER(4),  
    CONSTRAINT dept_id_pk PRIMARY KEY(department_id));
```

ORACLE

10-12

Copyright © Oracle Corporation, 2001. All rights reserved.

The PRIMARY KEY Constraint (continued)

PRIMARY KEY constraints can be defined at the column level or table level. A composite PRIMARY KEY is created by using the table-level definition.

A table can have only one PRIMARY KEY constraint but can have several UNIQUE constraints.

The example on the slide defines a PRIMARY KEY constraint on the DEPARTMENT_ID column of the DEPARTMENTS table. The name of the constraint is DEPT_ID_PK.

Note: A UNIQUE index is automatically created for a PRIMARY KEY column.

Instructor Note

The example shown will not work in your schema because the DEPARTMENTS table already exists. To demonstrate this code, modify the name of the table within the script and then run the script.

The FOREIGN KEY Constraint

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

PRIMARY
KEY

EMPLOYEES

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
100	King	90
101	Kochhar	90
102	De Haan	90
103	Hunold	60
104	Ernst	60
107	Lorentz	60

FOREIGN
KEY

INSERT INTO

200	Ford	9
201	Ford	60

Not allowed
(9 does not
exist)

Allowed

ORACLE

10-13

Copyright © Oracle Corporation, 2001. All rights reserved.

The FOREIGN KEY Constraint

The **FOREIGN KEY**, or **referential integrity constraint**, designates a column or combination of columns as a foreign key and establishes a relationship between a primary key or a unique key in the same table or a different table. In the example on 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).

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, not physical, pointers.

Instructor Note

Explain to students that you cannot create a foreign key without existing primary key values.

The 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

10-14

Copyright © Oracle Corporation, 2001. All rights reserved.

The FOREIGN KEY Constraint (continued)

FOREIGN KEY constraints can be defined at the column or table constraint level. A composite foreign key must be created by using the table-level definition.

The example on 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_DEPTID_FK**.

The foreign key can also be defined at the column level, provided the constraint is based on a single column. The syntax differs in that the keywords **FOREIGN KEY** do not appear. For 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

ORACLE

10-15

Copyright © Oracle Corporation, 2001. All rights reserved.

The FOREIGN KEY Constraint (continued)

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 column in the parent table.
- **ON DELETE CASCADE** indicates that when the row in the parent table is deleted, the dependent rows in the child table will also be deleted.
- **ON DELETE SET NULL** converts foreign key values to null when the parent value is removed.

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, the row in the parent table cannot be deleted if it is referenced in the child table.

The CHECK Constraint

- Defines a condition that each row must satisfy
- The following expressions are not allowed:
 - References to CURRVAL, NEXTVAL, LEVEL, and ROWNUM pseudocolumns
 - Calls to SYSDATE, UID, USER, and USERENV functions
 - Queries that refer to other values in other rows

```
..., salary NUMBER(2)
      CONSTRAINT emp_salary_min
      CHECK (salary > 0),...
```

ORACLE

10-16

Copyright © Oracle Corporation, 2001. All rights reserved.

The CHECK Constraint

The **CHECK constraint** defines a condition that each row must satisfy. The condition can use the same constructs as query conditions, with the following exceptions:

- References to the CURRVAL, NEXTVAL, LEVEL, and ROWNUM pseudocolumns
- Calls to SYSDATE, UID, USER, and USERENV functions
- Queries that refer to other values in other rows

A single column can have multiple CHECK constraints which refer to the column in its definition. There is no limit to the number of CHECK constraints which you can define on a column.

CHECK constraints can be defined at the column level or table level.

```
CREATE TABLE employees
(
  ...
  salary NUMBER(8,2) CONSTRAINT emp_salary_min
                        CHECK (salary > 0),
  ...
)
```

Instructor Note

Explain what pseudocolumns are. Pseudocolumns are not actual columns in a table but they behave like columns. For example, you can select values from a pseudocolumn. However, you cannot insert into, update, or delete from a pseudocolumn. Pseudocolumns can be used in SQL statements.

Adding a Constraint Syntax

Use the **ALTER TABLE** statement to:

- Add or drop a constraint, but not modify its structure
- Enable or disable constraints
- Add a **NOT NULL** constraint by using the **MODIFY** clause

```
ALTER TABLE table  
ADD [CONSTRAINT constraint] type (column);
```

ORACLE

10-17

Copyright © Oracle Corporation, 2001. All rights reserved.

Adding a Constraint

You can add a constraint for existing tables by using the **ALTER TABLE** statement with the **ADD** clause.

In the syntax:

<i>table</i>	is the name of the table
<i>constraint</i>	is the name of the constraint
<i>type</i>	is the constraint type
<i>column</i>	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 will generate 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.

Instructor Note

You can defer checking constraints for validity until the end of the transaction.

A constraint is *deferred* if the system checks that it is satisfied only on commit. If a deferred constraint is violated, then committing causes the transaction to roll back.

A constraint is *immediate* if it is checked at the end of each statement. If it is violated, the statement is rolled back immediately.

Adding a Constraint

Add a FOREIGN KEY constraint to the EMPLOYEES table indicating that a manager must already exist as a valid employee in the EMPLOYEES table.

```
ALTER TABLE      employees
ADD CONSTRAINT    emp_manager_fk
    FOREIGN KEY(manager_id)
    REFERENCES employees(employee_id);
Table altered.
```

ORACLE

10-18

Copyright © Oracle Corporation, 2001. All rights reserved.

Adding a Constraint (continued)

The example on the slide creates a FOREIGN KEY constraint on the EMPLOYEES table. The constraint ensures that a manager exists as a valid employee in the EMPLOYEES table.

Instructor Note

To add a NOT NULL constraint, use the ALTER TABLE MODIFY syntax:

```
ALTER TABLE employees
MODIFY (salary CONSTRAINT emp_salary_nn NOT NULL);
```

Dropping a Constraint

- Remove the manager constraint from the **EMPLOYEES** table.

```
ALTER TABLE      employees
DROP CONSTRAINT    emp_manager_fk;
Table altered.
```

- Remove the **PRIMARY KEY** constraint on the **DEPARTMENTS** table and drop the associated **FOREIGN KEY** constraint on the **EMPLOYEES.DEPARTMENT_ID** column.

```
ALTER TABLE      departments
DROP PRIMARY KEY CASCADE;
Table altered.
```

ORACLE

10-19

Copyright © Oracle Corporation, 2001. All rights reserved.

Dropping a Constraint

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:

<i>table</i>	is the name of the table
<i>column</i>	is the name of the column affected by the constraint
<i>constraint</i>	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.

Disabling Constraints

- Execute the **DISABLE** clause of the **ALTER TABLE** statement to deactivate an integrity constraint.
- Apply the **CASCADE** option to disable dependent integrity constraints.

```
ALTER TABLE          employees
DISABLE CONSTRAINT    emp_emp_id_pk CASCADE;
Table altered.
```

ORACLE

10-20

Copyright © Oracle Corporation, 2001. All rights reserved.

Disabling a Constraint

You can disable a constraint without dropping it or re-creating it by using the **ALTER TABLE statement** with the **DISABLE clause**.

Syntax

```
ALTER    TABLE    table
DISABLE CONSTRAINT constraint [CASCADE];
```

In the syntax:

<i>table</i>	is the name of the table
<i>constraint</i>	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 currently disabled in the table definition by using the `ENABLE` clause.**

```
ALTER TABLE      employees
ENABLE CONSTRAINT  emp_emp_id_pk;
Table altered.
```

- **A `UNIQUE` or `PRIMARY KEY` index is automatically created if you enable a `UNIQUE` key or `PRIMARY KEY` constraint.**

ORACLE

10-21

Copyright © Oracle Corporation, 2001. All rights reserved.

Enabling a 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, that constraint applies to all the data in the table. All the data in the table must fit the constraint.
- If you enable a `UNIQUE` key or `PRIMARY KEY` constraint, a `UNIQUE` or `PRIMARY KEY` index is created automatically.
- You can use the `ENABLE` clause in both the `CREATE TABLE` statement and the `ALTER TABLE` statement.
- Enabling a primary key constraint that was disabled with the `CASCADE` option does not enable any foreign keys that are dependent upon the primary key.

Instructor Note

Please read the [Instructor Note on page 10-29](#) for information on the `VALIDATE` and `NOVALIDATE` options.

Cascading Constraints

- The **CASCADE CONSTRAINTS** clause is used along with the **DROP COLUMN** clause.
- The **CASCADE CONSTRAINTS** clause drops all referential integrity constraints that refer to the primary and unique keys defined on the dropped columns.
- The **CASCADE CONSTRAINTS** clause also drops all multicolumn constraints defined on the dropped columns.

ORACLE

10-22

Copyright © Oracle Corporation, 2001. All rights reserved.

Cascading Constraints

This statement illustrates the usage of the **CASCADE CONSTRAINTS** clause. Assume table TEST1 is created as follows:

```
CREATE TABLE test1 (  
    pk NUMBER PRIMARY KEY,  
    fk NUMBER,  
    col1 NUMBER,  
    col2 NUMBER,  
    CONSTRAINT fk_constraint FOREIGN KEY (fk) REFERENCES test1,  
    CONSTRAINT ck1 CHECK (pk > 0 and col1 > 0),  
    CONSTRAINT ck2 CHECK (col2 > 0));
```

An error is returned for the following statements:

```
ALTER TABLE test1 DROP (pk); -- pk is a parent key  
ALTER TABLE test1 DROP (col1); -- col1 is referenced by multicolumn constraint ck1
```

Cascading Constraints

Example:

```
ALTER TABLE test1  
DROP (pk) CASCADE CONSTRAINTS;  
Table altered.
```

```
ALTER TABLE test1  
DROP (pk, fk, coll) CASCADE CONSTRAINTS;  
Table altered.
```

ORACLE

10-23

Copyright © Oracle Corporation, 2001. All rights reserved.

Cascading Constraints (continued)

Submitting the following statement drops column PK, the primary key constraint, the `fk_constraint` foreign key constraint, and the check constraint, CK1:

```
ALTER TABLE test1 DROP (pk) CASCADE CONSTRAINTS;
```

If all columns referenced by the constraints defined on the dropped columns are also dropped, then `CASCADE CONSTRAINTS` is not required. For example, assuming that no other referential constraints from other tables refer to column PK, it is valid to submit the following statement without the `CASCADE CONSTRAINTS` clause:

```
ALTER TABLE test1 DROP (pk, fk, coll);
```

Instructor Note

Let the students know that if any constraint is referenced by columns from other tables or remaining columns in the target table, then you must specify `CASCADE CONSTRAINTS`. Otherwise, the statement aborts and the error `ORA-12991: column is referenced in a multicolumn constraint` is returned.

Viewing Constraints

Query the **USER_CONSTRAINTS** table to view all constraint definitions and names.

```
SELECT    constraint_name, constraint_type,  
          search_condition  
FROM      user_constraints  
WHERE     table_name = 'EMPLOYEES';
```

CONSTRAINT_NAME	C	SEARCH_CONDITION
EMP_LAST_NAME_NN	C	"LAST_NAME" IS NOT NULL
EMP_EMAIL_NN	C	"EMAIL" IS NOT NULL
EMP_HIRE_DATE_NN	C	"HIRE_DATE" IS NOT NULL
EMP_JOB_NN	C	"JOB_ID" IS NOT NULL
EMP_SALARY_MIN	C	salary > 0
EMP_EMAIL_UK	U	

...

ORACLE

10-24

Copyright © Oracle Corporation, 2001. All rights reserved.

Viewing Constraints

After creating a table, you can confirm its existence by issuing a **DESCRIBE command**. The only constraint that you can verify is the NOT NULL constraint. To view all constraints on your table, query the **USER_CONSTRAINTS** table.

The example on the slide displays the constraints on the EMPLOYEES table.

Note: Constraints that are not named by the table owner receive the system-assigned constraint name. In constraint type, C stands for CHECK, P for PRIMARY KEY, R for referential integrity, and U for UNIQUE key. Notice that the NOT NULL constraint is really a CHECK constraint.

Instructor Note

Point out to students that the NOT NULL constraint is stored in the data dictionary as a CHECK constraint. Draw their attention to the constraint type, for the NOT NULL constraints in the slide. The entry in the constraint_type field is C (as in CHECK) for these constraints.

Viewing the Columns Associated with Constraints

View the columns associated with the constraint names in the `USER_CONS_COLUMNS` view.

```
SELECT  constraint_name, column_name
FROM    user_cons_columns
WHERE   table_name = 'EMPLOYEES';
```

CONSTRAINT_NAME	COLUMN_NAME
EMP_DEPT_FK	DEPARTMENT_ID
EMP_EMAIL_NN	EMAIL
EMP_EMAIL_UK	EMAIL
EMP_EMP_ID_PK	EMPLOYEE_ID
EMP_HIRE_DATE_NN	HIRE_DATE
EMP_JOB_FK	JOB_ID
EMP_JOB_NN	JOB_ID

...

ORACLE

Viewing Constraints (continued)

You can view the names of the columns involved in constraints by querying the `USER_CONS_COLUMNS` data dictionary view. This view is especially useful for constraints that use system-assigned names.

Summary

In this lesson, you should have learned how to create constraints.

- **Types of constraints:**
 - NOT NULL
 - UNIQUE
 - PRIMARY KEY
 - FOREIGN KEY
 - CHECK
- **You can query the USER_CONSTRAINTS table to view all constraint definitions and names.**

ORACLE

10-26

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary

In this lesson, you should have learned how the Oracle server uses constraints to prevent invalid data entry into tables. You also learned how to implement the constraints in DDL statements.

The following constraint types are valid:

- NOT NULL
- UNIQUE
- PRIMARY KEY
- FOREIGN KEY
- CHECK

You can query the USER_CONSTRAINTS table to view all constraint definitions and names.

Practice 10 Overview

This practice covers the following topics:

- **Adding constraints to existing tables**
- **Adding more columns to a table**
- **Displaying information in data dictionary views**

ORACLE

10-27

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 10 Overview

In this practice, you will add constraints and more columns to a table using the statements covered in this lesson.

Note: It is recommended that you name the constraints that you define during the practices.

Practice 10

1. Add a table-level PRIMARY KEY constraint to the EMP table on the ID column. The constraint should be named at creation. Name the constraint my_emp_id_pk.

Hint: The constraint is enabled as soon as the ALTER TABLE command executes successfully.

2. Create a PRIMARY KEY constraint to the DEPT table using the ID column. The constraint should be named at creation. Name the constraint my_dept_id_pk.

Hint: The constraint is enabled as soon as the ALTER TABLE command executes successfully.

3. Add a column DEPT_ID to the EMP table. Add a foreign key reference on the EMP table that ensures that the employee is not assigned to a nonexistent department. Name the constraint my_emp_dept_id_fk.
4. Confirm that the constraints were added by querying the USER_CONSTRAINTS view. Note the types and names of the constraints. Save your statement text in a file called lab10_4.sql.

CONSTRAINT_NAME	C
MY_DEPT_ID_PK	P
SYS_C002541	C
MY_EMP_ID_PK	P
MY_EMP_DEPT_ID_FK	R

5. Display the object names and types from the USER_OBJECTS data dictionary view for the EMP and DEPT tables. Notice that the new tables and a new index were created.

If you have time, complete the following exercise:

6. Modify the EMP table. Add a COMMISSION column of NUMBER data type, precision 2, scale 2. Add a constraint to the commission column that ensures that a commission value is greater than zero.

Instructor Note (for pages 10-21)

You can also set constraints to `VALIDATE` or `NOVALIDATE`, in any combination with `ENABLE` or `DISABLE`, where:

- `VALIDATE` ensures that existing data conforms to the constraint.
- `NOVALIDATE` means that some existing data may not conform to the constraint.

In addition:

- `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 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.

Transitions between these states are governed by the following rules:

- `ENABLE` implies `VALIDATE`, unless `NOVALIDATE` is specified.
- `DISABLE` implies `NOVALIDATE`, unless `VALIDATE` is specified.
- `VALIDATE` and `NOVALIDATE` do not have any default implications for the `ENABLE` and `DISABLE` states.
- When a unique or primary key moves from the `DISABLE` state to the `ENABLE` state, and there is no existing index, a unique index is automatically created.
- Similarly, when a unique or primary key moves from `ENABLE` to `DISABLE` and it is enabled with a unique index, the unique index is dropped.
- When any constraint is moved from the `NOVALIDATE` state to the `VALIDATE` state, all data must be checked. (This can be very slow.) However, moving from `VALIDATE` to `NOVALIDATE` simply forgets that the data was ever checked.
- Moving a single constraint from the `ENABLE NOVALIDATE` state to the `ENABLE VALIDATE` state does not block reads, writes, or other DDL statements. It can be done in parallel.

The following statements enable novalidate disabled integrity constraints:

```
ALTER TABLE employees
  ENABLE NOVALIDATE CONSTRAINT EMP_EMAIL_UK;
ALTER TABLE employees
  ENABLE NOVALIDATE PRIMARY KEY
  ENABLE NOVALIDATE UNIQUE (employee_id, last_name);
```

The following statements enable or validate disabled integrity constraints:

```
ALTER TABLE employees
  MODIFY CONSTRAINT emp_email_uk VALIDATE;
ALTER TABLE employees
  MODIFY PRIMARY KEY ENABLE NOVALIDATE;
```


11

Creating Views

ORACLE

Copyright © Oracle Corporation, 2001. All rights reserved.

Schedule:

Timing

20 minutes

20 minutes

40 minutes

Topic

Lecture

Practice

Total

Objectives

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

- Describe a view
- Create, alter the definition of, and drop a view
- Retrieve data through a view
- Insert, update, and delete data through a view
- Create and use an inline view
- Perform “Top-N” analysis

ORACLE

11-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Lesson Aim

In this lesson, you learn how to create and use **views**. You also learn to query the relevant data dictionary object to retrieve information about views. Finally, you learn to create and use **inline views**, and perform **Top-N analysis** using inline views.

Database Objects

Object	Description
Table	Basic unit of storage; composed of rows and columns
View	Logically represents subsets of data from one or more tables
Sequence	Generates primary key values
Index	Improves the performance of some queries
Synonym	Alternative name for an object

ORACLE

What is a View?

EMPLOYEES Table:

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY
100	Steven	King	SKING	515.123.4567	17-JUN-87	AD_PRES	24000
101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-SEP-89	AD_VP	17000
102	Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-93	AD_VP	17000
103	Alexander	Hunold	AHUNOLD	590.423.4567	03-JAN-90	IT_PROG	9000
104	Bruce	Ernst	BERNST	590.423.4568	21-MAY-91	IT_PROG	6000
107	Diana	Lorentz	DLORENTZ	590.423.5567	07-FEB-99	IT_PROG	4200
124	Kevin	Mourgos	KMOURGOS	650.123.5234	16-NOV-99	ST_MAN	5800
141	Trenna	Rajs	TRAJS	650.121.8009	17-OCT-95	ST_CLERK	3500
142	Curtis	Davies	CDAVIES	650.121.2994	29-JAN-97	ST_CLERK	3100
143	Randall	Matos	RMATOS	650.121.2874	15-MAR-98	ST_CLERK	2600
149	Zlotkey				JUL-98	ST_CLERK	2500
174	Abel				JAN-00	SA_MAN	10500
176	Taylor				MAY-96	SA_REP	11000
176	Kimberely	Grant	KGRANT	515.144.1044, 423203	MAR-98	SA_REP	8600
176	Kimberely	Grant	KGRANT	515.144.1044, 423203	24-MAY-99	SA_REP	7000
200	Jennifer	Whalen	JWHALEN	515.123.4444	17-SEP-87	AD_ASST	4400
201	Michael	Hartstein	MHARTSTE	515.123.5555	17-FEB-96	MK_MAN	13000
202	Pat	Fay	PFAY	603.123.6666	17-AUG-97	MK_REP	6000
205	Shelley	Higgins	SHIGGINS	515.123.8080	07-JUN-94	AC_MGR	12000
206	William	Gietz	WGIEZT	515.123.8181	07-JUN-94	AC_ACCOUNT	8300

20 rows selected.

ORACLE

11-4

Copyright © Oracle Corporation, 2001. All rights reserved.

What Is a View?

You can present **logical subsets** or combinations of data by creating views of tables. A view is a logical table 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.

Instructor Note

Demo: 11_easyvu.sql

Purpose: The view shown on the slide is created as follows:

```
CREATE OR REPLACE VIEW simple_vu
AS SELECT employee_id, last_name, salary
FROM employees;
```

Why Use Views?

- **To restrict data access**
- **To make complex queries easy**
- **To provide data independence**
- **To present different views of the same data**

ORACLE

11-5

Copyright © Oracle Corporation, 2001. All rights reserved.

Advantages of Views

- Views restrict access to the data because the view can display selective 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 *Oracle9i SQL Reference*, “CREATE VIEW.”

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

11-6

Copyright © Oracle Corporation, 2001. All rights reserved.

Simple Views versus Complex Views

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
 - Can perform 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

Creating a View

- You embed a subquery within the **CREATE VIEW** statement.

```
CREATE [OR REPLACE] [FORCE|NOFORCE] VIEW view
  [(alias[, alias]...)]
  AS subquery
  [WITH CHECK OPTION [CONSTRAINT constraint]]
  [WITH READ ONLY [CONSTRAINT constraint]];
```

- The subquery can contain complex **SELECT** syntax.

ORACLE

11-7

Copyright © Oracle Corporation, 2001. All rights reserved.

Creating a View

You can create a view by **embedding a subquery within the CREATE VIEW** statement.

In the syntax:

<code>OR REPLACE</code>	re-creates the view if it already exists
<code>FORCE</code>	creates the view regardless of whether or not the base tables exist
<code>NOFORCE</code>	creates the view only if the base tables exist (This is the default.)
<code><i>view</i></code>	is the name of the view
<code><i>alias</i></code>	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.)
<code><i>subquery</i></code>	is a complete SELECT statement (You can use aliases for the columns in the SELECT list.)
<code>WITH CHECK OPTION</code>	specifies that only rows accessible to the view can be inserted or updated
<code><i>constraint</i></code>	is the name assigned to the CHECK OPTION constraint
<code>WITH READ ONLY</code>	ensures that no DML operations can be performed on this view

Creating a View

- Create a view, **EMPVU80**, that contains details of employees in department 80.

```
CREATE VIEW empvu80
AS SELECT employee_id, last_name, salary
FROM employees
WHERE department_id = 80;
```

View created.

- Describe the structure of the view by using the **iSQL*Plus DESCRIBE** command.

```
DESCRIBE empvu80
```

ORACLE

11-8

Copyright © Oracle Corporation, 2001. All rights reserved.

Creating a View (continued)

The example on 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 **iSQL*Plus DESCRIBE** command.

Name	Null?	Type
EMPLOYEE_ID	NOT NULL	NUMBER(6)
LAST_NAME	NOT NULL	VARCHAR2(25)
SALARY		NUMBER(8,2)

Guidelines for creating a view:

- The subquery that defines a view can contain complex **SELECT** syntax, including joins, groups, and subqueries.
- The subquery that defines the view cannot contain an **ORDER BY** clause. The **ORDER BY** clause is specified when you retrieve data from the view.
- If you do not specify a constraint name for a view created with the **WITH CHECK OPTION**, the system assigns a default name in the format **SYS_Cn**.
- You can use the **OR REPLACE** option to change the definition of the view without dropping and re-creating it or regranting 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 created.
```

- Select the columns from this view by the given alias names.

ORACLE

11-9

Copyright © Oracle Corporation, 2001. All rights reserved.

Creating a View (continued)

You can control the column names by including column aliases within the subquery.

The example on 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.

As an alternative, you can use an alias after the CREATE statement and prior to the SELECT subquery. The number of aliases listed must match the number of expressions selected in the subquery.

```
CREATE VIEW    salvu50 (ID_NUMBER, NAME, ANN_SALARY)
AS SELECT     employee_id, last_name, salary*12
FROM          employees
WHERE         department_id = 50;
View created.
```

Instructor Note

Let students know about materialized views or snapshots. The terms *snapshot* and *materialized view* are synonymous. Both refer to a table that contains the results of a query of one or more tables, each of which may be located on the same or on a remote database. The tables in the query are called master tables or detail tables. The databases containing the master tables are called the master databases. For more information regarding materialized views refer to: *Oracle9i SQL Reference*, "CREATE MATERIALIZED VIEW / SNAPSHOT."

Introduction to Oracle9i: SQL 11-9

Retrieving Data from a View

```
SELECT *  
FROM salvu50;
```

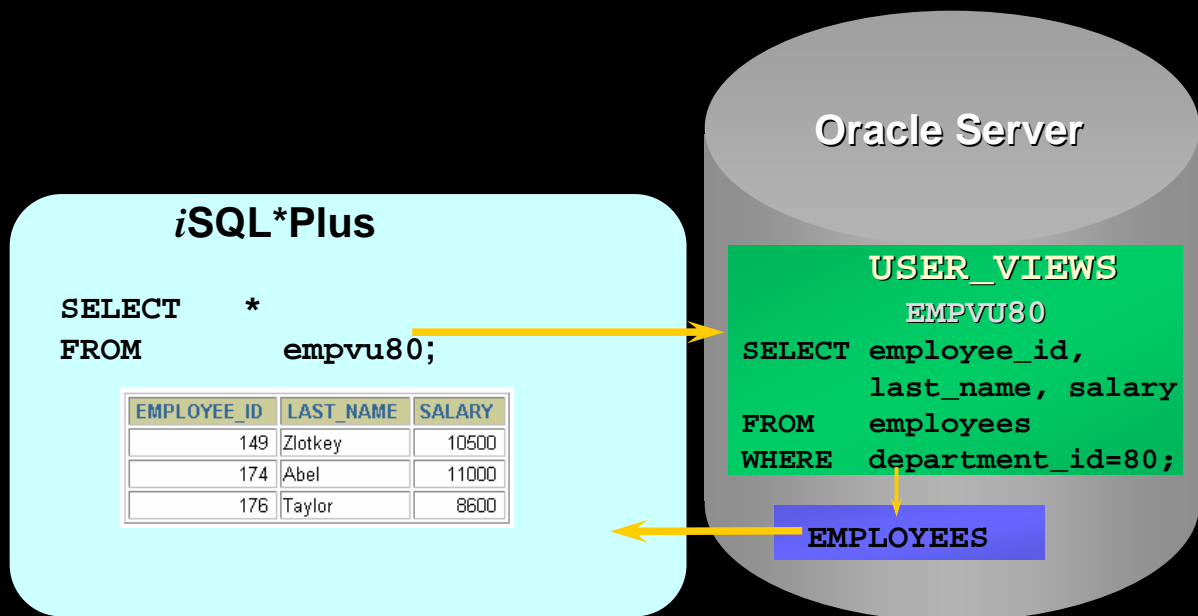
ID_NUMBER	NAME	ANN_SALARY
124	Mourgos	69600
141	Rajs	42000
142	Davies	37200
143	Matos	31200
144	Vargas	30000

ORACLE

Retrieving Data from a View

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.

Querying a View



ORACLE

11-11

Copyright © Oracle Corporation, 2001. All rights reserved.

Views in the Data Dictionary

Once your view has been 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.

Data Access Using Views

When you access data using a view, the Oracle server performs the following operations:

1. It retrieves the view definition from the data dictionary table **USER_VIEWS**.
2. It checks access privileges for the view base table.
3. It converts the view query into an equivalent operation on the underlying base table or tables. In other words, data is retrieved from, or an update is made to, the base tables.

Instructor Note

The view text is stored in a column of **LONG** data type. You may need to set **ARRAYSIZE** to a smaller value or increase the value of **LONG** to view the text.

Modifying a View

- **Modify the EMPVU80 view by using CREATE OR REPLACE VIEW clause. Add an alias for each column name.**

```
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 created.
```

- **Column aliases in the CREATE VIEW clause are listed in the same order as the columns in the subquery.**

ORACLE

11-12

Copyright © Oracle Corporation, 2001. All rights reserved.

Modifying a View

With the **OR 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 VIEW clause, remember that the aliases are listed in the same order as the columns in the subquery.

Instructor Note

The **OR REPLACE** option started with Oracle7. With earlier versions of Oracle, if the view needed to be changed, it had to be dropped and re-created.

Demo: 11_emp.sql

Purpose: To illustrate creating a view using aliases

Creating a Complex View

Create a complex view that contains group functions to display values from two tables.

```
CREATE 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, departments d
  WHERE         e.department_id = d.department_id
  GROUP BY     d.department_name;
```

View created.

ORACLE

11-13

Copyright © Oracle Corporation, 2001. All rights reserved.

Creating a Complex View

The example on the slide creates a complex view of department names, minimum salaries, maximum salaries, and 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 *iSQL*Plus* **DESCRIBE** command. Display the contents of the view by issuing a **SELECT** statement.

```
SELECT  *
FROM    dept_sum_vu;
```

NAME	MINSAL	MAXSAL	AVGSAL
Accounting	8300	12000	10150
Administration	4400	4400	4400
Executive	17000	24000	19333.3333
IT	4200	9000	6400
Marketing	6000	13000	9500
Sales	8600	11000	10033.3333
Shipping	2500	5800	3500

7 rows selected.

Rules for Performing DML Operations on a View

- You can 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

11-14

Copyright © Oracle Corporation, 2001. All rights reserved.

Performing DML Operations on a View

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

Instructor Note

For each row returned by a query, the ROWNUM pseudocolumn returns a number indicating the order in which Oracle server selects the row from a table or set of joined rows. The first row selected has a ROWNUM of 1, the second has 2, and so on.

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**
- **Columns defined by expressions**

ORACLE

11-15

Copyright © Oracle Corporation, 2001. All rights reserved.

Performing DML Operations on a View (continued)

You can modify data through a view unless it contains any of the conditions mentioned in the previous slide or columns defined by expressions—for example, `SALARY * 12`.

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 in the base tables that are not selected by the view

ORACLE

11-16

Copyright © Oracle Corporation, 2001. All rights reserved.

Performing DML Operations on a View (continued)

You can **add data through a view** unless it contains any of the items listed in the slide or there are NOT NULL columns without default values in the base table that are not selected by the view. All required values must be present in the view. Remember that you are adding values directly into the underlying table *through* the view.

For more information, see *Oracle9i SQL Reference*, “CREATE VIEW.”

Instructor Note

With Oracle7.3 and later, you can modify views that involve joins with some restrictions. The restrictions for DML operations described in the slide also apply to join views. Any UPDATE, INSERT, or DELETE statement on a join view can modify only one underlying base table. If at least one column in the subquery join has a unique index, then it may be possible to modify one base table in a join view. You can query USER_UPDATABLE_COLUMNS to see whether the columns in a join view can be updated.

Using the WITH CHECK OPTION Clause

- You can ensure that DML operations performed on the view stay within 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 created.

- Any attempt to change the department number for any row in the view fails because it violates the WITH CHECK OPTION constraint.

ORACLE

11-17

Copyright © Oracle Corporation, 2001. All rights reserved.

Using the WITH CHECK OPTION Clause

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 which the view cannot select, and therefore it allows 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, with the constraint name if that has been specified.

```
UPDATE empvu20
   SET   department_id = 10
  WHERE  employee_id = 201;
UPDATE empvu20
   *
```

ERROR at line 1:
ORA-01402: view WITH CHECK OPTION where-clause violation

Note: No rows are updated because if the department number were to change to 10, the view would no longer be able to see that employee. Therefore, with the WITH CHECK OPTION clause, the view can see only 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 on any row in the view results in an Oracle server error.

ORACLE

11-18

Copyright © Oracle Corporation, 2001. All rights reserved.

Denying DML Operations

You can ensure that no DML operations occur on your view by creating it with the **WITH READ ONLY** option. The example on the slide modifies the EMPVU10 view to prevent any DML operations on the view.

Instructor Note (for pages 11-17)

If the user does not supply a constraint name, the system assigns a name in the form SYS_Cn, where *n* is an integer that makes the constraint name unique within the system.

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 created.
```

ORACLE

11-19

Copyright © Oracle Corporation, 2001. All rights reserved.

Denying DML Operations

Any attempts to remove a row from a view with a **read-only constraint** results in an error.

```
DELETE FROM empvu10
WHERE  employee_number = 200;
DELETE FROM empvu10
      *
```

ERROR at line 1:

ORA-01752: cannot delete from view without exactly one key-
preserved table

Any attempt to insert a row or modify a row using the view with a read-only constraint results in Oracle server error:

01733: virtual column not allowed here.

Removing a View

You can remove a view without losing data because a view is based on underlying tables in the database.

```
DROP VIEW view;
```

```
DROP VIEW empvu80;  
View dropped.
```

ORACLE

11-20

Copyright © Oracle Corporation, 2001. All rights reserved.

Removing a View

You use the **DROP VIEW statement** to remove a view. The statement removes the view definition from the database. Dropping views has no effect on the tables on which the view was based. Views or other applications based on 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

Inline Views

- An inline view is a subquery with an alias (or correlation name) that you can use within a SQL statement.
- A named subquery in the FROM clause of the main query is an example of an inline view.
- An inline view is not a schema object.

ORACLE

11-21

Copyright © Oracle Corporation, 2001. All rights reserved.

Inline Views

An **inline view** is created by placing a **subquery in the FROM clause** and giving that subquery an alias. The subquery defines a data source that can be referenced in the main query. In the following example, the inline view **b** returns the details of all department numbers and the maximum salary for each department from the **EMPLOYEES** table. The **WHERE a.department_id = b.department_id AND a.salary < b.maxsal** clause of the main query displays employee names, salaries, department numbers, and maximum salaries for all the employees who earn less than the maximum salary in their department.

```
SELECT a.last_name, a.salary, a.department_id, b.maxsal
FROM   employees a, (SELECT department_id, max(salary) maxsal
                     FROM   employees
                     GROUP BY department_id) b
WHERE  a.department_id = b.department_id
AND    a.salary < b.maxsal;
```

LAST_NAME	SALARY	DEPARTMENT_ID	MAXSAL
Fay	6000	20	13000
Rajs	3500	50	5800
Davies	3100	50	5800
Matos	2600	50	5800
Vargas	2500	50	5800

■ ■ ■

12 rows selected.

Top-N Analysis

- **Top-N queries ask for the n largest or smallest values of a column. For example:**
 - What are the ten best selling products?
 - What are the ten worst selling products?
- **Both largest values and smallest values sets are considered Top-N queries.**

ORACLE

11-22

Copyright © Oracle Corporation, 2001. All rights reserved.

“Top-N” Analysis

Top-N queries are useful in scenarios where the need is to display only the n top-most or the n bottom-most records from a table based on a condition. This result set can be used for further analysis. For example, using Top-N analysis you can perform the following types of queries:

- The top three earners in the company
- The four most recent recruits in the company
- The top two sales representatives who have sold the maximum number of products
- The top three products that have had the maximum sales in the last six months

Instructor Note

The capability to include the `ORDER BY` clause in a subquery makes Top-N analysis possible.

Performing Top-N Analysis

The high-level structure of a Top-N analysis query is:

```
SELECT [column_list], ROWNUM
FROM   (SELECT [column_list]
        FROM table
        ORDER BY Top-N_column)
WHERE  ROWNUM <= N;
```

ORACLE

11-23

Copyright © Oracle Corporation, 2001. All rights reserved.

Performing “Top-N” Analysis

Top-N queries use a consistent nested query structure with the elements described below:

- A subquery or an inline view to generate the sorted list of data. The subquery or the inline view includes the `ORDER BY` clause to ensure that the ranking is in the desired order. For results retrieving the largest values, a `DESC` parameter is needed.
- An outer query to limit the number of rows in the final result set. The outer query includes the following components:
 - The **ROWNUM pseudocolumn**, which assigns a sequential value starting with 1 to each of the rows returned from the subquery.
 - A `WHERE` clause, which specifies the *n* rows to be returned. The outer `WHERE` clause must use a `<` or `<=` operator.

Example of Top-N Analysis

To display the top three earner names and salaries from the EMPLOYEES table:

1 2 3

```
SELECT ROWNUM as RANK, last_name, salary
FROM (SELECT last_name,salary FROM employees
      ORDER BY salary DESC)
WHERE ROWNUM <= 3;
```

RANK	LAST_NAME	SALARY
1	King	24000
2	Kochhar	17000
3	De Haan	17000

1

2

3

ORACLE

11-24

Copyright © Oracle Corporation, 2001. All rights reserved.

Example of “Top-N” Analysis

The example on the slide illustrates how to display the names and salaries of the top three earners from the EMPLOYEES table. The subquery returns the details of all employee names and salaries from the EMPLOYEES table, sorted in the descending order of the salaries. The WHERE ROWNUM < 3 clause of the main query ensures that only the first three records from this result set are displayed.

Here is another **example of Top-N analysis** that uses an inline view. The example below uses the inline view E to display the four most senior employees in the company.

```
SELECT ROWNUM as SENIOR,E.last_name, E.hire_date
FROM (SELECT last_name,hire_date FROM employees
      ORDER BY hire_date)E
WHERE rownum <= 4;
```

SENIOR	LAST_NAME	HIRE_DATE
1	King	17-JUN-87
2	Whalen	17-SEP-87
3	Kochhar	21-SEP-89
4	Hunold	03-JAN-90

Summary

In this lesson, you should have learned that a view is derived from data in other tables or views and provides the following advantages:

- **Restricts database access**
- **Simplifies queries**
- **Provides data independence**
- **Provides multiple views of the same data**
- **Can be dropped without removing the underlying data**
- **An inline view is a subquery with an alias name.**
- **Top-N analysis can be done using subqueries and outer queries.**

ORACLE

11-25

Copyright © Oracle Corporation, 2001. All rights reserved.

What Is a View?

A view is based on a table or another view and acts as a window through which data on tables can be viewed or changed. A view does not contain data. The definition of the view is stored in the data dictionary. You can see the definition of the view in the `USER_VIEWS` data dictionary table.

Advantages of Views

- Restrict database access
- Simplify queries
- Provide data independence
- Provide multiple views of the same data
- Can be removed without affecting the underlying data

View Options

- Can be a simple view, based on one table
- Can be a complex view based on more than one table or can contain groups of functions
- Can replace other views with the same name
- Can contain a check constraint
- Can be read-only

Practice 11 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**
- **Displaying view definitions**
- **Removing views**

ORACLE

11-26

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 11 Overview

In this practice, you create simple and complex views and attempt to perform DML statements on the views.

Practice 11

1. Create a view called EMPLOYEES_VU based on the employee numbers, employee names, and department numbers from the EMPLOYEES table. Change the heading for the employee name to EMPLOYEE.
2. Display the contents of the EMPLOYEES_VU view.

EMPLOYEE_ID	EMPLOYEE	DEPARTMENT_ID
100	King	90
101	Kochhar	90
102	De Haan	90
103	Hunold	60
104	Ernst	60
107	Lorentz	60
...		
206	Gietz	110

20 rows selected.

3. Select the view name and text from the USER_VIEWS data dictionary view.

Note: Another view already exists. The EMP_DETAILS_VIEW was created as part of your schema.

Note: To see more contents of a LONG column, use the *iSQL*Plus* command SET LONG *n*, where *n* is the value of the number of characters of the LONG column that you want to see.

VIEW_NAME	TEXT
EMPLOYEES_VU	SELECT employee_id, last_name employee, department_id FROM employees
EMP_DETAILS_VIEW	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_name FROM employees e, departments d, jobs j, locations l, countries c, regions r WHERE 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

4. Using your EMPLOYEES_VU view, enter a query to display all employee names and department numbers.

EMPLOYEE	DEPARTMENT_ID
King	90
Kochhar	90
...	
Gietz	110

20 rows selected.

Practice 11 (continued)

5. Create a view named DEPT50 that contains the employee numbers, employee last names, and department numbers for all employees in department 50. Label the view columns EMPNO, EMPLOYEE, and DEPTNO. Do not allow an employee to be reassigned to another department through the view.
6. Display the structure and contents of the DEPT50 view.

Name	Null?	Type
EMPNO	NOT NULL	NUMBER(6)
EMPLOYEE	NOT NULL	VARCHAR2(25)
DEPTNO		NUMBER(4)

EMPNO	EMPLOYEE	DEPTNO
124	Mourgos	50
141	Rajs	50
142	Davies	50
143	Matos	50
144	Vargas	50

7. Attempt to reassign Matos to department 80.

If you have time, complete the following exercise:

8. Create a view called SALARY_VU based on the employee last names, department names, salaries, and salary grades for all employees. Use the EMPLOYEES, DEPARTMENTS, and JOB_GRADES tables. Label the columns Employee, Department, Salary, and Grade, respectively.

12

Other Database Objects

ORACLE®

Copyright © Oracle Corporation, 2001. All rights reserved.

Schedule:	Timing	Topic
	20 minutes	Lecture
	20 minutes	Practice
	40 minutes	Total

Objectives

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

- **Create, maintain, and use sequences**
- **Create and maintain indexes**
- **Create private and public synonyms**

ORACLE

12-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Lesson Aim

In this lesson, you learn how to create and maintain some of the other commonly used database objects. These objects include **sequences**, **indexes**, and **synonyms**.

Database Objects

Object	Description
Table	Basic unit of storage; composed of rows and columns
View	Logically represents subsets of data from one or more tables
Sequence	Generates primary key values
Index	Improves the performance of some queries
Synonym	Alternative name for an object

ORACLE

12-3

Copyright © Oracle Corporation, 2001. All rights reserved.

Database Objects

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 some 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**.

What Is a Sequence?

A sequence:

- Automatically generates unique numbers
- Is a sharable object
- Is typically used to create a primary key value
- Replaces application code
- Speeds up the efficiency of accessing sequence values when cached in memory

ORACLE

12-4

Copyright © Oracle Corporation, 2001. All rights reserved.

What Is a Sequence?

A **sequence** is a user created database object that can be shared by multiple users to generate unique integers.

A typical usage for sequences is to create a primary key value, which must be unique for each row. The 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 independently of tables. Therefore, the same sequence can be used for multiple tables.

The CREATE SEQUENCE Statement Syntax

Define a sequence to generate sequential numbers automatically:

```
CREATE SEQUENCE sequence
  [INCREMENT BY n]
  [START WITH n]
  [{MAXVALUE n | NOMAXVALUE}]
  [{MINVALUE n | NOMINVALUE}]
  [{CYCLE | NOCYCLE}]
  [{CACHE n | NOCACHE}] ;
```

ORACLE

12-5

Copyright © Oracle Corporation, 2001. All rights reserved.

Creating a Sequence

Automatically generate sequential numbers by using the **CREATE SEQUENCE statement**.

In the syntax:

<i>sequence</i>	is the name of the sequence generator
INCREMENT BY <i>n</i>	specifies the interval between sequence numbers where <i>n</i> is an integer (If this clause is omitted, the sequence increments by 1.)
START WITH <i>n</i>	specifies the first sequence number to be generated (If this clause is omitted, the sequence starts with 1.)
MAXVALUE <i>n</i>	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 <i>n</i>	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.)
CYCLE NOCYCLE	specifies whether the sequence continues to generate values after reaching its maximum or minimum value (NOCYCLE is the default option.)
CACHE <i>n</i> NOCACHE	specifies how many values the Oracle server preallocates and keep in memory (By default, the Oracle server caches 20 values.)

Creating a Sequence

- 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  
    INCREMENT BY 10  
    START WITH 120  
    MAXVALUE 9999  
    NOCACHE  
    NOCYCLE;
```

Sequence created.

ORACLE

12-6

Copyright © Oracle Corporation, 2001. All rights reserved.

Creating a Sequence (continued)

The example on 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 120, does not allow caching, and does not cycle.

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 *Oracle9i SQL Reference*, “CREATE SEQUENCE.”

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.

Instructor Note

If the **INCREMENT BY** value is negative, the sequence descends. Also, **ORDER | NOORDER** options are available. The **ORDER** option guarantees that sequence values are generated in order. It is not important if you use the sequence to generate primary key values. This option is relevant only with the **Parallel Server** option.

If sequence values are cached, they will be lost if there is a system failure.

Confirming Sequences

- **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;
```

- **The `LAST_NUMBER` column displays the next available sequence number if `NOCACHE` is specified.**

ORACLE

12-7

Copyright © Oracle Corporation, 2001. All rights reserved.

Confirming Sequences

Once you have created your sequence, it is documented in the data dictionary. Since 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.

SEQUENCE_NAME	MIN_VALUE	MAX_VALUE	INCREMENT_BY	LAST_NUMBER
DEPARTMENTS_SEQ	1	9990	10	280
DEPT_DEPTID_SEQ	1	9999	10	120
EMPLOYEES_SEQ	1	1.0000E+27	1	207
LOCATIONS_SEQ	1	9900	100	3300

Instructor Note

Demo: [12_dd.sql](#)

Purpose: To illustrate the `USER_SEQUENCES` data dictionary view and its contents.

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.

ORACLE

12-8

Copyright © Oracle Corporation, 2001. All rights reserved.

Using a Sequence

After you create your sequence, it generates sequential numbers for use in your tables. Reference the sequence values by using the **NEXTVAL and CURRVAL pseudocolumns**.

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. 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 *sequence*.CURRVAL is referenced, 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 GROUP BY, HAVING, or ORDER BY clauses
- A subquery in a SELECT, DELETE, or UPDATE statement
- The DEFAULT expression in a CREATE TABLE or ALTER TABLE statement

For more information, see *Oracle9i SQL Reference*, “Pseudocolumns” section and “CREATE SEQUENCE.”

Instructor Note

Be sure to point out the rules listed on [this page](#).

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 row created.
```

- View the current value for the DEPT_DEPTID_SEQ sequence.

```
SELECT  dept_deptid_seq.CURRVAL  
FROM    dual;
```

ORACLE

12-10

Copyright © Oracle Corporation, 2001. All rights reserved.

Using a Sequence

The example on the slide inserts a new department in the DEPARTMENTS table. It **uses the DEPT_DEPTID_SEQ sequence** for generating a new department number as follows:

You can view the current value of the sequence:

```
SELECT dept_deptid_seq.CURRVAL  
FROM    dual;
```

CURRVAL	
	120

Suppose now you 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 for generating new employee numbers.

Using a Sequence

- **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
- **If the sequence was created with NOCACHE, view the next available value, by querying the USER_SEQUENCES table.**

ORACLE

12-11

Copyright © Oracle Corporation, 2001. All rights reserved.

Caching Sequence Values

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 independent 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 the memory, then 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. If you do so, each table can contain gaps in the sequential numbers.

Viewing the Next Available Sequence Value without Incrementing It

If the sequence was created with NOCACHE, it is possible to **view the next available sequence value** without incrementing it by querying the USER_SEQUENCES table.

Instructor Note

Frequently used sequences should be created with caching to improve efficiency. For cached sequences, there is no way to find out what the next available sequence value will be without actually obtaining, and using up, that value. It is recommended that users resist finding the next sequence value. Trust the system to provide a unique value each time a sequence is used in an INSERT statement.

Modifying a Sequence

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 altered.

ORACLE

12-12

Copyright © Oracle Corporation, 2001. All rights reserved.

Altering a Sequence

If you reach the MAXVALUE limit for your sequence, no additional values from the sequence are allocated and you will receive an error indicating that the sequence exceeds the 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 *Oracle9i SQL Reference*, “ALTER SEQUENCE.”

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.

ORACLE

12-13

Copyright © Oracle Corporation, 2001. All rights reserved.

Guidelines for Modifying Sequences

- You must be the owner or have the ALTER privilege for the sequence in order to modify it.
- Only future sequence numbers are affected by the ALTER SEQUENCE statement.
- The START WITH option cannot be changed using ALTER SEQUENCE. The sequence must be dropped and re-created in order 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;
```

```
ALTER SEQUENCE dept_deptid_seq
```

```
*
```

```
ERROR at line 1:
```

```
ORA-04009: MAXVALUE cannot be made to be less than the current  
value
```

Removing a Sequence

- Remove a sequence from the data dictionary by using the **DROP SEQUENCE** statement.
- Once removed, the sequence can no longer be referenced.

```
DROP SEQUENCE dept_deptid_seq;  
Sequence dropped.
```

ORACLE

12-14

Copyright © Oracle Corporation, 2001. All rights reserved.

Removing a Sequence

To remove a sequence from the data dictionary, use the **DROP SEQUENCE statement**. You must be the owner of the sequence or have the **DROP ANY SEQUENCE** privilege to remove it.

Syntax

```
DROP SEQUENCE sequence;
```

In the syntax:

sequence is the name of the sequence generator

For more information, see *Oracle9i SQL Reference*, “**DROP SEQUENCE**.”

What is an Index?

An index:

- Is a schema object
- Is used by the Oracle server to speed up the retrieval of rows by using a pointer
- Can reduce disk I/O by using a rapid path access method to locate data quickly
- Is independent of the table it indexes
- Is used and maintained automatically by the Oracle server

ORACLE

12-15

Copyright © Oracle Corporation, 2001. All rights reserved.

Indexes

An Oracle server **index** is a schema object that can speed up the retrieval of rows by using a pointer. Indexes can be created explicitly or automatically. If you do not have an index on the column, then a full table scan occurs.

An index provides direct and fast access to rows in a table. Its purpose is to reduce the necessity of disk I/O by using an indexed path to locate data quickly. The index is used and maintained automatically by the Oracle server. Once an index is created, no direct activity is required by the user.

Indexes are logically and physically independent of the table they index. 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, corresponding indexes are also dropped.

For more information, see *Oracle9i Concepts*, “Schema Objects” section, “Indexes” topic.

Instructor Note

The decision to create indexes is a global, high-level decision. Creation and maintenance of indexes is often a task for the database administrator.

Reference the column that has an index in the predicate WHERE clause without modifying the indexed column with a function or expression.

A ROWID is a hexadecimal string representation of the row address containing block identifier, row location in the block, and the database file identifier. The fastest way to access any particular row is by referencing its ROWID.

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:** Users can create nonunique indexes on columns to speed up access to the rows.

ORACLE

12-16

Copyright © Oracle Corporation, 2001. All rights reserved.

Types of Indexes

Two types of indexes can be created. One type is a **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` key constraint. The name of the index is the name given to the constraint.

The other type of index is a **nonunique index**, which a user can create. For example, you can create a `FOREIGN KEY` column index for a join in a query to improve retrieval speed.

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 INDEX index
ON table (column[, column]...);
```

- 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 created.
```

ORACLE

12-17

Copyright © Oracle Corporation, 2001. All rights reserved.

Creating an Index

Create an index on one or more columns by issuing the **CREATE INDEX** statement.

In the syntax:

<i>index</i>	is the name of the index
<i>table</i>	is the name of the table
<i>column</i>	is the name of the column in the table to be indexed

For more information, see *Oracle9i SQL Reference*, “CREATE INDEX.”

Instructor Note

To create an index in your schema, you must have the **CREATE TABLE** privilege. To create an index in any schema, you need the **CREATE ANY INDEX** privilege or the **CREATE TABLE** privilege on the table on which you are creating the index.

Another option in the syntax is the **UNIQUE** keyword. Emphasize that you should not explicitly define unique indexes on tables. Instead define uniqueness in the table as a constraint. The Oracle server enforces unique integrity constraints by automatically defining a unique index on the unique key.

When to Create an Index

You should create an index if:

- A column contains a wide range of values
- A column contains a large number of null values
- One or more columns are frequently used together in a `WHERE` clause or a join condition
- The table is large and most queries are expected to retrieve less than 2 to 4 percent of the rows

ORACLE

12-18

Copyright © Oracle Corporation, 2001. All rights reserved.

More Is Not Always Better

More indexes on a table does not mean faster queries. Each DML operation that is committed on a table with indexes means that the indexes must be updated. The more indexes you have associated with a table, the more effort the Oracle server must make to update all the indexes after a DML operation.

When to Create an Index

Therefore, you should create indexes only if:

- The column contains a wide range of values
- The column contains a large number of null values
- One or more columns are frequently used together in a `WHERE` clause or join condition
- The table is large and most queries are expected to retrieve less than 2–4% of the rows

Remember that if you want to enforce uniqueness, you should define a unique constraint in the table definition. Then a unique index is created automatically.

Instructor Note

A composite index (also called a concatenated index) is an index that you create on multiple columns in a table. Columns in a composite index can appear in any order and need not be adjacent in the table.

Composite indexes can speed retrieval of data for `SELECT` statements in which the `WHERE` clause references all or the leading portion of the columns in the composite index.

Introduction to Oracle9i: SQL 12-18

When Not to Create an Index

It is usually not worth creating an index if:

- The table is small
- The columns are not often used as a condition in the query
- Most queries are expected to retrieve more than 2 to 4 percent of the rows in the table
- The table is updated frequently
- The indexed columns are referenced as part of an expression

ORACLE

Instructor Note

Null values are not included in the index.

To optimize joins, you can create an index on the FOREIGN KEY column, which speeds up the search to match rows to the PRIMARY KEY column.

The optimizer does not use an index if the WHERE clause contains the IS NULL expression.

Confirming Indexes

- The **USER_INDEXES** data dictionary view contains the name of the index and its uniqueness.
- The **USER_IND_COLUMNS** view contains the index name, the table name, and the column name.

```
SELECT    ic.index_name, ic.column_name,  
          ic.column_position col_pos, ix.uniqueness  
FROM      user_indexes ix, user_ind_columns ic  
WHERE     ic.index_name = ix.index_name  
AND       ic.table_name = 'EMPLOYEES';
```

ORACLE

12-20

Copyright © Oracle Corporation, 2001. All rights reserved.

Confirming Indexes

Confirm the existence of indexes from the **USER_INDEXES** data dictionary view. You can also check the columns involved in an index by querying the **USER_IND_COLUMNS** view.

The example on the slide displays all the previously created indexes, with the names of the affected column, and the index's uniqueness, on the **EMPLOYEES** table.

INDEX_NAME	COLUMN_NAME	COL_POS	UNIQUENES
EMP_EMAIL_UK	EMAIL	1	UNIQUE
EMP_EMP_ID_PK	EMPLOYEE_ID	1	UNIQUE
EMP_DEPARTMENT_IX	DEPARTMENT_ID	1	NONUNIQUE
EMP_JOB_IX	JOB_ID	1	NONUNIQUE
EMP_MANAGER_IX	MANAGER_ID	1	NONUNIQUE
EMP_NAME_IX	LAST_NAME	1	NONUNIQUE
EMP_NAME_IX	FIRST_NAME	2	NONUNIQUE
EMP_LAST_NAME_IDX	LAST_NAME	1	NONUNIQUE

8 rows selected.

Function-Based Indexes

- A function-based index is an index 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 departments(UPPER(department_name));
```

Index created.

```
SELECT *  
FROM departments  
WHERE UPPER(department_name) = 'SALES';
```

ORACLE

12-21

Copyright © Oracle Corporation, 2001. All rights reserved.

Function-Based Index

Function-based indexes defined with the `UPPER(column_name)` or `LOWER(column_name)` keywords allow case-insensitive searches. For example, the following index:

```
CREATE INDEX upper_last_name_idx ON employees (UPPER(last_name));
```

Facilitates processing queries such as:

```
SELECT * FROM employees WHERE UPPER(last_name) = 'KING';
```

To ensure that the Oracle server uses the index rather than performing a full table scan, be sure that the value of the function is not null in subsequent queries. For example, the following statement is guaranteed to 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);
```

Function-Based Index (continued)

The Oracle server treats indexes with columns marked `DESC` as function-based indexes. The columns marked `DESC` are sorted in descending order.

Instructor Note

Let students know that to create a function-based index in your own schema on your own table, you must have the `CREATE INDEX` and `QUERY REWRITE` system privileges. To create the index in another schema or on another schema's table, you must have the `CREATE ANY INDEX` and `GLOBAL QUERY REWRITE` privileges. The table owner must also have the `EXECUTE` object privilege on the functions used in the function-based index.

Removing an Index

- Remove an index from the data dictionary by using the **DROP INDEX** command.

```
DROP INDEX index;
```

- Remove the **UPPER_LAST_NAME_IDX** index from the data dictionary.

```
DROP INDEX upper_last_name_idx;  
Index dropped.
```

- To drop an index, you must be the owner of the index or have the **DROP ANY INDEX** privilege.

ORACLE

12-23

Copyright © Oracle Corporation, 2001. All rights reserved.

Removing an Index

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

Note: If you drop a table, indexes and constraints are automatically dropped, but views and sequences remain.

Synonyms

Simplify access to objects by creating a synonym (another name for an object). With synonyms, you can:

- **Ease referring to a table owned by another user**
- **Shorten lengthy object names**

```
CREATE [PUBLIC] SYNONYM synonym
FOR    object;
```

ORACLE

12-24

Copyright © Oracle Corporation, 2001. All rights reserved.

Creating a Synonym for an Object

To refer to a table 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:

<code>PUBLIC</code>	creates a synonym accessible to all users
<code><i>synonym</i></code>	is the name of the synonym to be created
<code><i>object</i></code>	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 owned by the same user.

For more information, see *Oracle9i SQL Reference*, “CREATE SYNONYM.”

Creating and Removing Synonyms

- **Create a shortened name for the DEPT_SUM_VU view.**

```
CREATE SYNONYM d_sum
FOR dept_sum_vu;
Synonym Created.
```

- **Drop a synonym.**

```
DROP SYNONYM d_sum;
Synonym dropped.
```

ORACLE

12-25

Copyright © Oracle Corporation, 2001. All rights reserved.

Creating a Synonym for an Object (continued)

The slide example creates a synonym for the DEPT_SUM_VU view for quicker reference.

The database administrator can create a public synonym 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;
Synonym created.
```

Removing a Synonym

To drop a synonym, use the **DROP SYNONYM** statement. Only the database administrator can drop a public synonym.

```
DROP PUBLIC SYNONYM dept;
Synonym dropped.
```

For more information, see *Oracle9i SQL Reference*, "DROP SYNONYM."

Instructor Note

In the Oracle server, the DBA can specifically grant the **CREATE PUBLIC SYNONYM** privilege to any user, and that user can create public synonyms.

Summary

In this lesson, you should have learned how to:

- Automatically generate sequence numbers by using a sequence generator
- View sequence information in the `USER_SEQUENCES` data dictionary table
- Create indexes to improve query retrieval speed
- View index information in the `USER_INDEXES` dictionary table
- Use synonyms to provide alternative names for objects

ORACLE

12-26

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary

In this lesson you should have learned about some of the other database objects including sequences, indexes, and views.

Sequences

The sequence generator can be used to automatically generate sequence numbers for rows in tables. This can save time and can reduce the amount of application code needed.

A sequence is a database object that can be shared with other users. Information about the sequence can be found in the `USER_SEQUENCES` table of the data dictionary.

To use a sequence, reference it with either the `NEXTVAL` or the `CURRVAL` pseudocolumns.

- Retrieve the next number in the sequence by referencing *sequence*.`NEXTVAL`.
- Return the current available number by referencing *sequence*.`CURRVAL`.

Indexes

Indexes are used to improve query retrieval speed. Users can view the definitions of the indexes in the `USER_INDEXES` data dictionary view. An index can be dropped by the creator, or a user with the `DROP ANY INDEX` privilege, by using the `DROP INDEX` statement.

Synonyms

Database administrators can create public synonyms and users can create private synonyms for convenience, by using the `CREATE SYNONYM` statement. Synonyms permit short names or alternative names for objects. Remove synonyms by using the `DROP SYNONYM` statement.

Practice 12 Overview

This practice covers the following topics:

- **Creating sequences**
- **Using sequences**
- **Creating nonunique indexes**
- **Displaying data dictionary information about sequences and indexes**
- **Dropping indexes**

ORACLE

12-27

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 12 Overview

In this practice, you create a sequence to be used when populating your table. You also create implicit and explicit indexes.

Practice 12

1. Create a sequence to be used with the primary key column of the DEPT table. The sequence should start at 200 and have a maximum value of 1000. Have your sequence increment by ten numbers. Name the sequence DEPT_ID_SEQ.
2. Write a query in a script to display the following information about your sequences: sequence name, maximum value, increment size, and last number. Name the script lab12_2.sql. Run the statement in your script.

SEQUENCE_NAME	MAX_VALUE	INCREMENT_BY	LAST_NUMBER
DEPARTMENTS_SEQ	9990	10	280
DEPT_ID_SEQ	1000	10	200
EMPLOYEES_SEQ	1.0000E+27	1	207
LOCATIONS_SEQ	9900	100	3300

3. Write a script to insert two rows into the DEPT table. Name your script lab12_3.sql. Be sure to use the sequence that you created for the ID column. Add two departments named Education and Administration. Confirm your additions. Run the commands in your script.
4. Create a nonunique index on the foreign key column (DEPT_ID) in the EMP table.
5. Display the indexes and uniqueness that exist in the data dictionary for the EMP table. Save the statement into a script named lab12_5.sql.

INDEX_NAME	TABLE_NAME	UNIQUENES
EMP_DEPT_ID_IDX	EMP	NONUNIQUE
MY_EMP_ID_PK	EMP	UNIQUE

13

Controlling User Access

ORACLE®

Copyright © Oracle Corporation, 2001. All rights reserved.

Schedule:	Timing	Topic
	20 minutes	Lecture
	20 minutes	Practice
	40 minutes	Total

Objectives

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

- **Create users**
- **Create roles to ease setup and maintenance of the security model**
- **Use the GRANT and REVOKE statements to grant and revoke object privileges**
- **Create and access database links**

ORACLE

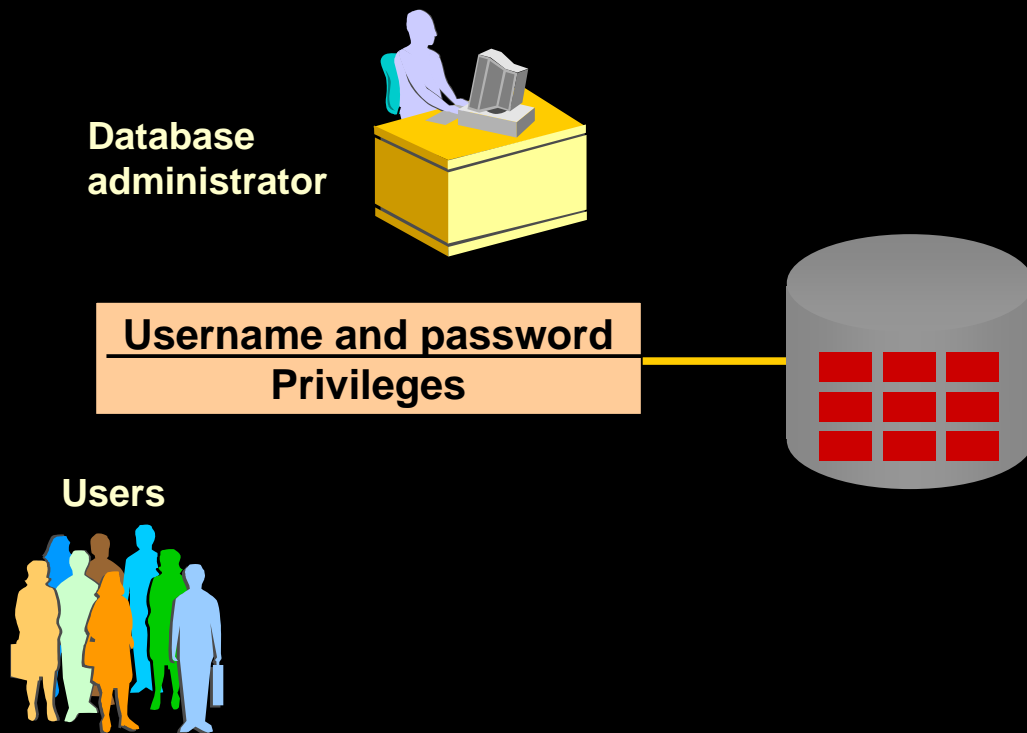
13-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Lesson Aim

In this lesson, you learn how to **control database access** to specific objects and add new users with different levels of access privileges.

Controlling User Access



ORACLE

13-3

Copyright © Oracle Corporation, 2001. All rights reserved.

Controlling User Access

In a multiple-user environment, you want to maintain security of the 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 given and received *privileges* with the Oracle data dictionary
- Create *synonyms* for database objects

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. Database security covers access and use of the database objects and the actions that those users can have on the objects.

Privileges

- **Database security:**
 - System security
 - Data security
- **System privileges: Gaining access to the database**
- **Object privileges: Manipulating the content of the database objects**
- **Schemas: Collections of objects, such as tables, views, and sequences**

ORACLE

13-4

Copyright © Oracle Corporation, 2001. All rights reserved.

Privileges

Privileges are the right to execute particular SQL statements. The database administrator (DBA) is a high-level user with the ability to grant users access to the database and its objects. The 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.

For more information, see *Oracle9i Application Developer's Guide - Fundamentals*, “Establishing a Security Policy” section, and *Oracle9i Concepts*, “Database Security” topic.

System Privileges

- More than 100 privileges are available.
- The database administrator has high-level system privileges for tasks such as:
 - Creating new users
 - Removing users
 - Removing tables
 - Backing up tables

ORACLE

13-5

Copyright © Oracle Corporation, 2001. All rights reserved.

System Privileges

More than 100 distinct system privileges are available for users and roles. System privileges typically are provided by the database administrator.

Typical DBA Privileges

System Privilege	Operations Authorized
CREATE USER	Grantee can create other Oracle users (a privilege required for a DBA role).
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 snapshots in any schema.
CREATE ANY TABLE	Grantee can create tables in any schema.

Creating Users

The DBA creates users by using the **CREATE USER** statement.

```
CREATE USER user  
IDENTIFIED BY password;
```

```
CREATE USER scott  
IDENTIFIED BY tiger;  
User created.
```

ORACLE

13-6

Copyright © Oracle Corporation, 2001. All rights reserved.

Creating a User

The DBA creates the 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 *Oracle9i SQL Reference*, “GRANT” and “CREATE USER.”

Instructor Note

For information on **DROP USER**, refer to *Oracle9i SQL Reference*, “DROP USER.”

User System Privileges

- Once a user is created, the DBA can grant specific system privileges to a user.

```
GRANT privilege [, privilege...]  
TO user [, user/ role, PUBLIC...];
```

- An application developer, for example, may have the following system privileges:
 - CREATE SESSION
 - CREATE TABLE
 - CREATE SEQUENCE
 - CREATE VIEW
 - CREATE PROCEDURE

ORACLE

13-7

Copyright © Oracle Corporation, 2001. All rights reserved.

Typical User Privileges

Now that the DBA has created a user, the DBA can **assign privileges** to that user.

System Privilege	Operations Authorized
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

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 designates that every user is granted the privilege

Note: Current system privileges can be found in the dictionary view SESSION_PRIVS.

Instructor Note

The syntax displayed for the GRANT command is not the full syntax for the statement.

Granting System Privileges

The DBA can grant a user specific system privileges.

```
GRANT  create session, create table,  
        create sequence, create view  
TO      scott;  
Grant succeeded.
```

ORACLE

13-8

Copyright © Oracle Corporation, 2001. All rights reserved.

Granting System Privileges

The DBA uses the **GRANT statement** to allocate system privileges to the user. Once the user has been granted the privileges, the user can immediately use those privileges.

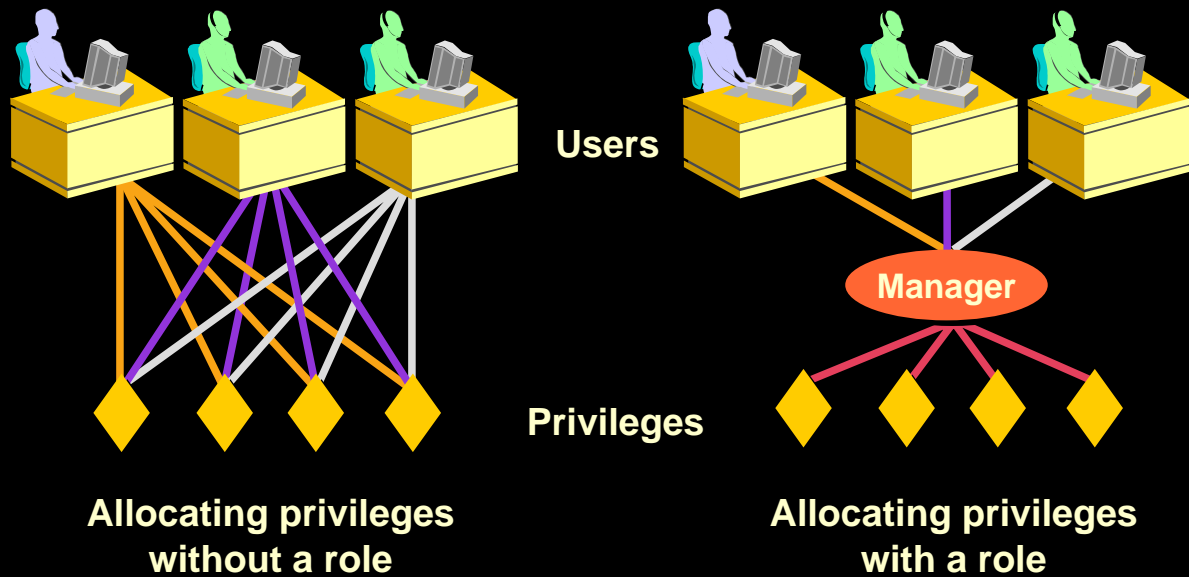
In the example on the slide, user Scott has been assigned the privileges to create sessions, tables, sequences, and views.

Instructor Note

A user needs to have the required space quota to create tables.

Introduction to Oracle9i: SQL 13-8

What is a Role?



ORACLE

13-9

Copyright © Oracle Corporation, 2001. All rights reserved.

What is a Role?

A **role** is a named group of related privileges that can be granted to the 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 users to the role.

Syntax

```
CREATE    ROLE    role;
```

In the syntax:

role is the name of the role to be created

Now that the role is created, the DBA can use the GRANT statement to assign users to the role as well as assign privileges to the role.

Instructor Note

Discuss the following four points about roles:

- Are named groups of related privileges
- Can be granted to users
- Simplify the process of granting and revoking privileges
- Are created by a DBA

Creating and Granting Privileges to a Role

- **Create a role**

```
CREATE ROLE manager;  
Role created.
```

- **Grant privileges to a role**

```
GRANT create table, create view  
TO manager;  
Grant succeeded.
```

- **Grant a role to users**

```
GRANT manager TO DEHAAN, KOCHHAR;  
Grant succeeded.
```

ORACLE

13-10

Copyright © Oracle Corporation, 2001. All rights reserved.

Creating a Role

The example on the slide creates a manager role and then allows managers to create tables and views. It then grants DeHaan and Kochhar the role of managers. Now DeHaan and Kochhar can create tables and views.

If users have multiple roles granted to them, they receive all of the privileges associated with all of 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 scott  
IDENTIFIED BY lion;  
User altered.
```

ORACLE

13-11

Copyright © Oracle Corporation, 2001. All rights reserved.

Changing Your Password

The DBA **creates an account** and initializes a password for every user. You can change your password by using the `ALTER USER` statement.

Syntax

```
ALTER USER user IDENTIFIED BY password;
```

In the syntax:

user is the name of the user

password specifies the new password

Although this statement can be used to change your password, there are many other options. You must have the `ALTER USER` privilege to change any other option.

For more information, see *Oracle9i SQL Reference*, “`ALTER USER`.”

Object Privileges

Object Privilege	Table	View	Sequence	Procedure
ALTER	√		√	
DELETE	√	√		
EXECUTE				√
INDEX	√			
INSERT	√	√		
REFERENCES	√	√		
SELECT	√	√	√	
UPDATE	√	√		

ORACLE

13-12

Copyright © Oracle Corporation, 2001. All rights reserved.

Object Privileges

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. The table on the slide lists the privileges for various objects. Note that the only privileges that apply to a sequence are SELECT and ALTER. UPDATE, REFERENCES, and INSERT can be restricted by specifying a subset of updateable columns. 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.

Instructor Note

You can use the ALTER VIEW and ALTER PROCEDURE commands to recompile views and PL/SQL procedures, functions, and packages.

Object Privileges

- Object privileges vary from object to object.
- An owner has all the privileges on the object.
- An owner can give specific privileges on that owner's object.

```
GRANT      object_priv [(columns)]  
ON         object  
TO         {user|role|PUBLIC}  
[WITH GRANT OPTION];
```

ORACLE

13-13

Copyright © Oracle Corporation, 2001. All rights reserved.

Granting Object Privileges

Different object privileges are available for different types of schema objects. 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**, then 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.

In the syntax:

<i>object_priv</i>	is an object privilege to be granted
ALL	specifies all object privileges
<i>columns</i>	specifies the column from a table or view on which privileges are granted
ON <i>object</i>	is the object on which the privileges are granted
TO	identifies to whom the privilege is granted
PUBLIC	grants object privileges to all users
WITH GRANT OPTION	allows the grantee to grant the object privileges to other users and roles

Granting Object Privileges

- Grant query privileges on the **EMPLOYEES** table.

```
GRANT  select
ON      employees
TO      sue, rich;
Grant succeeded.
```

- Grant privileges to update specific columns to users and roles.

```
GRANT  update (department_name, location_id)
ON      departments
TO      scott, manager;
Grant succeeded.
```

ORACLE

13-14

Copyright © Oracle Corporation, 2001. All rights reserved.

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 on the slide grants users Sue and Rich the privilege to query your **EMPLOYEES** table. The second example grants **UPDATE privileges** on specific columns in the **DEPARTMENTS** table to Scott and to the manager role.

If Sue or Rich now want to **SELECT** data from the employees table, the syntax they must use is:

```
SELECT  *
FROM    scott.employees;
```

Alternatively, they can create a synonym for the table and **SELECT** from the synonym:

```
CREATE SYNONYM emp FOR scott.employees;
SELECT * FROM emp;
```

Note: DBAs generally allocate system privileges; any user who owns an object can grant object privileges.

Instructor Note

Please read the [Instructor Note](#) at the end of this lesson.

Using the WITH GRANT OPTION and PUBLIC Keywords

- Give a user authority to pass along privileges.

```
GRANT  select, insert
ON     departments
TO     scott
WITH   GRANT OPTION;
Grant succeeded.
```

- Allow all users on the system to query data from Alice's DEPARTMENTS table.

```
GRANT  select
ON     alice.departments
TO     PUBLIC;
Grant succeeded.
```

ORACLE

13-15

Copyright © Oracle Corporation, 2001. All rights reserved.

The WITH GRANT OPTION Keyword

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.

The example on the slide gives user Scott access to your DEPARTMENTS table with the privileges to query the table and add rows to the table. The example also allows Scott to give others these privileges.

The 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 Alice's DEPARTMENTS table.

Instructor Note

If a statement does not use the full name of an object, the Oracle server implicitly prefixes the object name with the current user's name (or schema). If user Scott queries the DEPARTMENTS table, for example, the system selects from the SCOTT.DEPARTMENTS table.

If a statement does not use the full name of an object, and the current user does not own an object of that name, the system prefixes the object name with PUBLIC. For example, if user Scott queries the USER_OBJECTS table, and Scott does not own such a table, the system selects from the data dictionary view by way of the PUBLIC.USER_OBJECTS public synonym.

Confirming Privileges Granted

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_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
USER_SYS_PRIVS	Lists system privileges granted to the user

ORACLE

13-16

Copyright © Oracle Corporation, 2001. All rights reserved.

Confirming Granted Privileges

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

You can access the data dictionary to view the privileges that you have. The chart on the slide describes various data dictionary views.

How to Revoke 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}
[CASCADE CONSTRAINTS];
```

ORACLE

13-17

Copyright © Oracle Corporation, 2001. All rights reserved.

Revoking Object Privileges

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 through the **WITH GRANT OPTION** clause.

In the syntax:

CASCADE	is required to remove any referential integrity constraints made to the
CONSTRAINTS	object by means of the REFERENCES privilege

For more information, see *Oracle9i SQL Reference*, “REVOKE.”

Revoking Object Privileges

As user Alice, revoke the **SELECT** and **INSERT** privileges given to user **Scott** on the **DEPARTMENTS** table.

```
REVOKE select, insert
ON      departments
FROM    scott;
Revoke succeeded.
```

ORACLE

13-18

Copyright © Oracle Corporation, 2001. All rights reserved.

Revoking Object Privileges (continued)

The example on the slide revokes **SELECT** and **INSERT** privileges given to user **Scott** 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 are permitted. If the owner revokes a privilege from a user who granted the privilege to other users, the revoking cascades to all privileges granted.

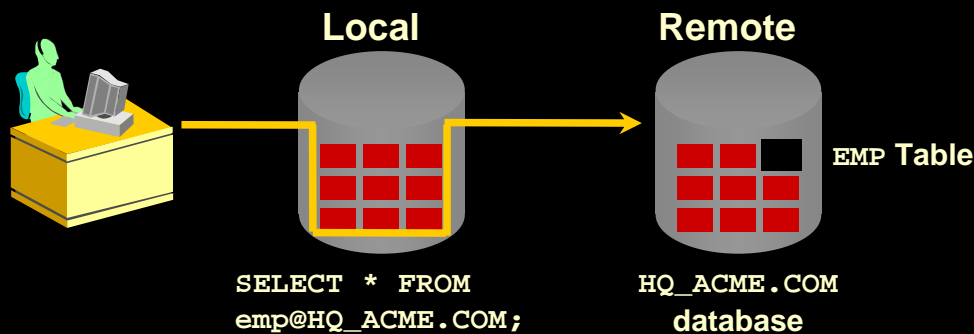
For example, if user A grants **SELECT** privilege on a table to user B including the **WITH GRANT OPTION** clause, user B can grant to user C the **SELECT** privilege with the **WITH GRANT OPTION** clause as well, and user C can then grant to user D the **SELECT** privilege. If user A revokes privilege from user B, then the privileges granted to users C and D are also revoked.

Instructor Note

Revoking system privileges is not within the scope of this lesson. For information on this topic refer to: *Oracle9i SQL Reference*, “[REVOKE system_privileges_and_roles](#).”

Database Links

A database link connection allows local users to access data on a remote database.



ORACLE

13-19

Copyright © Oracle Corporation, 2001. All rights reserved.

Database Links

A database link is a pointer that defines a one-way communication path from an Oracle database server to another database server. The link pointer is actually defined as an entry in a data dictionary table. To access the link, you must be connected to the local database that contains the data dictionary entry.

A **database link** connection is one-way in the sense that a client connected to local database A can use a link stored in database A to access information in remote database B, but users connected to database B cannot use the same link to access data in database A. If local users on database B want to access data on database A, they must define a link that is stored in the data dictionary of database B.

A database link connection gives local users access to data on a remote database. For this connection to occur, each database in the distributed system must have a unique global database name. The global database name uniquely identifies a database server in a distributed system.

The great advantage of database links is that they allow users to access another user's objects in a remote database so that they are bounded by the privilege set of the object's owner. In other words, a local user can access a remote database without having to be a user on the remote database.

The example shows a user SCOTT accessing the EMP table on the remote database with the global name HQ.ACME.COM.

Note: Typically, the DBA is responsible for creating the database link. The dictionary view **USER_DB_LINKS** contains information on links to which a user has access.

Database Links

- **Create the database link.**

```
CREATE PUBLIC DATABASE LINK hq.acme.com  
USING 'sales';  
Database link created.
```

- **Write SQL statements that use the database link.**

```
SELECT *  
FROM emp@HQ.ACME.COM;
```

ORACLE

13-20

Copyright © Oracle Corporation, 2001. All rights reserved.

Using Database Links

The example shown creates a database link. The **USING clause** identifies the service name of a remote database.

Once the database link is created, you can write SQL statements against the data in the remote site. If a synonym is set up, you can write SQL statements using the synonym.

For example:

```
CREATE PUBLIC SYNONYM HQ_EMP FOR emp@HQ.ACME.COM;
```

Then write a SQL statement that uses the synonym:

```
SELECT * FROM HQ_EMP;
```

You cannot grant privileges on remote objects.

Instructor Note

Let the students know that using distributed databases encompasses much more than what is shown here. If the students want more information, refer them to the *Oracle9i Concepts*, “Distributed Database Concepts.”

Summary

In this lesson, you should have learned about DCL statements that control access to the database and database objects:

Statement	Action
CREATE USER	Creates a user (usually performed by a DBA)
GRANT	Gives other users privileges to access the your objects
CREATE ROLE	Creates a collection of privileges (usually performed by a DBA)
ALTER USER	Changes a user's password
REVOKE	Removes privileges on an object from users

ORACLE

13-21

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary

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.
- Once 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 password 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.
- With database links, you can access data on remote databases. Privileges cannot be granted on remote objects.

Practice 13 Overview

This practice covers the following topics:

- **Granting other users privileges to your table**
- **Modifying another user's table through the privileges granted to you**
- **Creating a synonym**
- **Querying the data dictionary views related to privileges**

ORACLE

13-22

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 13 Overview

Team up with other students for this exercise about controlling access to database objects.

Instructor Note

For this practice, divide the students into teams, and then pair off the teams so that half are Team 1s and the other half are Team 2s.

Practice 13

1. What privilege should a user be given to log on to the Oracle Server? Is this a system or an object privilege?

2. What privilege should a user be given to create tables?

3. If you create a table, who can pass along privileges to other users on your table?

4. You are the DBA. You are creating many users who require the same system privileges. What should you use to make your job easier?

5. What command do you use to change your password?

6. Grant another user access to your DEPARTMENTS table. Have the user grant you query access to his or her DEPARTMENTS table.
7. Query all the rows in your DEPARTMENTS table.

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		1700

8 rows selected.

8. Add a new row to your DEPARTMENTS table. Team 1 should add Education as department number 500. Team 2 should add Human Resources department number 510. Query the other team's table.
9. Create a synonym for the other team's DEPARTMENTS table.

Practice 13 (continued)

10. Query all the rows in the other team's DEPARTMENTS table by using your synonym.

Team 1 SELECT statement results:

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		1700
500	Education		

9 rows selected.

Team 2 SELECT statement results:

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		1700
510	Human Resources		

9 rows selected.

Practice 13 (continued)

11. Query the USER_TABLES data dictionary to see information about the tables that you own.

TABLE_NAME
COUNTRIES
DEPARTMENTS
DEPT
EMP
EMPLOYEES
JOBS
JOB_GRADES
JOB_HISTORY
LOCATIONS
REGIONS

10 rows selected.

12. Query the ALL_TABLES data dictionary view to see information about all the tables that you can access. Exclude tables that you own.

Note: Your list may not exactly match the list shown below.

TABLE_NAME	OWNER
DEPARTMENTS	<i>owner</i>

13. Revoke the SELECT privilege on your table from the other team.
14. Remove the row you inserted into the DEPARTMENTS table in step 8 and save the changes.

Instructor Note (for pages 13-14)

Let students know that the *privilege(col,col)* syntax can be used only with UPDATE. Most students try to use this syntax with SELECT as shown below:

```
GRANT SELECT(salary,last_name)
ON employees TO scott;
```

The above syntax returns the error ERROR at line 1:ORA-00969: missing ON keyword.

Instructor Note

Let students know about fine-grained access control. Using fine-grained access control, you can implement security policies with functions and then associate those security policies with tables or views. The database server automatically enforces those security policies, no matter how the data is accessed (for example, by ad hoc queries).

You can:

- Use different policies for SELECT, INSERT, UPDATE, and DELETE commands
- Use security policies only where you need them (for example, on salary information)
- Use more than one policy for each table, including building on top of base policies in packaged applications

For the implementation of fine-grained access control, you may need to use functions or packages in PL/SQL. The PL/SQL DBMS_RLS package enables you to administer your security policies. Using this package, you can add, drop, enable, disable, and refresh the policies you create. For more information on implementing fine-grained access control, refer to: *Oracle9i Concepts*, “Fine-Grained Access Control.”

14

SQL Workshop

ORACLE®

Copyright © Oracle Corporation, 2001. All rights reserved.

Workshop Overview

This workshop covers:

- **Creating tables and sequences**
- **Modifying data in the tables**
- **Modifying table definitions**
- **Creating views**
- **Writing scripts containing SQL and *iSQL*Plus* commands**
- **Generating a simple report**

ORACLE

14-2

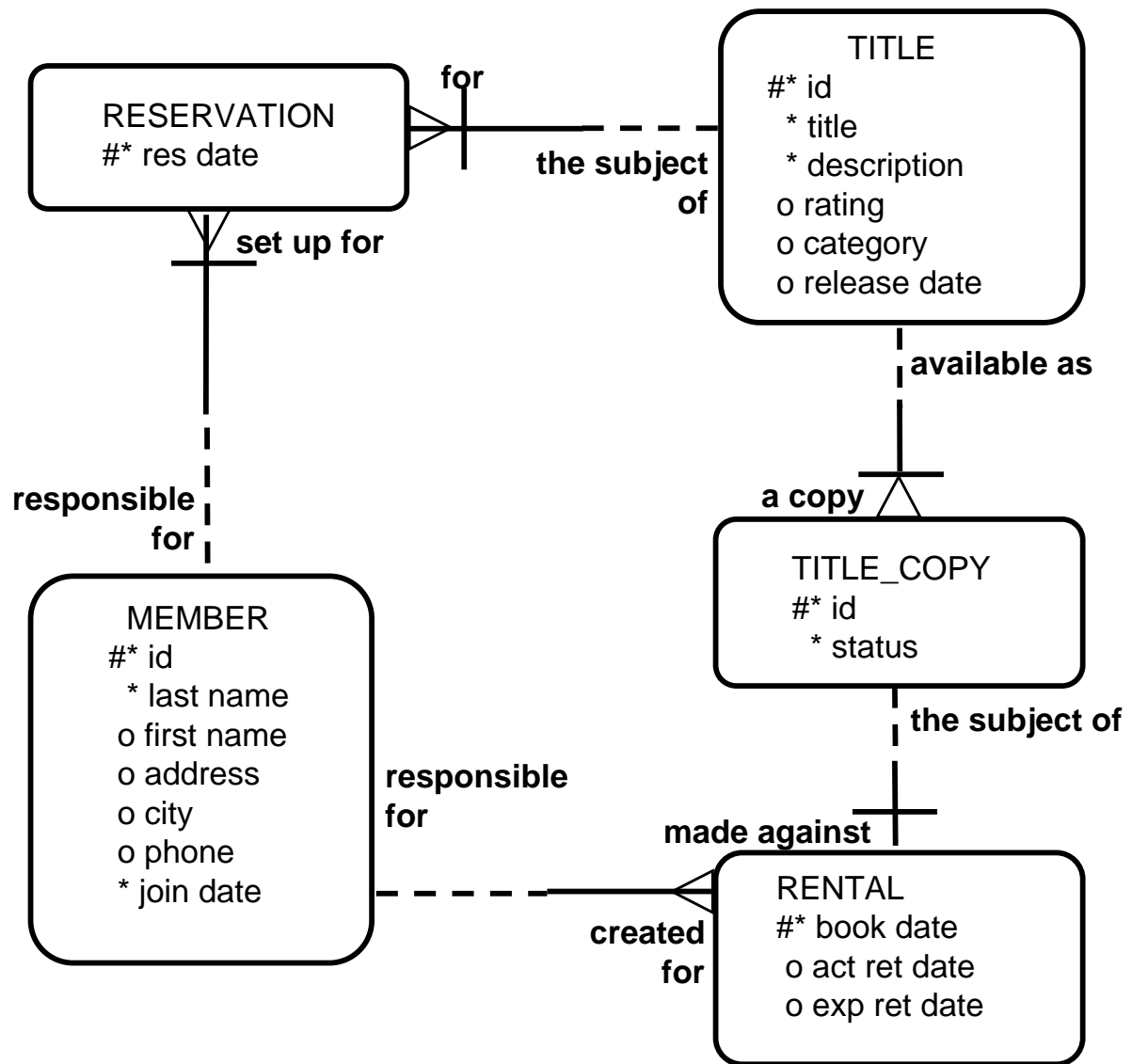
Copyright © Oracle Corporation, 2001. All rights reserved.

Workshop Overview

In this workshop you build a set of database tables for a video application. After you create the tables, you insert, update, and delete records in a video store database and generate a report. The database contains only the essential tables.

Note: If you want to build the tables, you can execute the commands in the `buildtab.sql` script in *iSQL*Plus*. If you want to drop the tables, you can execute the commands in `dropvid.sql` script in *iSQL*Plus*. Then you can execute the commands in `buildvid.sql` script in *iSQL*Plus* to create and populate the tables. If you use the `buildvid.sql` script to build and populate the tables, start with step 6b.

Video Application Entity Relationship Diagram



Practice 14

- Create the tables based on the following table instance charts. Choose the appropriate data types and be sure to add integrity constraints.

a. Table name: MEMBER

Column_ Name	MEMBER_ ID	LAST_ NAME	FIRST_NAM E	ADDRESS	CITY	PHONE	JOIN _ DATE
Key Type	PK						
Null/ Unique	NN,U	NN					NN
Default Value							System Date
Data Type	NUMBER	VARCHAR2	VARCHAR2	VARCHAR2	VARCHAR2	VARCHAR2	DATE
Length	10	25	25	100	30	15	

b. Table name: TITLE

Column_ Name	TITLE_ID	TITLE	DESCRIPTION	RATING	CATEGORY	RELEASE_ DATE
Key Type	PK					
Null/ Unique	NN,U	NN	NN			
Check				G, PG, R, NC17, NR	DRAMA, COMEDY, ACTION, CHILD, SCIFI, DOCUMENTARY	
Data Type	NUMBER	VARCHAR2	VARCHAR2	VARCHAR2	VARCHAR2	DATE
Length	10	60	400	4	20	

Practice 14 (continued)

c. Table name: TITLE_COPY

Column Name	COPY_ID	TITLE_ID	STATUS
Key Type	PK	PK,FK	
Null/Unique	NN,U	NN,U	NN
Check			AVAILABLE, DESTROYED, RENTED, RESERVED
FK Ref Table		TITLE	
FK Ref Col		TITLE_ID	
Data Type	NUMBER	NUMBER	VARCHAR2
Length	10	10	15

d. Table name: RENTAL

Column Name	BOOK_DATE	MEMBER_ID	COPY_ID	ACT_RET_DATE	EXP_RET_DATE	TITLE_ID
Key Type	PK	PK,FK1	PK,FK2			PK,FK2
Default Value	System Date				System Date + 2 days	
FK Ref Table		MEMBER	TITLE_COPY			TITLE_COPY
FK Ref Col		MEMBER_ID	COPY_ID			TITLE_ID
Data Type	DATE	NUMBER	NUMBER	DATE	DATE	NUMBER
Length		10	10			10

Practice 14 (continued)

e. Table name: RESERVATION

Column Name	RES_ DATE	MEMBER_ ID	TITLE_ ID
Key Type	PK	PK,FK1	PK,FK2
Null/Unique	NN,U	NN,U	NN
FK Ref Table		MEMBER	TITLE
FK Ref Column		MEMBER_ID	TITLE_ID
Data Type	DATE	NUMBER	NUMBER
Length		10	10

- Verify that the tables and constraints were created properly by checking the data dictionary.

TABLE_NAME
MEMBER
RENTAL
RESERVATION
TITLE
TITLE_COPY

CONSTRAINT_NAME	C	TABLE_NAME
MEMBER_LAST_NAME_NN	C	MEMBER
MEMBER_JOIN_DATE_NN	C	MEMBER
MEMBER_MEMBER_ID_PK	P	MEMBER
RENTAL_BOOK_DATE_COPY_TITLE_PK	P	RENTAL
RENTAL_MEMBER_ID_FK	R	RENTAL
RENTAL_COPY_ID_TITLE_ID_FK	R	RENTAL
RESERVATION_RESDATE_MEM_TIT_PK	P	RESERVATION
RESERVATION_MEMBER_ID	R	RESERVATION
RESERVATION_TITLE_ID	R	RESERVATION
TITLE_TITLE_NN	C	TITLE

■ ■ ■

18 rows selected.

Practice 14 (continued)

3. Create sequences to uniquely identify each row in the MEMBER table and the TITLE table.
 - a. Member number for the MEMBER table: Start with 101; do not allow caching of the values. Name the sequence MEMBER_ID_SEQ.
 - b. Title number for the TITLE table: Start with 92; no caching. Name the sequence TITLE_ID_SEQ.
 - c. Verify the existence of the sequences in the data dictionary.

SEQUENCE_NAME	INCREMENT_BY	LAST_NUMBER
TITLE_ID_SEQ	1	92
MEMBER_ID_SEQ	1	101

4. Add data to the tables. Create a script for each set of data to add.
 - a. Add movie titles to the TITLE table. Write a script to enter the movie information. Save the statements in a script named lab14_4a.sql. Use the sequences to uniquely identify each title. Enter the release dates in the DD-MON-YYYY format. Remember that single quotation marks in a character field must be specially handled. Verify your additions.

TITLE
Willie and Christmas Too
Alien Again
The Glob
My Day Off
Miracles on Ice
Soda Gang

6 rows selected.

Practice 14 (continued)

Title	Description	Rating	Category	Release_date
Willie and Christmas Too	All of Willie's friends make a Christmas list for Santa, but Willie has yet to add his own wish list.	G	CHILD	05-OCT-1995
Alien Again	Yet another installation of science fiction history. Can the heroine save the planet from the alien life form?	R	SCIFI	19-MAY-1995
The Glob	A meteor crashes near a small American town and unleashes carnivorous goo in this classic.	NR	SCIFI	12-AUG-1995
My Day Off	With a little luck and a lot of ingenuity, a teenager skips school for a day in New York.	PG	COMEDY	12-JUL-1995
Miracles on Ice	A six-year-old has doubts about Santa Claus, but she discovers that miracles really do exist.	PG	DRAMA	12-SEP-1995
Soda Gang	After discovering a cache of drugs, a young couple find themselves pitted against a vicious gang.	NR	ACTION	01-JUN-1995

Practice 14 (continued)

- b. Add data to the MEMBER table. Place the insert statements in a script named lab14_4b.sql. Execute commands in the script. Be sure to use the sequence to add the member numbers.

First_Name	Last_Name	Address	City	Phone	Join_Date
Carmen	Velasquez	283 King Street	Seattle	206-899-6666	08-MAR-1990
LaDoris	Ngao	5 Modrany	Bratislava	586-355-8882	08-MAR-1990
Midori	Nagayama	68 Via Centrale	Sao Paolo	254-852-5764	17-JUN-1991
Mark	Quick-to-See	6921 King Way	Lagos	63-559-7777	07-APR-1990
Audry	Ropeburn	86 Chu Street	Hong Kong	41-559-87	18-JAN-1991
Molly	Urguhart	3035 Laurier	Quebec	418-542-9988	18-JAN-1991

Practice 14 (continued)

- c. Add the following movie copies in the TITLE_COPY table:

Note: Have the TITLE_ID numbers available for this exercise.

Title	Copy_Id	Status
Willie and Christmas Too	1	AVAILABLE
Alien Again	1	AVAILABLE
	2	RENTED
The Glob	1	AVAILABLE
My Day Off	1	AVAILABLE
	2	AVAILABLE
	3	RENTED
Miracles on Ice	1	AVAILABLE
Soda Gang	1	AVAILABLE

- d. Add the following rentals to the RENTAL table:

Note: Title number may be different depending on sequence number.

Title_Id	Copy_Id	Member_Id	Book_date	Exp_Ret_Date	Act_Ret_Date
92	1	101	3 days ago	1 day ago	2 days ago
93	2	101	1 day ago	1 day from now	
95	3	102	2 days ago	Today	
97	1	106	4 days ago	2 days ago	2 days ago

Practice 14 (continued)

5. Create a view named `TITLE_AVAIL` to show the movie titles and the availability of each copy and its expected return date if rented. Query all rows from the view. Order the results by title.

Note: Your results may be different.

TITLE	COPY_ID	STATUS	EXP_RET_D
Alien Again	1	AVAILABLE	
Alien Again	2	RENTED	26-SEP-01
Miracles on Ice	1	AVAILABLE	
My Day Off	1	AVAILABLE	
My Day Off	2	AVAILABLE	
My Day Off	3	RENTED	27-SEP-01
Soda Gang	1	AVAILABLE	25-SEP-01
The Glob	1	AVAILABLE	
Willie and Christmas Too	1	AVAILABLE	26-SEP-01

9 rows selected.

6. Make changes to data in the tables.
 - a. Add a new title. The movie is “Interstellar Wars,” which is rated PG and classified as a science fiction movie. The release date is 07-JUL-77. The description is “Futuristic interstellar action movie. Can the rebels save the humans from the evil empire?” Be sure to add a title copy record for two copies.
 - b. Enter two reservations. One reservation is for Carmen Velasquez, who wants to rent “Interstellar Wars.” The other is for Mark Quick-to-See, who wants to rent “Soda Gang.”

Practice 14 (continued)

- c. Customer Carmen Velasquez rents the movie “Interstellar Wars,” copy 1. Remove her reservation for the movie. Record the information about the rental. Allow the default value for the expected return date to be used. Verify that the rental was recorded by using the view you created.

Note: Your results may be different.

TITLE	COPY_ID	STATUS	EXP_RET_D
Alien Again	1	AVAILABLE	
Alien Again	2	RENTED	26-SEP-01
Interstellar Wars	1	RENTED	29-SEP-01
Interstellar Wars	2	AVAILABLE	
Miracles on Ice	1	AVAILABLE	
My Day Off	1	AVAILABLE	
My Day Off	2	AVAILABLE	
My Day Off	3	RENTED	27-SEP-01
Soda Gang	1	AVAILABLE	25-SEP-01
The Glob	1	AVAILABLE	
Willie and Christmas Too	1	AVAILABLE	26-SEP-01

11 rows selected.

7. Make a modification to one of the tables.
 - a. Add a PRICE column to the TITLE table to record the purchase price of the video. The column should have a total length of eight digits and two decimal places. Verify your modifications.

Name	Null?	Type
TITLE_ID	NOT NULL	NUMBER(10)
TITLE	NOT NULL	VARCHAR2(60)
DESCRIPTION	NOT NULL	VARCHAR2(400)
RATING		VARCHAR2(4)
CATEGORY		VARCHAR2(20)
RELEASE_DATE		DATE
PRICE		NUMBER(8,2)

Practice 14 (continued)

- b. Create a script named `lab14_7b.sql` that contains update statements that update each video with a price according to the following list. Run the commands in the script.

Note: Have the `TITLE_ID` numbers available for this exercise.

Title	Price
Willie and Christmas Too	25
Alien Again	35
The Glob	35
My Day Off	35
Miracles on Ice	30
Soda Gang	35
Interstellar Wars	29

- c. Make sure that in the future all titles contain a price value. Verify the constraint.

CONSTRAINT_NAME	C	SEARCH_CONDITION
TITLE_TITLE_NN	C	"TITLE" IS NOT NULL
■ ■ ■		
TITLE_PRICE_NN	C	"PRICE" IS NOT NULL

6 rows selected.

8. Create a report titled Customer History Report. This report contains each customer's history of renting videos. Be sure to include the customer name, movie rented, dates of the rental, and duration of rentals. Total the number of rentals for all customers for the reporting period. Save the commands that generate the report in a script file named `lab14_8.sql`.

Note: Your results may be different.

Thu Sep 27

Customer History Report

page 1

MEMBER	TITLE	BOOK_DATE	DURATION
Carmen Velasquez	Willie and Christmas Too	24-SEP-01	1
	Alien Again	26-SEP-01	
	Interstellar Wars	27-SEP-01	
LaDoris Ngao	My Day Off	25-SEP-01	
Molly Urguhart	Soda Gang	23-SEP-01	2

15

Using SET Operators

ORACLE

Copyright © Oracle Corporation, 2001. All rights reserved.

Schedule:	Timing	Topic
	30 minutes	Lecture
	20 minutes	Practice
	50 minutes	Total

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

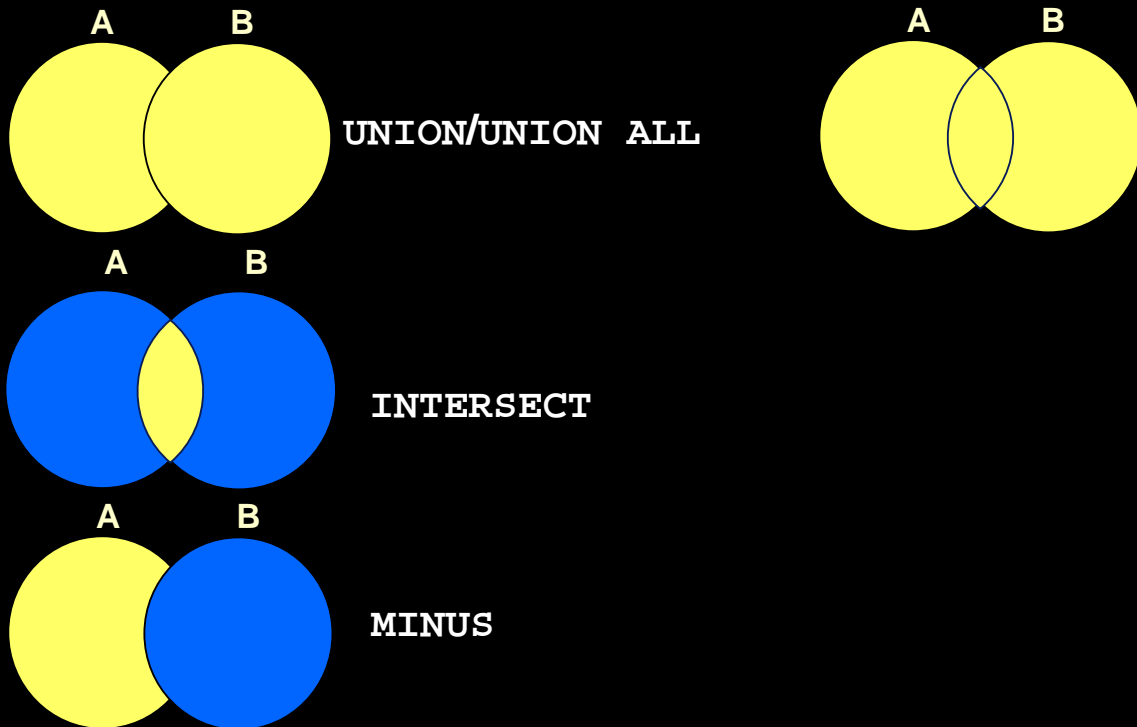
15-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Lesson Aim

In this lesson, you learn how to write queries by using `SET` operators.

The SET Operators



ORACLE

15-3

Copyright © Oracle Corporation, 2001. All rights reserved.

The SET Operators

The **SET operators** combine the results of two or more component queries into one result. Queries containing SET operators are called *compound queries*.

Operator	Returns
UNION	All distinct rows selected by either query
UNION ALL	All rows selected by either query, including all duplicates
INTERSECT	All distinct rows selected by both queries
MINUS	All distinct rows that are selected by the first SELECT statement and not selected in the second SELECT statement

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.

Note: In the slide, the light color (gray) in the diagram represents the query result.

Instructor Note

The INTERSECT and MINUS operators are not ANSI SQL-99 compliant. They are Oracle-specific.

Tables Used in This Lesson

The tables used in this lesson are:

- **EMPLOYEES:** Provides details regarding all current employees
- **JOB_HISTORY:** Records the details of the start date and end date of the former job, and the job identification number and department when an employee switches jobs

ORACLE

15-4

Copyright © Oracle Corporation, 2001. All rights reserved.

Tables Used in This Lesson

Two tables are used in this lesson. They are the `EMPLOYEES` table and the `JOB_HISTORY` table.

The `EMPLOYEES` table stores the employee details. For the human resource records, this table stores a unique identification number and email address for each employee. The details of the employee's job identification number, salary, and manager are also stored. Some of the employees earn a commission in addition to their salary; this information is tracked too. The company organizes the roles of employees into jobs. Some of the employees have been with the company for a long time and have switched to different jobs. This is monitored using the `JOB_HISTORY` table. When an employee switches jobs, the details of the start date and end date of the former job, the job identification number and department are recorded in the `JOB_HISTORY` table.

The structure and the data from the `EMPLOYEES` and the `JOB_HISTORY` tables are shown on the next page.

There have been instances in the company of people who have held the same position more than once during their tenure with the company. For example, consider the employee Taylor, who joined the company on 24-MAR-1998. Taylor held the job title `SA_REP` for the period 24-MAR-98 to 31-DEC-98 and the job title `SA_MAN` for the period 01-JAN-99 to 31-DEC-99. Taylor moved back into the job title of `SA_REP`, which is his current job title.

Similarly consider the employee Whalen, who joined the company on 17-SEP-1987. Whalen held the job title `AD_ASST` for the period 17-SEP-87 to 17-JUN-93 and the job title `AC_ACCOUNT` for the period 01-JUL-94 to 31-DEC-98. Whalen moved back into the job title of `AD_ASST`, which is his current job title.

Introduction to Oracle9i: SQL 15-4

Tables Used in This Lesson (continued)

DESC 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)
DEPARTMENT_NAME		VARCHAR2(14)

```
SELECT employee_id, last_name, job_id, hire_date, department_id
FROM employees;
```

EMPLOYEE_ID	LAST_NAME	JOB_ID	HIRE_DATE	DEPARTMENT_ID
100	King	AD_PRES	17-JUN-87	90
101	Kochhar	AD_VP	21-SEP-89	90
102	De Haan	AD_VP	13-JAN-93	90
103	Hunold	IT_PROG	03-JAN-90	60
104	Ernst	IT_PROG	21-MAY-91	60
107	Lorentz	IT_PROG	07-FEB-99	60
124	Mourgos	ST_MAN	16-NOV-99	50
141	Rajs	ST_CLERK	17-OCT-95	50
142	Davies	ST_CLERK	29-JAN-97	50
143	Matos	ST_CLERK	15-MAR-98	50
144	Vargas	ST_CLERK	09-JUL-98	50
149	Zlotkey	SA_MAN	29-JAN-00	80
174	Abel	SA_REP	11-MAY-96	80
176	Taylor	SA_REP	24-MAR-98	80
EMPLOYEE_ID	LAST_NAME	JOB_ID	HIRE_DATE	DEPARTMENT_ID
178	Grant	SA_REP	24-MAY-99	
200	Whalen	AD_ASST	17-SEP-87	10
201	Hartstein	MK_MAN	17-FEB-96	20

■ ■ ■

20 rows selected.

Tables Used in This Lesson (continued)

DESC job_history

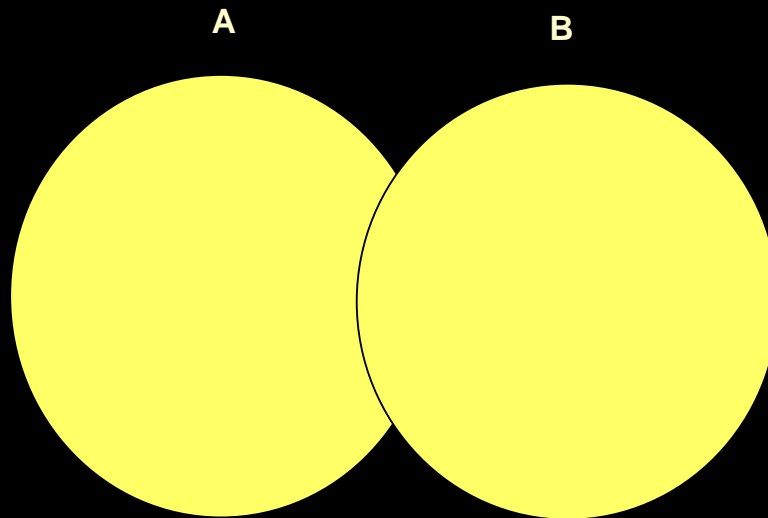
Name	Null?	Type
EMPLOYEE_ID	NOT NULL	NUMBER(6)
START_DATE	NOT NULL	DATE
END_DATE	NOT NULL	DATE
JOB_ID	NOT NULL	VARCHAR2(10)
DEPARTMENT_ID		NUMBER(4)

SELECT * FROM job_history;

EMPLOYEE_ID	START_DATE	END_DATE	JOB_ID	DEPARTMENT_ID
102	13-JAN-93	24-JUL-98	IT_PROG	60
101	21-SEP-89	27-OCT-93	AC_ACCOUNT	110
101	28-OCT-93	15-MAR-97	AC_MGR	110
201	17-FEB-96	19-DEC-99	MK_REP	20
114	24-MAR-98	31-DEC-99	ST_CLERK	50
122	01-JAN-99	31-DEC-99	ST_CLERK	50
200	17-SEP-87	17-JUN-93	AD_ASST	90
176	24-MAR-98	31-DEC-98	SA_REP	80
176	01-JAN-99	31-DEC-99	SA_MAN	80
200	01-JUL-94	31-DEC-98	AC_ACCOUNT	90

10 rows selected.

The UNION Operator



The **UNION** operator returns results from both queries after eliminating duplications.

ORACLE

15-7

Copyright © Oracle Corporation, 2001. All rights reserved.

The UNION Operator

The **UNION** operator returns all rows 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 and the datatypes of the columns being selected must be identical in all the **SELECT** statements used in the query. The names of the columns need not be identical.
- **UNION** operates over all of the columns being selected.
- **NULL** values are not ignored during duplicate checking.
- The **IN** operator has a higher precedence than the **UNION** operator.
- By default, the output is sorted in ascending order of the first column of the **SELECT** clause.

Instructor Note

To illustrate the **UNION SET** operator, run the script `demo\15_union1.sql`.

Point out that the output is sorted in ascending order of the first column of the **SELECT** clause.

Using the UNION Operator

Display the current and previous job details of all employees. Display each employee only once.

```
SELECT employee_id, job_id
FROM employees
UNION
SELECT employee_id, job_id
FROM job_history;
```

EMPLOYEE_ID	JOB_ID
100	AD_PRES
101	AC_ACCOUNT
...	
200	AC_ACCOUNT
200	AD_ASST
...	
205	AC_MGR
206	AC_ACCOUNT

ORACLE

15-8

Copyright © Oracle Corporation, 2001. All rights reserved.

Using the UNION SET Operator

The **UNION operator** eliminates any duplicate records. If there are records that occur both in the EMPLOYEES and the JOB_HISTORY tables and are identical, the records will be displayed only once. Observe in the output shown on the slide that the record for the employee with the EMPLOYEE_ID 200 appears twice as the JOB_ID is different in each row.

Consider the following example:

```
SELECT employee_id, job_id, department_id
FROM employees
UNION
SELECT employee_id, job_id, department_id
FROM job_history;
```

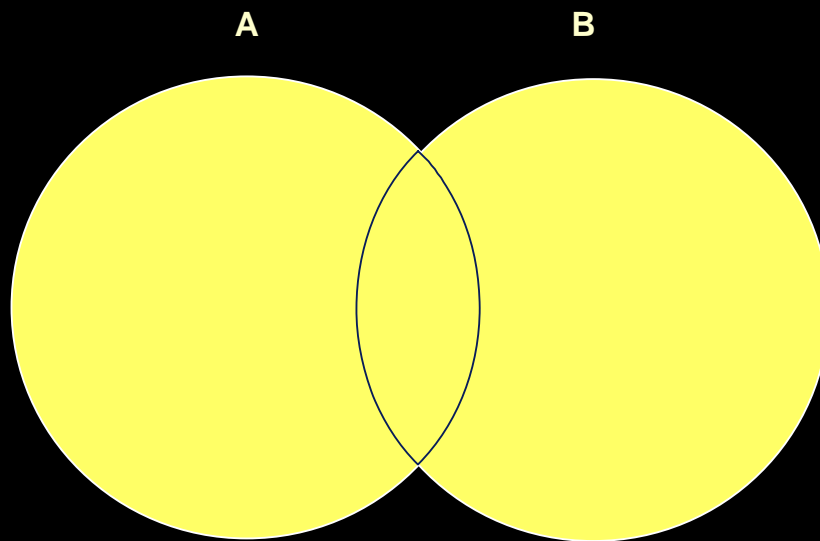
EMPLOYEE_ID	JOB_ID	DEPARTMENT_ID
...		
200	AC_ACCOUNT	90
200	AD_ASST	10
200	AD_ASST	90

29 rows selected.

Using the UNION SET Operator (continued)

In the preceding output, employee 200 appears three times. Why? Notice the DEPARTMENT_ID values for employee 200. One row has a DEPARTMENT_ID of 90, another 10, and the third 90. Because of these unique combinations of job IDs and department IDs, each row for employee 200 is unique and therefore not considered a duplicate. Observe that the output is sorted in ascending order of the first column of the SELECT clause, EMPLOYEE_ID in this case.

The UNION ALL Operator



The UNION ALL operator returns results from both queries, including all duplications.

ORACLE

15-10

Copyright © Oracle Corporation, 2001. All rights reserved.

The UNION ALL Operator

Use the UNION ALL operator to return all rows from multiple queries.

Guidelines

- Unlike UNION, duplicate rows are not eliminated and the output is not sorted by default.
- The DISTINCT keyword cannot be used.

Note: With the exception of the above, the guidelines for UNION and UNION ALL are the same.

Using the UNION ALL Operator

Display the current and previous departments of all employees.

```
SELECT employee_id, job_id, department_id
FROM employees
UNION ALL
SELECT employee_id, job_id, department_id
FROM job_history
ORDER BY employee_id;
```

EMPLOYEE_ID	JOB_ID	DEPARTMENT_ID
100	AD_PRES	90
101	AD_VP	90
...		
200	AD_ASST	10
200	AD_ASST	90
200	AC_ACCOUNT	90
...		
205	AC_MGR	110
206	AC_ACCOUNT	110

30 rows selected.

ORACLE

15-11

Copyright © Oracle Corporation, 2001. All rights reserved.

The UNION ALL Operator (continued)

In the example, 30 rows are selected. The combination of the two tables totals to 30 rows. The **UNION ALL operator** does not eliminate **duplicate rows**. The duplicate rows are highlighted in the output shown in the slide. UNION returns all distinct rows selected by either query. UNION ALL returns all rows selected by either query, including all duplicates. Consider the query on the slide, now written with the **UNION clause**:

```
SELECT employee_id, job_id, department_id
FROM employees
UNION
SELECT employee_id, job_id, department_id
FROM job_history
ORDER BY employee_id;
```

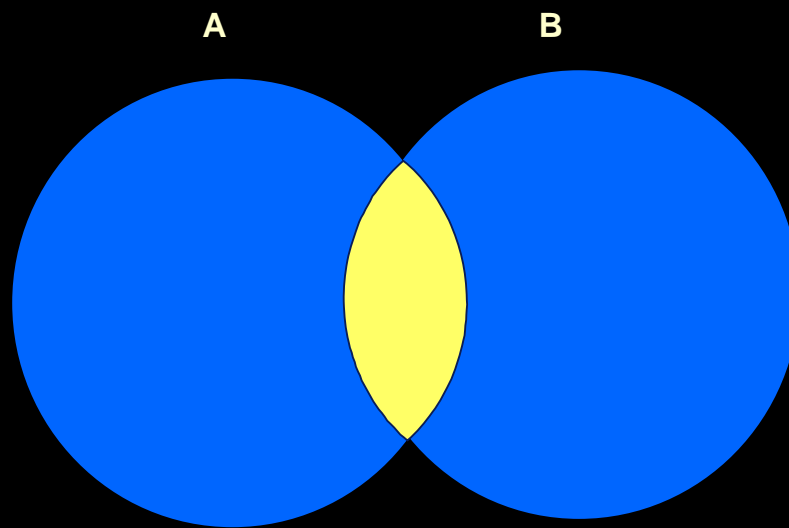
The preceding query returns 29 rows. This is because it eliminates the following row (as it is a duplicate):

EMPLOYEE_ID	JOB_ID	DEPARTMENT_ID
176	SA_REP	80

Instructor Note

Note that this is the example from page 15-8.

The INTERSECT Operator



ORACLE

15-12

Copyright © Oracle Corporation, 2001. All rights reserved.

The INTERSECT Operator

Use the **INTERSECT** operator to return all rows common to multiple queries.

Guidelines

- The number of columns and the datatypes 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 need not be identical.
- Reversing the order of the intersected tables does not alter the result.
- **INTERSECT** does not ignore **NULL** values.

Instructor Note

To illustrate the **INTERSECT SET** operator, run the script `demo\15_inters.sql`.

Using the INTERSECT Operator

Display the employee IDs and job IDs of employees who currently have a job title that they held before beginning their tenure with the company.

```
SELECT employee_id, job_id
FROM employees
INTERSECT
SELECT employee_id, job_id
FROM job_history;
```

EMPLOYEE_ID	JOB_ID
176	SA_REP
200	AD_ASST

ORACLE

15-13

Copyright © Oracle Corporation, 2001. All rights reserved.

The INTERSECT Operator (continued)

In the example in this slide, the query returns only the records that have the same values in the selected columns in both tables.

What will be the results if you add the DEPARTMENT_ID column to the SELECT statement from the EMPLOYEES table and add the DEPARTMENT_ID column to the SELECT statement from the JOB_HISTORY table and run this query? The results may be different because of the introduction of another column whose values may or may not be duplicates.

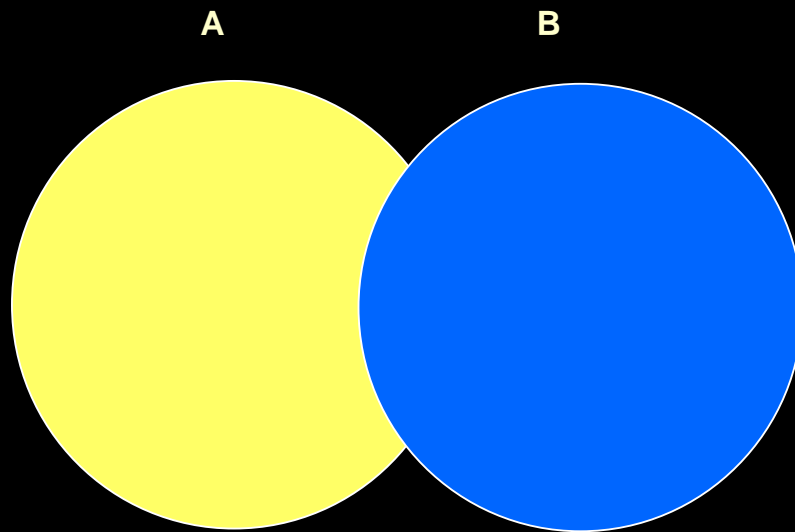
Example

```
SELECT employee_id, job_id, department_id
FROM employees
INTERSECT
SELECT employee_id, job_id, department_id
FROM job_history;
```

EMPLOYEE_ID	JOB_ID	DEPARTMENT_ID
176	SA_REP	80

Employee 200 is no longer part of the results because the EMPLOYEES.DEPARTMENT_ID value is different from the JOB_HISTORY.DEPARTMENT_ID value.

The MINUS Operator



ORACLE

15-14

Copyright © Oracle Corporation, 2001. All rights reserved.

The MINUS Operator

Use the **MINUS** operator to return rows returned by the first query that are not present in the second query (the first SELECT statement MINUS the second SELECT statement).

Guidelines

- The number of columns and the datatypes 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 need not be identical.
- All of the columns in the WHERE clause must be in the SELECT clause for the MINUS operator to work.

Instructor Note

To illustrate the MINUS operator, run the script `demo\15_minus.sql`.

The MINUS Operator

Display the employee IDs of those employees who have not changed their jobs even once.

```
SELECT employee_id, job_id
FROM employees
MINUS
SELECT employee_id, job_id
FROM job_history;
```

EMPLOYEE_ID	JOB_ID
100	AD_PRES
101	AD_VP
102	AD_VP
103	IT_PROG
...	
201	MK_MAN
202	MK_REP
205	AC_MGR
206	AC_ACCOUNT

18 rows selected.

ORACLE

15-15

Copyright © Oracle Corporation, 2001. All rights reserved.

The MINUS Operator (continued)

In the example in the slide, the employee IDs and Job IDs in the JOB_HISTORY 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 JOB_HISTORY table. These are the records of the employees who have not changed their jobs even once.

SET Operator Guidelines

- The expressions in the **SELECT** lists must match in number and data type.
- Parentheses can be used to alter the sequence of execution.
- The **ORDER BY** clause:
 - Can appear only at the very end of the statement
 - Will accept the column name, aliases from the first **SELECT** statement, or the positional notation

ORACLE

15-16

Copyright © Oracle Corporation, 2001. All rights reserved.

SET Operator Guidelines

- The expressions in the select lists of the queries must match in number and datatype. Queries that use UNION, UNION ALL, INTERSECT, and MINUS SET operators in their WHERE clause must have the same number and type of columns in their SELECT list. For example:

```
SELECT employee_id, department_id
FROM   employees
WHERE  (employee_id, department_id)
       IN (SELECT employee_id, department_id
           FROM   employees
           UNION
           SELECT employee_id, department_id
           FROM   job_history);
```
- The **ORDER BY** clause:
 - Can appear only at the very end of the statement
 - Will accept the column name, an alias, or the positional notation
- The column name or alias, if used in an **ORDER BY** clause, must be from the first **SELECT** list.
- SET operators can be used in subqueries.

Instructor Note

You might want to mention that the **ORDER BY** clause accepts the column name only if the column has the same name from both queries.

The 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`.

ORACLE

15-17

Copyright © Oracle Corporation, 2001. All rights reserved.

The Oracle Server and SET Operators

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 datatype. If component queries select character data, the datatype of the return values are determined as follows:

- If both queries select values of datatype `CHAR`, the returned values have datatype `CHAR`.
- If either or both of the queries select values of datatype `VARCHAR2`, the returned values have datatype `VARCHAR2`.

Instructor Note

You might want to mention that the output is sorted in ascending order of the first column, then the second column, and so on, of the `SELECT` clause.

Introduction to Oracle9i: SQL 15-17

Matching the SELECT Statements

Using the UNION operator, display the department ID, location, and hire date for all employees.

```
SELECT department_id, TO_NUMBER(null)
      location, hire_date
FROM   employees
UNION
SELECT department_id, location_id,  TO_DATE(null)
FROM   departments;
```

DEPARTMENT_ID	LOCATION	HIRE_DATE
10	1700	
10		17-SEP-87
20	1800	
20		17-FEB-96
...		
110	1700	
110		07-JUN-94
190	1700	
		24-MAY-99

27 rows selected.

ORACLE

15-18

Copyright © Oracle Corporation, 2001. All rights reserved.

Matching the SELECT Statements

As the expressions in the select lists of the queries must match in number, you can use dummy columns and the datatype conversion functions to comply with this rule. In the slide, the name `location` is given as the dummy column heading. The `TO_NUMBER` function is used in the first query to match the `NUMBER` datatype of the `LOCATION_ID` column retrieved by the second query. Similarly, the `TO_DATE` function in the second query is used to match the `DATE` datatype of the `HIRE_DATE` column retrieved by the first query.

Instructor Note

Demonstration: `demo\15_union3.sql`, `demo\15_dummy.sql`

Purpose: The demonstration `15_union3.sql` illustrates using conversion functions while matching columns in the two select lists. The demonstration `15_dummy.sql` uses dummy columns in order to match the select lists. For the `15_dummy.sql`, run the script, then uncomment the `REMARKS`, add `ORDER BY 2`, and rerun.

You might want to mention that the conversion functions in the code shown on the slide are not mandatory. The code will work fine even without the conversion functions, but it is recommended to explicitly convert values for performance benefits.

Matching the SELECT Statement

- Using the UNION operator, display the employee ID, job ID, and salary of all employees.

```
SELECT employee_id, job_id,salary
FROM   employees
UNION
SELECT employee_id, job_id,0
FROM   job_history;
```

EMPLOYEE_ID	JOB_ID	SALARY
100	AD PRES	24000
101	AC_ACCOUNT	0
101	AC MGR	0
...		
205	AC_MGR	12000
206	AC_ACCOUNT	8300

30 rows selected.

ORACLE

Matching the SELECT Statement: Example

The EMPLOYEES and JOB_HISTORY 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 EMPLOYEE_ID, JOB_ID, and SALARY using the UNION operator, knowing that the salary exists only in the, EMPLOYEES table?

The code example in the slide matches the EMPLOYEE_ID and the JOB_ID columns in the EMPLOYEES and in the JOB_HISTORY tables. A literal value of 0 is added to the JOB_HISTORY SELECT statement to match the numeric SALARY column in the EMPLOYEES SELECT statement.

In the preceding results, each row in the output that corresponds to a record from the JOB_HISTORY table contains a 0 in the SALARY column.

Controlling the Order of Rows

Produce an English sentence using two UNION operators.

```
COLUMN a_dummy NOPRINT
SELECT 'sing' AS "My dream", 3 a_dummy
FROM dual
UNION
SELECT 'I'd like to teach', 1
FROM dual
UNION
SELECT 'the world to', 2
FROM dual
ORDER BY 2;
```

My dream
I'd like to teach
the world to
sing

ORACLE

15-20

Copyright © Oracle Corporation, 2001. All rights reserved.

Controlling the Order of Rows

By default, the output is **sorted** in ascending order on the first column. You can use the ORDER BY clause to change this.

Using ORDER BY to Order Rows

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, an alias, or the positional notation. Without the ORDER BY clause, the code example in the slide produces the following output in the alphabetical order of the first column:

My dream
I'd like to teach
sing
the world to

Note: Consider a compound query where the UNION SET operator is used more than once. In this case, the ORDER BY clause can use only positions rather than explicit expressions.

Instructor Note

To illustrate the ordering of rows with a SET operator, run the script `demo\15_setord.sql`. Briefly explain the COLUMN command with the NOPRINT option. Highlight the usage of the single quotes in the 'I'd like to teach' literal, in the second SELECT statement. You might want to mention that one can only use ORDER BY with a column, alias, or position of the column of the first query.

Introduction to Oracle9i: SQL 15-20

Summary

In this lesson, you should have learned how to:

- **Use `UNION` to return all distinct rows**
- **Use `UNION ALL` to returns all rows, including duplicates**
- **Use `INTERSECT` to return all rows shared by both queries**
- **Use `MINUS` to return all distinct rows selected by the first query but not by the second**
- **Use `ORDER BY` only at the very end of the statement**

ORACLE

15-21

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary

- The `UNION` operator returns all rows selected by either 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 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 datatype.

Practice 15 Overview

This practice covers using the Oracle9i datetime functions.

ORACLE

15-22

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 15 Overview

In this practice, you write queries using the SET operators.

Practice 15

1. List the department IDs for departments that do not contain the job ID ST_CLERK, using SET operators.

DEPARTMENT_ID
10
20
60
80
90
110
190

7 rows selected.

2. Display the country ID and the name of the countries that have no departments located in them, using SET operators.

CO	COUNTRY_NAME
DE	Germany

3. Produce a list of jobs for departments 10, 50, and 20, in that order. Display job ID and department ID, using SET operators.

JOB_ID	DEPARTMENT_ID
AD_ASST	10
ST_CLERK	50
ST_MAN	50
MK_MAN	20
MK_REP	20

4. List the employee IDs and job IDs of those employees who currently hold the job title that they held before beginning their tenure with the company.

EMPLOYEE_ID	JOB_ID
176	SA_REP
200	AD_ASST

Practice 15 (continued)

5. Write a compound query that lists the following:

- Last names and department ID of all the employees from the EMPLOYEES table, regardless of whether or not they belong to any department or not
- Department ID and department name of all the departments from the DEPARTMENTS table, regardless of whether or not they have employees working in them

LAST_NAME	DEPARTMENT_ID	TO_CHAR(NULL)
Abel	80	
Davies	50	
De Haan	90	
Ernst	60	
Fay	20	
Gietz	110	
Grant		
Hartstein	20	
Higgins	110	
Hunold	60	
King	90	
LAST_NAME	DEPARTMENT_ID	TO_CHAR(NULL)
Mourgos	50	
Rajs	50	
Taylor	80	
Vargas	50	
Whalen	10	
Zlotkey	80	
	10	Administration
	20	Marketing
	50	Shipping
	60	IT
	80	Sales
	90	Executive
	110	Accounting
	190	Contracting

28 rows selected.

16

Oracle9i Datetime Functions

ORACLE

Copyright © Oracle Corporation, 2001. All rights reserved.

Schedule:	Timing	Topic
	30 minutes	Lecture
	20 minutes	Practice
	50 minutes	Total

Objectives

After completing this lesson, you should be able use the following datetime functions:

- TZ_OFFSET
- CURRENT_DATE
- CURRENT_TIMESTAMP
- LOCALTIMESTAMP
- DBTIMEZONE
- SESSIONTIMEZONE
- EXTRACT
- FROM_TZ
- TO_TIMESTAMP
- TO_TIMESTAMP_TZ
- TO_YMINTERVAL

ORACLE

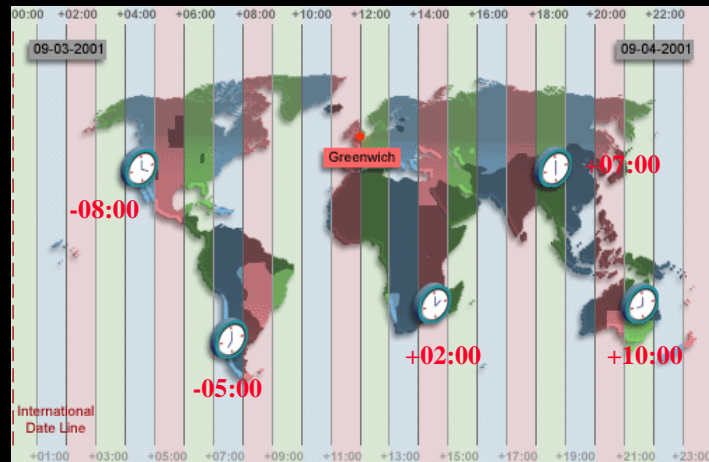
16-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Lesson Aim

This lesson addresses some of the **datetime functions** introduced in Oracle9i.

TIME ZONES



The image represents the time for each time zone when Greenwich time is 12:00.

ORACLE

16-3

Copyright © Oracle Corporation, 2001. All rights reserved.

Time Zones

In Oracle9i, you can include the **time zone** in your date and time data, as well as provide support for fractional seconds. This lesson focuses on how to manipulate the new datetime data types included with Oracle9i using the new datetime functions. To understand the working of these functions, it is necessary to be familiar with the concept of time zones and **Greenwich Mean Time**, or GMT. **Greenwich Mean Time**, or GMT is now referred to as UTC (Coordinated Universal Time).

The hours of the day are measured by the turning of the earth. 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, or GMT. GMT 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 effected by summer time or daylight savings 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 Greenwich mean time (GMT) 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).

Daylight Saving Time

Most western nations advance the clock ahead one hour during the summer months. This period is called daylight saving time. Daylight saving time lasts from the first Sunday in April to the last Sunday in October in the 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.

Introduction to Oracle9i: SQL 16-3

Oracle9i Datetime Support

- In Oracle9i, you can include the time zone in your date and time data, and provide support for fractional seconds.
- Three new data types are added to DATE:
 - `TIMESTAMP`
 - `TIMESTAMP WITH TIME ZONE (TSTZ)`
 - `TIMESTAMP WITH LOCAL TIME ZONE (TSLTZ)`
- Oracle9i provides daylight savings support for datetime data types in the server.

ORACLE

16-4

Copyright © Oracle Corporation, 2001. All rights reserved.

Oracle9i Datetime Support

With Oracle9i, three new data types are added to DATE, with the following differences:

Data Type	Time Zone	Fractional Seconds
DATE	No	No
TIMESTAMP	No	Yes
TIMESTAMP (<i>fractional_seconds_precision</i>) WITH TIMEZONE	All values of TIMESTAMP as well as the time zone displacement value which indicates the hours and minutes before or after UTC (Coordinated Universal Time, formerly Greenwich mean time).	<i>fractional_seconds_precision</i> is the number of digits in the fractional part of the SECOND datetime field. Accepted values are 0 to 9. The default is 6.
TIMESTAMP (<i>fractional_seconds_precision</i>) WITH LOCAL TIME ZONE	All values of TIMESTAMP WITH TIME ZONE, with the following exceptions: <ul style="list-style-type: none">• 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.	Yes

Oracle9i Datetime Support (continued)

TIMESTAMP WITH LOCAL TIME ZONE is stored in the database time zone. When a user selects the data, the value is adjusted to the user's session time zone.

Example:

A San Francisco database has system time zone = -8:00. When a New York client (session time zone = -5:00) inserts into or selects from the San Francisco database, **TIMESTAMP WITH LOCAL TIME ZONE** data is adjusted as follows:

- The New York client inserts **TIMESTAMP '1998-1-23 6:00:00-5:00'** into a **TIMESTAMP WITH LOCAL TIME ZONE** column in the San Francisco database. The inserted data is stored in San Francisco as binary value 1998-1-23 3:00:00.
- When the New York client selects that inserted data from the San Francisco database, the value displayed in New York is '1998-1-23 6:00:00'.
- A San Francisco client, selecting the same data, see the value '1998-1-23 3:00:00'.

Support for Daylight Savings Times

The Oracle Server automatically determines, for any given time zone region, whether **daylight savings** is in effect and returns local time values based accordingly. The datetime value is sufficient for the server to determine whether daylight savings time is in effect for a given region in all cases except boundary cases. A boundary case occurs during the period when daylight savings goes into or comes out of effect. For example, in the U.S.-Pacific region, when daylight savings comes into effect, the time changes from 2:00 a.m. to 3:00 a.m. The one hour interval between 2:00 a.m. and 3:00 a.m. does not exist. When daylight savings goes out of effect, the time changes from 2:00 a.m. back to 1:00 a.m., and the one-hour interval between 1:00 a.m. and 2:00 a.m. is repeated.

Oracle9i also significantly reduces the cost of developing and deploying applications globally on a single database instance. Requirements for multigeographic applications include named time zones and multilanguage support through Unicode. The datetime data types TSLTZ and TSTZ are time-zone-aware. Datetime values can be specified as local time in a particular region (rather than a particular offset). Using the time zone rules tables for a given region, the time zone offset for a local time is calculated, taking into consideration daylight savings time adjustments, and used in further operations.

This lesson addresses some of the new datetime functions introduced in Oracle9i.

TZ_OFFSET

- Display the time zone offset for the time zone 'US/Eastern'

```
SELECT TZ_OFFSET('US/Eastern') FROM DUAL;
```

TZ_OFFSET
-04:00

- Display the time zone offset for the time zone 'Canada/Yukon'

```
SELECT TZ_OFFSET('Canada/Yukon') FROM DUAL;
```

TZ_OFFSET
-07:00

- Display the time zone offset for the time zone 'Europe/London'

```
SELECT TZ_OFFSET('Europe/London') FROM DUAL;
```

TZ_OFFSET
+01:00

ORACLE

16-6

Copyright © Oracle Corporation, 2001. All rights reserved.

TZ_OFFSET

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, the return value can be interpreted as the time zone from where the command was executed is eight hours after 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:mm' ]  
          [ SESSIONTIMEZONE ] [ DBTIMEZONE ] )
```

The examples in the slide can be interpreted as follows:

- The time zone 'US/Eastern' is four hours behind UTC
- The time zone 'Canada/Yukon' is seven hours behind UTC
- The time zone 'Europe/London' is one hour ahead of UTC

For a listing of valid time zone name values, query the V\$TIMEZONE_NAMES dynamic performance view.

```
DESC V$TIMEZONE_NAMES
```

Name	Null?	Type
TZNAME		VARCHAR2(64)
TZABBREV		VARCHAR2(64)

TZ_OFFSET (continued)

```
SELECT * FROM V$TIMEZONE_NAMES;
```

TZNAME	TZABBREV
Africa/Cairo	LMT
Africa/Cairo	EET
Africa/Cairo	EEST
Africa/Tripoli	LMT
Africa/Tripoli	CET
Africa/Tripoli	CEST
Africa/Tripoli	EET
America/Adak	LMT
America/Adak	NST
America/Adak	NWT
America/Adak	BST
America/Adak	BDT
America/Adak	HAST
America/Adak	HADT
TZNAME	TZABBREV
America/Anchorage	LMT
America/Anchorage	CAT
America/Anchorage	CAWT
America/Anchorage	AHST
America/Anchorage	AHDT
America/Anchorage	AKST
■ ■ ■	
W-SU	MDST
W-SU	S
W-SU	MSD
W-SU	MSK
W-SU	EET
W-SU	EEST
WET	WEST
WET	WET

616 rows selected.

CURRENT_DATE

- Display the current date and time in the session's time zone .

```
ALTER SESSION
SET NLS_DATE_FORMAT = 'DD-MON-YYYY HH24:MI:SS';
```

```
ALTER SESSION SET TIME_ZONE = '-5:0';
SELECT SESSIONTIMEZONE, CURRENT_DATE FROM DUAL;
```

SESSIONTIMEZONE	CURRENT_DATE
-05:00	03-OCT-2001 09:37:06

```
ALTER SESSION SET TIME_ZONE = '-8:0';
SELECT SESSIONTIMEZONE, CURRENT_DATE FROM DUAL;
```

SESSIONTIMEZONE	CURRENT_DATE
-08:00	03-OCT-2001 06:38:07

- CURRENT_DATE is sensitive to the session time zone.
- The return value is a date in the Gregorian calendar.

ORACLE

16-8

Copyright © Oracle Corporation, 2001. All rights reserved.

CURRENT_DATE

The **CURRENT_DATE** function returns the current date in the session's time zone. The return value is a date in the Gregorian calendar.

The examples in the slide illustrate that CURRENT_DATE is sensitive to the session time zone. In the first example, the session is altered to set the TIME_ZONE parameter to -5:0. The TIME_ZONE parameter specifies the default local time zone displacement for the current SQL session. TIME_ZONE is a session parameter only, 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 (Coordinated Universal Time, formerly known as Greenwich mean time).

Observe in the output that the value of CURRENT_DATE changes when the TIME_ZONE parameter value is changed to -8:0 in the second example.

Note: 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).

Instructor Note

You might also want to select the SYSDATE for each TIME_ZONE and draw the attention of the students to the fact that SYSDATE remains the same irrespective of the change in the TIME_ZONE . SYSDATE is not sensitive to the session's time zone.

CURRENT_TIMESTAMP

- Display the current date and fractional time in the session's time zone.

```
ALTER SESSION SET TIME_ZONE = '-5:0';  
SELECT SESSIONTIMEZONE, CURRENT_TIMESTAMP  
FROM DUAL;
```

SESSIONTIMEZONE	CURRENT_TIMESTAMP
-05:00	03-OCT-01 09.40.59.000000 AM -05:00

```
ALTER SESSION SET TIME_ZONE = '-8:0';  
SELECT SESSIONTIMEZONE, CURRENT_TIMESTAMP  
FROM DUAL;
```

SESSIONTIMEZONE	CURRENT_TIMESTAMP
-08:00	03-OCT-01 06.41.38.000000 AM -08:00

- **CURRENT_TIMESTAMP** is sensitive to the session time zone.
- The return value is of the **TIMESTAMP WITH TIME ZONE** datatype.

ORACLE

16-9

Copyright © Oracle Corporation, 2001. All rights reserved.

CURRENT_TIMESTAMP

The **CURRENT_TIMESTAMP** function returns the current date and time in the session time zone, as a value of the data type **TIMESTAMP WITH TIME ZONE**. The time zone displacement reflects the current local time of the SQL session. The syntax of the **CURRENT_TIMESTAMP** function is:

CURRENT_TIMESTAMP (*precision*)

where *precision* is an optional argument that specifies the fractional second precision of the time value returned. If you omit precision, the default is 6.

The examples in the slide illustrates that **CURRENT_TIMESTAMP** is sensitive to the session time zone. In the first example, the session is altered to set the **TIME_ZONE** parameter to **-5:0**. Observe in the output that the value of **CURRENT_TIMESTAMP** changes when the **TIME_ZONE** parameter value is changed to **-8:0** in the second example.

LOCALTIMESTAMP

- Display the current date and time in the session time zone in a value of **TIMESTAMP** data type.

```
ALTER SESSION SET TIME_ZONE = '-5:0';  
SELECT CURRENT_TIMESTAMP, LOCALTIMESTAMP  
FROM DUAL;
```

CURRENT_TIMESTAMP	LOCALTIMESTAMP
03-OCT-01 09.44.21.000000 AM -05:00	03-OCT-01 09.44.21.000000 AM

```
ALTER SESSION SET TIME_ZONE = '-8:0';  
SELECT CURRENT_TIMESTAMP, LOCALTIMESTAMP  
FROM DUAL;
```

CURRENT_TIMESTAMP	LOCALTIMESTAMP
03-OCT-01 06.45.21.000001 AM -08:00	03-OCT-01 06.45.21.000001 AM

- **LOCALTIMESTAMP** returns a **TIMESTAMP** value, whereas **CURRENT_TIMESTAMP** returns a **TIMESTAMP WITH TIME ZONE** value.

ORACLE

16-10

Copyright © Oracle Corporation, 2001. All rights reserved.

LOCALTIMESTAMP

The **LOCALTIMESTAMP** function returns the current date and time in the session time zone in a value of data type **TIMESTAMP**. The difference between this function and **CURRENT_TIMESTAMP** is that **LOCALTIMESTAMP** returns a **TIMESTAMP** value, while **CURRENT_TIMESTAMP** returns a **TIMESTAMP WITH TIME ZONE** value. **TIMESTAMP WITH TIME ZONE** is a variant of **TIMESTAMP** that includes a time zone displacement in its value. The time zone displacement is the difference (in hours and minutes) between local time and UTC. The **TIMESTAMP WITH TIME ZONE** data type has the following 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 to 9. The default is 6. For example, you specify **TIMESTAMP WITH TIME ZONE** as a literal as follows:

TIMESTAMP '1997-01-31 09:26:56.66 +02:00'

The syntax of the **LOCAL_TIMESTAMP** function is:

LOCAL_TIMESTAMP (*TIMESTAMP_precision*)

Where, *TIMESTAMP_precision* is an optional argument that specifies the fractional second precision of the **TIMESTAMP** value returned.

The examples in the slide illustrates the difference between **LOCALTIMESTAMP** and **CURRENT_TIMESTAMP**. Observe that the **LOCALTIMESTAMP** does not display the time zone value, while the **CURRENT_TIMESTAMP** does.

Introduction to Oracle9i: SQL 16-10

DBTIMEZONE and SESSIONTIMEZONE

- Display the value of the database time zone.

```
SELECT DBTIMEZONE FROM DUAL;
```

DBTIME
-05:00

- Display the value of the session's time zone.

```
SELECT SESSIONTIMEZONE FROM DUAL;
```

SESSIONTIMEZONE
-08:00

ORACLE

16-11

Copyright © Oracle Corporation, 2001. All rights reserved.

DBTIMEZONE and SESSIONTIMEZONE

The default database time zone is the same as the operating system's time zone. You set 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 can 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 on the slide shows that the database time zone is set to UTC, as 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 set to UTC.

Observe that the database time zone is different from the current session's time zone.

EXTRACT

- Display the YEAR component from the SYSDATE.

```
SELECT EXTRACT (YEAR FROM SYSDATE) FROM DUAL;
```

EXTRACT(YEARFROMSYSDATE)
2001

- Display the MONTH component from the 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)
Kochhar	21-SEP-89	9
De Haan	13-JAN-93	1
Mourgos	16-NOV-99	11
Zlotkey	29-JAN-00	1
Hartstein	17-FEB-96	2

ORACLE

16-12

Copyright © Oracle Corporation, 2001. All rights reserved.

EXTRACT

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 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 [datetime_value_expression]  
     [interval_value_expression]);
```

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 in the **Gregorian calendar**. When extracting from a datetime with a time zone value, the value returned is in UTC. For a listing of time zone names and their corresponding abbreviations, query the **V\$TIMEZONE_NAMES** dynamic performance view. In the first example on the slide, the EXTRACT function is used to extract the YEAR from SYSDATE.

In the second example in the slide, the EXTRACT function is used to extract the MONTH from HIRE_DATE column of the EMPLOYEES table, for those employees who report to the manager whose EMPLOYEE_ID is 100.

Instructor Note

The Oracle Server lets you derive datetime and interval value expressions. Datetime value expressions yield values of datetime data type. Interval value expressions yield values of interval data type. For more information on these data types refer *Oracle9i SQL Reference*.

Introduction to Oracle9i: SQL 16-12

TIMESTAMP Conversion Using FROM_TZ

- Display the **TIMESTAMP** value '2000-03-28 08:00:00' as a **TIMESTAMP WITH TIME ZONE** value.

```
SELECT FROM_TZ(TIMESTAMP
               '2000-03-28 08:00:00', '3:00')
FROM DUAL;
```

```
FROM_TZ(TIMESTAMP'2000-03-2808:00:00','3:00')
28-MAR-00 08.00.00.000000000 AM +03:00
```

- Display the **TIMESTAMP** value '2000-03-28 08:00:00' as a **TIMESTAMP WITH TIME ZONE** value for the time zone region 'Australia/North'

```
SELECT FROM_TZ(TIMESTAMP
               '2000-03-28 08:00:00', 'Australia/North')
FROM DUAL;
```

```
FROM_TZ(TIMESTAMP'2000-03-2808:00:00','AUSTRALIA/NORTH')
28-MAR-00 08.00.00.000000000 AM AUSTRALIA/NORTH
```

ORACLE

16-13

Copyright © Oracle Corporation, 2001. All rights reserved.

TIMESTAMP Conversion Using FROM_TZ

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 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 optional TZD format (TZD is an abbreviated time zone string with daylight savings information.) TZR represents the time zone region in datetime input strings. Examples are 'Australia/North', 'UTC', and 'Singapore'. TZD represents an abbreviated form of the time zone region with daylight savings information. Examples are 'PST' for US/Pacific standard time and 'PDT' for US/Pacific daylight time. To see a listing of valid values for the TZR and TZD format elements, query the V\$TIMEZONE_NAMES dynamic performance view.

The example in the slide converts a **TIMESTAMP** value to **TIMESTAMP WITH TIME ZONE**.

STRING To TIMESTAMP Conversion Using TO_TIMESTAMP and TO_TIMESTAMP_TZ

- Display the character string '2000-12-01 11:00:00' as a **TIMESTAMP** value.

```
SELECT TO_TIMESTAMP ( '2000-12-01 11:00:00',  
                      'YYYY-MM-DD HH:MI:SS' )  
FROM DUAL;
```

```
TO_TIMESTAMP('2000-12-0111:00:00','YYYY-MM-DDHH:MI:SS')  
01-DEC-00 11.00.00.000000000 AM
```

- Display the character string '1999-12-01 11:00:00 -8:00' as a **TIMESTAMP WITH TIME ZONE** value.

```
SELECT  
  TO_TIMESTAMP_TZ( '1999-12-01 11:00:00 -8:00',  
                  'YYYY-MM-DD HH:MI:SS TZH:TZM' )  
FROM DUAL;
```

```
TO_TIMESTAMP_TZ('1999-12-0111:00:00-8:00','YYYY-MM-DDHH:MI:SSTZH:TZM')  
01-DEC-99 11.00.00.000000000 AM -08:00
```

ORACLE

16-14

Copyright © Oracle Corporation, 2001. All rights reserved.

STRING To TIMESTAMP Conversion Using TO_TIMESTAMP and TO_TIMESTAMP_TZ

The **TO_TIMESTAMP** function converts a string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type to a value of **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. This argument can have this form:

```
'NLS_DATE_LANGUAGE = language'
```

If you omit *nlsparams*, this function uses the default date language for your session. The example on the slide converts a character string to a value of **TIMESTAMP**.

The **TO_TIMESTAMP_TZ** function converts a string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type to a value of **TIMESTAMP WITH TIME ZONE** data type. The syntax of the **TO_TIMESTAMP_TZ** function is:

```
TO_TIMESTAMP_TZ ( char , [ fmt ] , [ 'nlsparam' ] )
```

The optional *fmt* specifies the format of *char*. If omitted, a string must be in the default format of the **TIMESTAMP WITH TIME ZONE** data type. The optional *nlsparam* has the same purpose in this function as in the **TO_TIMESTAMP** function. The example in the slide converts a character string to a value of **TIMESTAMP WITH TIME ZONE**.

Note: The **TO_TIMESTAMP_TZ** function does not convert character strings to **TIMESTAMP WITH LOCAL TIME ZONE**.

Time Interval Conversion with TO_YMINTERVAL

- Display a date that is one year two months after the hire date for the employees working in the department with the 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_YMININTERV
17-FEB-1996 00:00:00	17-APR-1997 00:00:00
17-AUG-1997 00:00:00	17-OCT-1998 00:00:00

ORACLE

16-15

Copyright © Oracle Corporation, 2001. All rights reserved.

Time Interval Conversion with TO_YMINTERVAL

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 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 two months after the hire date for the employees working in the department 20 of the EMPLOYEES table.

A reverse calculation can also be done using the TO_YMINTERVAL function. For example:

```
SELECT hire_date, hire_date + TO_YMINTERVAL('-02-04') AS  
       HIRE_DATE_YMINTERVAL  
FROM   employees WHERE department_id = 20;
```

Observe that the character string passed to the TO_YMINTERVAL function has a negative value. The example returns a date that is two years and four months before the hire date for the employees working in the department 20 of the EMPLOYEES table.

Introduction to Oracle9i: SQL 16-15

Summary

In this lesson, you should have learned how to use the following functions:

- TZ_OFFSET
- FROM_TZ
- TO_TIMESTAMP
- TO_TIMESTAMP_TZ
- TO_YMINTERVAL
- CURRENT_DATE
- CURRENT_TIMESTAMP
- LOCALTIMESTAMP
- DBTIMEZONE
- SESSIONTIMEZONE
- EXTRACT

ORACLE

16-16

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary

This lesson addressed some of the new datetime functions introduced in Oracle9i.

Practice 16 Overview

This practice covers using the Oracle9i datetime functions.

ORACLE

16-17

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 16 Overview

In this practice, you display time zone offsets, `CURRENT_DATE`, `CURRENT_TIMESTAMP`, and the `LOCALTIMESTAMP`. You also set time zones and use the `EXTRACT` function.

Instructor Note

1. If you have demonstrated the code example: `ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YYYY HH24:MI:SS'`, remember to issue `ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YYYY'` before moving on to the next lesson.
2. You might want to mention that the results of the questions are based on a different date, and in some cases they will not match the actual results that the students will get. Also, the time zone offset of the various countries might differ based on daylight saving time.

Practice 16

1. Alter the session to set the NLS_DATE_FORMAT to DD-MON-YYYY HH24:MI:SS.
2. a. Write queries to display the time zone offsets (TZ_OFFSET), for the following time zones.

– *US/Pacific-New*

TZ_OFFSET
-07:00

– *Singapore*

TZ_OFFSET
+08:00

– *Egypt*

TZ_OFFSET
+02:00

- b. Alter the session to set the TIME_ZONE parameter value to the time zone offset of US/Pacific-New.
- c. Display the CURRENT_DATE, CURRENT_TIMESTAMP, and LOCALTIMESTAMP for this session.

Note: The output might be different based on the date when the command is executed.

CURRENT_DATE	CURRENT_TIMESTAMP	LOCALTIMESTAMP
01-OCT-2001 13:40:54	01-OCT-01 01.40.54.000001 PM -07:00	01-OCT-01 01.40.54.000001 PM

- d. Alter the session to set the TIME_ZONE parameter value to the time zone offset of Singapore.
- e. Display the CURRENT_DATE, CURRENT_TIMESTAMP, and LOCALTIMESTAMP for this session. Note: The output might be different based on the date when the command is executed.

CURRENT_DATE	CURRENT_TIMESTAMP	LOCALTIMESTAMP
02-OCT-2001 04:42:34	02-OCT-01 04.42.34.000000 AM +08:00	02-OCT-01 04.42.34.000000 AM

Note: Observe in the preceding practice that CURRENT_DATE, CURRENT_TIMESTAMP, and LOCALTIMESTAMP are all sensitive to the session time zone.

3. Write a query to display the DBTIMEZONE and SESSIONTIMEZONE.

DBTIMEZONE	SESSIONTIMEZONE
-05:00	+08:00

Practice 16 (continued)

- Write a query to extract the YEAR from HIRE_DATE column of the EMPLOYEES table for those employees who work in department 80.

LAST_NAME	EXTRACT(YEARFROMHIRE_DATE)
Zlotkey	2000
Abel	1996
Taylor	1998

- Alter the session to set the NLS_DATE_FORMAT to DD-MON-YYYY.

17

Enhancements to the GROUP BY Clause

ORACLE

Copyright © Oracle Corporation, 2001. All rights reserved.

Schedule:

Timing

45 minutes

30 minutes

75 minutes

Topic

Lecture

Practice

Total

Objectives

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

- Use the `ROLLUP` operation to produce subtotal values
- Use the `CUBE` operation to produce cross-tabulation values
- Use the `GROUPING` function to identify the row values created by `ROLLUP` or `CUBE`
- Use `GROUPING SETS` to produce a single result set

ORACLE

17-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Lesson Aim

In this lesson you learn how to:

- **Group data** for obtaining the following:
 - Subtotal values by using the `ROLLUP` operator
 - Cross-tabulation values by using the `CUBE` operator
- Use the **`GROUPING` function** to identify the level of aggregation in the results set produced by a **`ROLLUP`** or **`CUBE` operator**.
- Use **`GROUPING SETS`** to produce a single result set that is equivalent to a `UNION ALL` approach.

Review of Group Functions

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

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

Example:

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

ORACLE

17-3

Copyright © Oracle Corporation, 2001. All rights reserved.

Group Functions

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. Group functions can appear in select lists and in ORDER BY and HAVING clauses. The Oracle Server applies the group functions to each group of rows and returns a single result row for each group.

Types of Group Functions

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

Guidelines for Using Group Functions

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

Instructor Note

You can skip this slide if the students are already familiar with these concepts.

Introduction to Oracle9i: SQL 17-3

Review of the GROUP BY Clause

Syntax:

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

Example:

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

ORACLE

17-4

Copyright © Oracle Corporation, 2001. All rights reserved.

Review of GROUP BY Clause

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

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

DEPARTMENT_ID	JOB_ID	SUM(SALARY)	COUNT(EMPLOYEE_ID)
10	AD_ASST	4400	1
20	MK_MAN	13000	1
20	MK_REP	6000	1
50	ST_CLERK	11700	4
...			
110	AC_ACCOUNT	8300	1
110	AC_MGR	12000	1
	SA_REP	7000	1

13 rows selected.

Review of the HAVING Clause

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

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

ORACLE

17-5

Copyright © Oracle Corporation, 2001. All rights reserved.

The HAVING Clause

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

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

1. Groups rows
2. Applies the group functions to the groups and displays the groups that match the criteria in the **HAVING** clause

```
SELECT department_id, AVG(salary)  
FROM   employees  
GROUP BY department_id  
HAVING AVG(salary) >9500;
```

DEPARTMENT_ID	AVG(SALARY)
80	10033.3333
90	19333.3333
110	10150

The example displays department ID and average salary for those departments whose average salary is greater than \$9,500.

GROUP BY with ROLLUP and CUBE Operators

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

ORACLE

17-6

Copyright © Oracle Corporation, 2001. All rights reserved.

GROUP BY with the ROLLUP and CUBE Operators

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

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

The ROLLUP and CUBE operators are available only in Oracle8i and later releases.

ROLLUP Operator

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

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

ORACLE

17-7

Copyright © Oracle Corporation, 2001. All rights reserved.

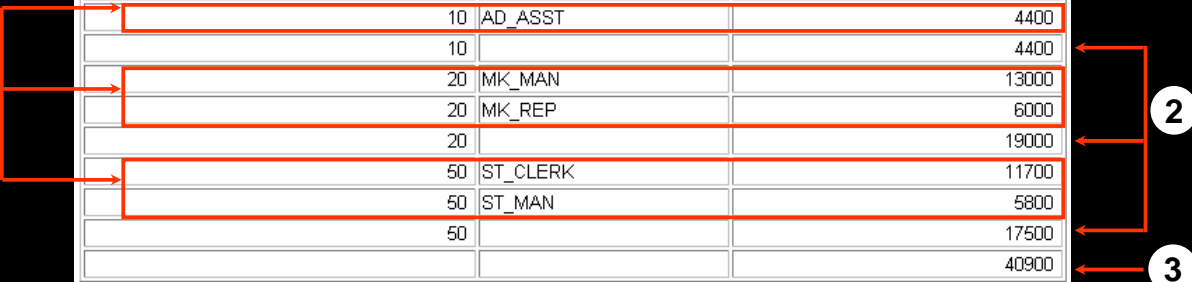
The ROLLUP Operator

The **ROLLUP** operator delivers aggregates and **superaggregates** for expressions within a GROUP BY statement. The ROLLUP operator can be used by report writers to extract statistics and summary information from results sets. The cumulative aggregates can be used in reports, charts, and graphs. The ROLLUP operator creates groupings by moving in one direction, from right to left, along the list of columns specified in the GROUP BY clause. It then applies the aggregate function to these groupings.

Note: To produce subtotals in n dimensions (that is, n columns in the GROUP BY clause) without a ROLLUP operator, $n+1$ SELECT statements must be linked with UNION ALL. This makes the query execution inefficient, because each of the SELECT statements causes table access. The ROLLUP operator gathers its results with just one table access. The ROLLUP operator is useful if there are many columns involved in producing the subtotals.

ROLLUP Operator Example

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



DEPARTMENT_ID	JOB_ID	SUM(SALARY)
10	AD_ASST	4400
10		4400
20	MK_MAN	13000
20	MK_REP	6000
20		19000
50	ST_CLERK	11700
50	ST_MAN	5800
50		17500
		40900

9 rows selected.

ORACLE

17-8

Copyright © Oracle Corporation, 2001. All rights reserved.

Example of a ROLLUP Operator

In the example in the slide:

- Total salaries for every job ID within a department for those departments whose department ID is less than 60 are displayed by the GROUP BY clause (labeled 1)
- The ROLLUP operator displays:
 - Total salary for those departments whose department ID is less than 60 (labeled 2)
 - Total salary for all departments whose department ID is less than 60, irrespective of the job IDs (labeled 3)
- All rows indicated as 1 are regular rows and all rows indicated as 2 and 3 are **superaggregate rows**.

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

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

CUBE Operator

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

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

ORACLE

17-9

Copyright © Oracle Corporation, 2001. All rights reserved.

The CUBE Operator

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

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

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

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

CUBE Operator: Example

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

DEPARTMENT_ID	JOB_ID	SUM(SALARY)
10	AD_ASST	4400
10		4400
20	MK_MAN	13000
20	MK_REP	6000
20		19000
50	ST_CLERK	11700
50	ST_MAN	5800
50		17500
	AD_ASST	4400
	MK_MAN	13000
	MK_REP	6000
	ST_CLERK	11700
	ST_MAN	5800
		40900

14 rows selected.

ORACLE

17-10

Copyright © Oracle Corporation, 2001. All rights reserved.

Example of a CUBE Operator

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

- The total salary for every job within a department (for those departments whose department ID is less than 60) is displayed by the GROUP BY clause (labeled 1)
- The total salary for those departments whose department ID is less than 60 (labeled 2)
- The total salary for every job irrespective of the department (labeled 3)
- Total salary for those departments whose department ID is less than 60, irrespective of the job titles (labeled 4)

In the preceding example, all rows indicated as 1 are regular rows, all rows indicated as 2 and 4 are superaggregate rows, and all rows indicated as 3 are **cross-tabulation values**.

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

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

GROUPING Function

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

- The GROUPING function can be used with either the CUBE or ROLLUP operator.
- Using the GROUPING function, you can find the groups forming the subtotal in a row.
- Using the GROUPING function, you can differentiate stored NULL values from NULL values created by ROLLUP or CUBE.
- The GROUPING function returns 0 or 1.

ORACLE

17-11

Copyright © Oracle Corporation, 2001. All rights reserved.

The GROUPING Function

The **GROUPING** function can be used with either the CUBE or ROLLUP operator to help you understand how a summary value has been obtained.

The GROUPING function uses a single column as its argument. The *expr* in the GROUPING function must match one of the expressions in the GROUP BY clause. The function returns a value of 0 or 1.

The values returned by the GROUPING function are useful to:

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

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

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

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

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

GROUPING Function: Example

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

DEPTID	JOB	SUM(SALARY)	GRP_DEPT	GRP_JOB
10	AD_ASST	4400	0	0
10		4400	0	1
20	MK_MAN	13000	0	0
20	MK_REP	6000	0	0
20		19000	0	1
		23400	1	1

6 rows selected.

1

2

3

ORACLE

17-12

Copyright © Oracle Corporation, 2001. All rights reserved.

Example of a GROUPING Function

In the example in the slide, consider the summary value 4400 in the first row (labeled 1). This summary value is the total salary for the job ID of AD_ASST within department 10. To calculate this summary value, both the columns DEPARTMENT_ID and JOB_ID have been taken into account. Thus a value of 0 is returned for both the expressions GROUPING(department_id) and GROUPING(job_id).

Consider the summary value 4400 in the second row (labeled 2). This value is the total salary for department 10 and has been calculated by taking into account the column DEPARTMENT_ID; thus a value of 0 has been returned by GROUPING(department_id). Because the column JOB_ID has not been taken into account to calculate this value, a value of 1 has been returned for GROUPING(job_id). You can observe similar output in the fifth row.

In the last row, consider the summary value 23400 (labeled 3). This is the total salary for those departments whose department ID is less than 50 and all job titles. To calculate this summary value, neither of the columns DEPARTMENT_ID and JOB_ID have been taken into account. Thus a value of 1 is returned for both the expressions GROUPING(department_id) and GROUPING(job_id).

Instructor Note

Explain that if the same example is run with the CUBE operator, it returns a results set that has 1 for GROUPING(department_id) and 0 for GROUPING(job_id) in the cross-tabulation rows, because the subtotal values are the result of grouping on job irrespective of department number.

GROUPING SETS

- **GROUPING SETS** are a further extension of the **GROUP BY** clause.
- You can use **GROUPING SETS** to define multiple groupings in the same query.
- The Oracle Server computes all groupings specified in the **GROUPING SETS** clause and combines the results of individual groupings with a **UNION ALL** operation.
- **Grouping set efficiency:**
 - Only one pass over the base table is required.
 - There is no need to write complex **UNION** statements.
 - The more elements the **GROUPING SETS** have, the greater the performance benefit.

ORACLE

17-13

Copyright © Oracle Corporation, 2001. All rights reserved.

GROUPING SETS

GROUPING SETS are a further extension of the **GROUP BY** clause that let you specify multiple groupings of data. Doing so facilitates efficient aggregation and hence facilitates analysis of data across multiple dimensions.

A single **SELECT** statement can now be written using **GROUPING SETS** to specify various groupings (that can also include **ROLLUP** or **CUBE** operators), rather than multiple **SELECT** statements combined by **UNION ALL** operators. For example, you can say:

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

This statement calculates aggregates over three groupings:

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

Without this enhancement in Oracle9i, multiple queries combined together with **UNION ALL** are required to get the output of the preceding **SELECT** statement. A multiquery approach is inefficient, for it requires multiple scans of the same data.

GROUPING SETS (continued)

Compare the preceding statement with this alternative:

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

The preceding statement computes all the 8 (2 *2 *2) groupings, though only the groups (department_id, job_id, manager_id), (department_id, manager_id) and (job_id, manager_id) are of interest to you.

Another alternative is the following statement:

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

This statement requires three scans of the base table, making it inefficient.

CUBE and ROLLUP can be thought of as grouping sets with very specific semantics. The following equivalencies show this fact:

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

GROUPING SETS: Example

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

DEPARTMENT_ID	JOB_ID	MANAGER_ID	AVG(SALARY)
10	AD_ASST		4400
20	MK_MAN		13000
20	MK_REP		6000
50	ST_CLERK		2925
...			
	SA_MAN	100	10500
	SA_REP	149	8866.66667
	ST_CLERK	124	2925
	ST_MAN	100	5800

26 rows selected.

1

2

ORACLE

17-15

Copyright © Oracle Corporation, 2001. All rights reserved.

GROUPING SETS: Example

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

- Department ID, Job ID
- Job ID, Manager ID

The average salaries for each of these groups are calculated. The results set displays average salary for each of the two groups.

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

- The average salary of all employees with the job ID AD_ASST in the department 10 is 4400.
- The average salary of all employees with the job ID MK_MAN in the department 20 is 13000.
- The average salary of all employees with the job ID MK_REP in the department 20 is 6000.
- The average salary of all employees with the job ID ST_CLERK in the department 50 is 2925 and so on.

GROUPING SETS: Example (continued)

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

- The average salary of all employees with the job ID MK_REP, who report to the manager with the manager ID 201, is 6000.
- The average salary of all employees with the job ID SA_MAN, who report to the manager with the manager ID 100, is 10500, and so on.

The example in the slide can also be written as:

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

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

Composite Columns

- A composite column is a collection of columns that are treated as a unit.
`ROLLUP (a, (b,c), d)`
- To specify composite columns, use the `GROUP BY` clause to group columns within parentheses so that the Oracle server treats them as a unit while computing `ROLLUP` or `CUBE` operations.
- When used with `ROLLUP` or `CUBE`, composite columns would mean skipping aggregation across certain levels.

ORACLE

17-17

Copyright © Oracle Corporation, 2001. All rights reserved.

Composite Columns

A **composite column** is a collection of columns that are treated as a unit during the computation of groupings. You specify the columns in parentheses as in the following statement:

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

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

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

is equivalent to

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

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

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

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

Composite Columns (continued)

Compare this with the normal ROLLUP as in:

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

which would be

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

```
GROUP BY a, b UNION ALL
```

```
GROUP BY a UNION ALL
```

```
GROUP BY ().
```

Similarly,

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

would be equivalent to

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

```
GROUP BY a, b UNION ALL
```

```
GROUP BY c UNION ALL
```

```
GROUP BY ().
```

The following table shows grouping sets specification and equivalent GROUP BY specification.

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

Composite Columns: Example

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

DEPARTMENT_ID	JOB_ID	MANAGER_ID	SUM(SALARY)
10	AD_ASST	101	4400
10			4400
20	MK_MAN	100	13000
20	MK_REP	201	6000
20			19000
50	ST_CLERK	124	11700
...			
			175500

23 rows selected.

ORACLE

17-19

Copyright © Oracle Corporation, 2001. All rights reserved.

Composite Columns: Example

Consider the example:

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

The preceding query results in the Oracle Server computing the following groupings:

1. (department_id, job_id, manager_id)
2. (department_id, job_id)
3. (department_id)
4. ()

If you are just interested in grouping of lines (1), (3), and (4) in the preceding example, you cannot limit the calculation to those groupings without using composite columns. With composite columns, this is possible by treating JOB_ID and MANAGER_ID columns as a single unit while rolling up. Columns enclosed in parentheses are treated as a unit while computing ROLLUP and CUBE. This is illustrated in the example on the slide. By enclosing JOB_ID and MANAGER_ID columns in parenthesis, we indicate to the Oracle Server to treat JOB_ID and MANAGER_ID as a single unit, as a composite column.

Composite Columns: Example (continued)

The example in the slide computes the following groupings:

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

The example in the slide displays the following:

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

The example in the slide can also be written as:

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

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

Concatenated Groupings

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

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

ORACLE

17-21

Copyright © Oracle Corporation, 2001. All rights reserved.

Concatenated Columns

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

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

The preceding SQL defines the following groupings:

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

Concatenation of grouping sets is very helpful for these reasons:

- Ease of query development: you need not manually enumerate all groupings
- Use by applications: SQL generated by OLAP applications often involves concatenation of grouping sets, with each grouping set defining groupings needed for a dimension

Concatenated Groupings Example

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

DEPARTMENT_ID	JOB_ID	MANAGER_ID	SUM(SALARY)
10	AD_ASST	101	4400
20	MK_MAN	100	13000
...			
10		101	4400
20		100	13000
...			
10	AD_ASST		4400
10			4400
...			
	SA_REP		7000
			7000

49 rows selected.

ORACLE

17-22

Copyright © Oracle Corporation, 2001. All rights reserved.

Concatenated Groupings Example

The example in the slide results in the following groupings:

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

The total salary for each of these groups is calculated.

The example in the slide displays the following:

- Total salary for every department, job ID, manager
- Total salary for every department, manager ID
- Total salary for every department, job ID
- Total salary for every department

For easier understanding, the details for the department 10 are highlighted in the output.

Summary

In this lesson, you should have learned how to:

- Use the **ROLLUP** operation to produce subtotal values
- Use the **CUBE** operation to produce cross-tabulation values
- Use the **GROUPING** function to identify the row values created by **ROLLUP** or **CUBE**
- Use the **GROUPING SETS** syntax to define multiple groupings in the same query
- Use the **GROUP BY** clause, to combine expressions in various ways:
 - Composite columns
 - Concatenated grouping sets

ORACLE

17-23

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary

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

Practice 17 Overview

This practice covers the following topics:

- **Using the ROLLUP operator**
- **Using the CUBE operator**
- **Using the GROUPING function**
- **Using GROUPING SETS**

ORACLE

17-24

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 17 Overview

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

Practice 17

1. Write a query to display the following for those employees whose manager ID is less than 120:

- Manager ID
- Job ID and total salary for every job ID for employees who report to the same manager
- Total salary of those managers
- Total salary of those managers, irrespective of the job IDs

MANAGER_ID	JOB_ID	SUM(SALARY)
100	AD_VP	34000
100	MK_MAN	13000
100	SA_MAN	10500
100	ST_MAN	5800
100		63300
101	AC_MGR	12000
101	AD_ASST	4400
101		16400
102	IT_PROG	9000
102		9000
103	IT_PROG	10200
103		10200
		98900

13 rows selected.

Practice 17 (continued)

2. Observe the output from question 1. Write a query using the GROUPING function to determine whether the NULL values in the columns corresponding to the GROUP BY expressions are caused by the ROLLUP operation.

MGR	JOB	SUM(SALARY)	GROUPING(MANAGER_ID)	GROUPING(JOB_ID)
100	AD_VP	34000	0	0
100	MK_MAN	13000	0	0
100	SA_MAN	10500	0	0
100	ST_MAN	5800	0	0
100		63300	0	1
101	AC_MGR	12000	0	0
101	AD_ASST	4400	0	0
101		16400	0	1
102	IT_PROG	9000	0	0
102		9000	0	1
103	IT_PROG	10200	0	0
103		10200	0	1
		98900	1	1

13 rows selected.

Practice 17 (continued)

3. Write a query to display the following for those employees whose manager ID is less than 120:

- Manager ID
- Job and total salaries for every job for employees who report to the same manager
- Total salary of those managers
- Cross-tabulation values to display the total salary for every job, irrespective of the manager
- Total salary irrespective of all job titles

MANAGER_ID	JOB_ID	SUM(SALARY)
100	AD_VP	34000
100	MK_MAN	13000
100	SA_MAN	10500
100	ST_MAN	5800
100		63300
101	AC_MGR	12000
101	AD_ASST	4400
101		16400
102	IT_PROG	9000
102		9000
103	IT_PROG	10200
103		10200
	AC_MGR	12000
	AD_ASST	4400
MANAGER_ID	JOB_ID	SUM(SALARY)
	AD_VP	34000
	IT_PROG	19200
	MK_MAN	13000
	SA_MAN	10500
	ST_MAN	5800
		98900

20 rows selected.

Practice 17 (continued)

- Observe the output from question 3. Write a query using the GROUPING function to determine whether the NULL values in the columns corresponding to the GROUP BY expressions are caused by the CUBE operation.

MGR	JOB	SUM(SALARY)	GROUPING(MANAGER_ID)	GROUPING(JOB_ID)
100	AD_VP	34000	0	0
100	MK_MAN	13000	0	0
100	SA_MAN	10500	0	0
100	ST_MAN	5800	0	0
100		63300	0	1
101	AC_MGR	12000	0	0
101	AD_ASST	4400	0	0
101		16400	0	1
102	IT_PROG	9000	0	0
102		9000	0	1
103	IT_PROG	10200	0	0
103		10200	0	1
	AC_MGR	12000	1	0
	AD_ASST	4400	1	0
MGR	JOB	SUM(SALARY)	GROUPING(MANAGER_ID)	GROUPING(JOB_ID)
	AD_VP	34000	1	0
	IT_PROG	19200	1	0
	MK_MAN	13000	1	0
	SA_MAN	10500	1	0
	ST_MAN	5800	1	0
		98900	1	1

20 rows selected.

Practice 17 (continued)

5. Using GROUPING SETS, write a query to display the following groupings :

- department_id, manager_id, job_id
- department_id, job_id
- manager_id, job_id

The query should calculate the sum of the salaries for each of these groups.

DEPARTMENT_ID	MANAGER_ID	JOB_ID	SUM(SALARY)
10	101	AD_ASST	4400
20	100	MK_MAN	13000
20	201	MK_REP	6000
50	124	ST_CLERK	11700
50	100	ST_MAN	5800
60	102	IT_PROG	9000
60	103	IT_PROG	10200
80	100	SA_MAN	10500
80	149	SA_REP	19600
90		AD_PRES	24000
90	100	AD_VP	34000
110	205	AC_ACCOUNT	8300
110	101	AC_MGR	12000
	149	SA_REP	7000
■ ■ ■			
	100	MK_MAN	13000
	100	SA_MAN	10500
	100	ST_MAN	5800
	101	AC_MGR	12000
	101	AD_ASST	4400
	102	IT_PROG	9000
	103	IT_PROG	10200
	124	ST_CLERK	11700
	149	SA_REP	26600
	201	MK_REP	6000
	205	AC_ACCOUNT	8300
		AD_PRES	24000

40 rows selected.

Instructor Note

Analytical Functions

Oracle8i, release 2 (8.1.6) introduces a set of analytical group functions that provide the use of flexible and powerful calculation expressions. These analytical functions eliminate complex programming outside of standard SQL for calculations such as moving averages, rankings, and lead and lag comparisons.

In Oracle8i, release 2, each group defined with GROUP BY clause in a SELECT statement is called a *partition*. A query result set may have just one partition holding all the rows, a few large partitions, or many small partitions holding just a few rows each. Analytical functions are applied to each row in each partition.

RANK Function

The RANK function produces an ordered ranking of rows starting with a rank of one. Users specify an optional PARTITION clause and a required ORDER BY clause. The PARTITION keyword is used to define where the rank resets. The specific column that is ranked is determined by the ORDER BY clause. If no partition is specified, ranking is performed over the entire result set. RANK assigns a rank of 1 to the smallest value unless descending order is used.

In the following example, the query ranks managers for each department based on the total salary of all employees working under that manager.

```
SELECT  department_id deptno, job_id job, SUM(salary),
        RANK() OVER(PARTITION BY department_id ORDER BY
                     SUM(salary) DESC)
        AS jobdep_rank,
        RANK() OVER(ORDER BY SUM(salary) DESC) AS sumsal_rank
FROM    employees
GROUP BY department_id, job_id;
```

DEPTNO	JOB	SUM(SALARY)	JOBDEP_RANK	SUMSAL_RANK
90	AD_VP	34000	1	1
90	AD PRES	24000	2	2
80	SA_REP	19600	1	3
60	IT_PROG	19200	1	4
20	MK_MAN	13000	1	5
110	AC_MGR	12000	1	6
50	ST_CLERK	11700	1	7
80	SA_MAN	10500	2	8
110	AC_ACCOUNT	8300	2	9
	SA_REP	7000	1	10
20	MK_REP	6000	2	11
50	ST_MAN	5800	2	12
10	AD_ASST	4400	1	13

13 rows selected.

Note: For ranking in groups provided by CUBE and ROLLUP, use GROUPING () flags in the PARTITION BY clause to trigger resetting.

Instructor Note (continued)

CUME_DIST Function

The cumulative distribution function computes the relative position of a value relative to the other values in its group (partition.) The CUME_DIST function defines the fraction of the rows, in the partition of a given row, that come before or are ties with the current value. It returns the results as a decimal value between zero and one, excluding zero and including one. The results of a CUME_DIST function are often called the percentile values. Default order is ascending, meaning that the lowest value in a partition gets the lowest CUME_DIST.

```
SELECT  department_id DEPTNO, job_id JOB, SUM(salary),
        CUME_DIST() OVER(PARTITION BY department_id ORDER BY
                          SUM(salary) DESC)
        AS cume_dist_per_dep
FROM    employees
GROUP BY department_id, job_id
ORDER BY department_id, SUM(salary);
```

DEPTNO	JOB	SUM(SALARY)	CUME_DIST_PER_DEP
10	AD_ASST	4400	1
20	MK_REP	6000	1
20	MK_MAN	13000	.5
50	ST_MAN	5800	1
50	ST_CLERK	11700	.5
60	IT_PROG	19200	1
80	SA_MAN	10500	1
80	SA_REP	19600	.5
90	AD_PRES	24000	1
90	AD_VP	34000	.5
110	AC_ACCOUNT	8300	1
110	AC_MGR	12000	.5
	SA_REP	7000	1

13 rows selected.

PERCENT_RANK Function:

This function returns the rank of a value relative to a group of values. It returns values in the range of zero to one. The formula used by this function is:

$$(\text{rank of row in its partition} - 1) / (\text{number of rows in the partition} - 1)$$

18

Advanced Subqueries

ORACLE

Copyright © Oracle Corporation, 2001. All rights reserved.

Schedule:	Timing	Topic
	60 minutes	Lecture
	50 minutes	Practice
	110 minutes	Total

Objectives

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

- Write a multiple-column subquery
- Describe and explain the behavior of subqueries when null values are retrieved
- Write a subquery in a FROM clause
- Use scalar subqueries in SQL
- Describe the types of problems that can be solved with correlated subqueries
- Write correlated subqueries
- Update and delete rows using correlated subqueries
- Use the EXISTS and NOT EXISTS operators
- Use the WITH clause

ORACLE

18-2

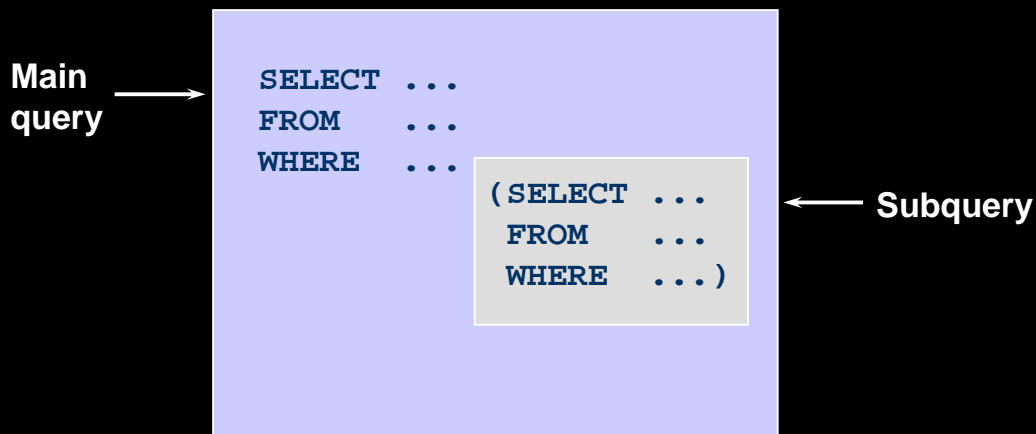
Copyright © Oracle Corporation, 2001. All rights reserved.

Lesson Aim

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 the **WITH clause**.

What Is a Subquery?

A subquery is a **SELECT** statement embedded in a clause of another SQL statement.



ORACLE

18-3

Copyright © Oracle Corporation, 2001. All rights reserved.

What Is a Subquery?

A *subquery* is a **SELECT** statement that is embedded in a clause of another SQL statement, called the parent statement.

The **subquery** (inner query) returns a value that is used by the parent statement. Using a nested subquery is equivalent to performing two sequential queries and using the result of the inner query as the search value in the outer query (main query).

Subqueries can be used for the following purposes:

- To provide values for conditions in **WHERE**, **HAVING**, and **START WITH** clauses of **SELECT** statements
- To define the set of rows to be inserted into the target table of an **INSERT** or **CREATE TABLE** statement
- To define the set of rows to be included in a view or snapshot in a **CREATE VIEW** or **CREATE SNAPSHOT** statement
- To define one or more values to be assigned to existing rows in an **UPDATE** statement
- To define a table to be operated on by a containing query. (You do this by placing the subquery in the **FROM** clause. This can be done in **INSERT**, **UPDATE**, and **DELETE** statements as well.)

Note: A subquery is evaluated once for the entire parent statement.

Instructor Note

You can skip this slide if the students are already familiar with these concepts.

Subqueries

```
SELECT select_list
FROM   table
WHERE  expr operator (SELECT select_list
                        FROM   table);
```

- The subquery (inner query) executes once before the main query.
- The result of the subquery is used by the main query (outer query).

ORACLE

18-4

Copyright © Oracle Corporation, 2001. All rights reserved.

Subqueries

You can build powerful statements out of simple ones by using subqueries. Subqueries 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 or some other table. Subqueries are very useful for writing SQL statements that need values based on one or more unknown conditional values.

In the syntax:

operator includes a comparison operator such as >, =, or IN

Note: **Comparison operators** fall into two classes: single-row operators (>, =, >=, <, <>, <=) and multiple-row operators (IN, ANY, ALL).


The subquery is often referred to as a nested SELECT, sub-SELECT, or inner SELECT statement. The inner and outer queries can retrieve data from either the same table or different tables.

Instructor Note

You can skip this slide if the students are already familiar with these concepts.

Using a Subquery

```
SELECT last_name
FROM   employees
WHERE  salary >
      (SELECT salary
       FROM   employees
       WHERE  employee_id = 149) ;
```

A red arrow points from the value 10500 to the subquery's result, which is then compared against the salary column in the outer query's WHERE clause.

LAST_NAME
King
Kochhar
De Haan
Abel
Hartstein
Higgins

6 rows selected.

ORACLE

18-5

Copyright © Oracle Corporation, 2001. All rights reserved.

Using a Subquery

In the example in the slide, the **inner query** returns the salary of the employee with employee number 149. The **outer query** uses the result of the inner query to display the names of all the employees who earn more than this amount.

Example

Display the names of all employees who earn less than the average salary in the company.

```
SELECT last_name, job_id, salary
FROM   employees
WHERE  salary < (SELECT AVG(salary)
                 FROM   employees) ;
```

Instructor Note

You can skip this slide if the students are already familiar with these concepts.

Multiple-Column Subqueries

Main query

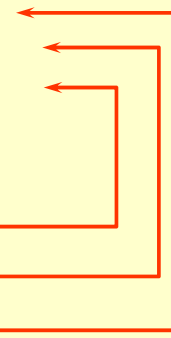
WHERE (MANAGER_ID, DEPARTMENT_ID) IN

Subquery

100 90

102 60

124 50



Each row of the main query is compared to values from a multiple-row and multiple-column subquery.

ORACLE

18-6

Copyright © Oracle Corporation, 2001. All rights reserved.

Multiple-Column Subqueries

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 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 the MANAGER_ID and DEPARTMENT_ID from the main query are being compared with the MANAGER_ID and DEPARTMENT_ID values retrieved by the subquery. Since the number of columns that are being compared are more than one, the example qualifies as a multiple-column subquery.

Column Comparisons

Column comparisons in a multiple-column subquery can be:

- Pairwise comparisons
- Nonpairwise comparisons

ORACLE

18-7

Copyright © Oracle Corporation, 2001. All rights reserved.

Pairwise versus Nonpairwise Comparisons

Column comparisons in a multiple-column subquery can be **pairwise comparisons** or **nonpairwise comparisons**.

In the example on the next slide, a pairwise comparison was executed in the WHERE clause. Each candidate row in the SELECT statement must have *both* the same MANAGER_ID column and the DEPARTMENT_ID as the employee with the EMPLOYEE_ID 178 or 174.

A multiple-column subquery can also be a nonpairwise comparison. In a nonpairwise comparison, each of the columns from the WHERE clause of the parent SELECT statement are individually compared to multiple values retrieved by the inner select statement. The individual columns can match any of the values retrieved by the inner select statement. But collectively, all the multiple conditions of the main SELECT statement must be satisfied for the row to be displayed. The example on the next page illustrates a nonpairwise comparison.

Pairwise Comparison Subquery

Display the details of the employees who are managed by the same manager *and* work in the same department as the employees with `EMPLOYEE_ID` 178 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 (178,174))
AND    employee_id NOT IN (178,174);
```

ORACLE

18-8

Copyright © Oracle Corporation, 2001. All rights reserved.

Pairwise Comparison Subquery

The example in the slide is that of a **multiple-column subquery** because the subquery returns more than one column. 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 the employees with the `EMPLOYEE_ID` 178 or 174.

First, the subquery to retrieve the `MANAGER_ID` and `DEPARTMENT_ID` values for the employees with the `EMPLOYEE_ID` 178 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 the employees with the `EMPLOYEE_ID` 178 or 174 will not be displayed. The output of the query in the slide follows.

EMPLOYEE_ID	MANAGER_ID	DEPARTMENT_ID
176	149	80

Nonpairwise Comparison Subquery

Display the details of the employees who are managed by the same manager as the employees with `EMPLOYEE_ID` 174 or 141 *and* work in the same department as the 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);
```

ORACLE

18-9

Copyright © Oracle Corporation, 2001. All rights reserved.

Nonpairwise Comparison Subquery

The example shows a nonpairwise comparison of the 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` match 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 the employees with the `EMPLOYEE_ID` 174 or 141 is executed. Similarly, the second subquery to retrieve the `DEPARTMENT_ID` values for the employees with the `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` column 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. The output of the query in the slide follows.

EMPLOYEE_ID	MANAGER_ID	DEPARTMENT_ID
142	124	50
143	124	50
144	124	50
176	149	80

Using a Subquery in the FROM Clause

```
SELECT  a.last_name, a.salary,  
        a.department_id, b.salavg  
FROM    employees a, (SELECT  department_id,  
                        AVG(salary) salavg  
                     FROM    employees  
                     GROUP BY department_id) b  
WHERE   a.department_id = b.department_id  
AND     a.salary > b.salavg;
```

LAST_NAME	SALARY	DEPARTMENT_ID	SALAVG
Hartstein	13000	20	9500
Mourgos	5800	50	3500
Hunold	9000	60	6400
Zlotkey	10500	80	10033.3333
Abel	11000	80	10033.3333
King	24000	90	19333.3333
Higgins	12000	110	10150

7 rows selected.

ORACLE

Using a Subquery in the FROM Clause

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. The example on the slide displays employee last names, salaries, department numbers, and average salaries for all the employees who earn more than the average salary in their department. The subquery in the FROM clause is named b, and the outer query references the SALAVG column using this alias.

Instructor Note

You may wish to point out that the example demonstrates a useful technique to combine detail row values and aggregate data in the same output.

Scalar Subquery Expressions

- A scalar subquery expression is a subquery that returns exactly one column value from one row.
- Scalar subqueries were supported in Oracle8i only in a limited set of cases, For example:
 - SELECT statement (FROM and WHERE clauses)
 - VALUES list of an INSERT statement
- In Oracle9i, scalar subqueries can be used in:
 - Condition and expression part of DECODE and CASE
 - All clauses of SELECT except GROUP BY

ORACLE

18-11

Copyright © Oracle Corporation, 2001. All rights reserved.

Scalar Subqueries in SQL

A subquery that returns exactly one column value from one row is also referred to as a **scalar subquery**. Multiple-column subqueries written to compare two or more columns, using a compound WHERE clause and logical operators, do not qualify as scalar subqueries.

The value of the 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. The usage of scalar subqueries has been enhanced in Oracle9i. You can now use scalar subqueries in:

- Condition and expression part of DECODE and CASE
- All clauses of SELECT except GROUP BY
- In the left-hand side of the operator in the SET clause and WHERE clause of UPDATE statement

However, scalar subqueries are not valid expressions in the following places:

- As default values for columns and hash expressions for clusters
- In the RETURNING clause of DML statements
- As the basis of a function-based index
- In GROUP BY clauses, CHECK constraints, WHEN conditions
- HAVING clauses
- In START WITH and 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 ORDER BY Clause

```
SELECT   employee_id, last_name  
FROM     employees e  
ORDER BY (SELECT department_name  
          FROM departments d  
          WHERE e.department_id = d.department_id);
```

ORACLE

18-12

Copyright © Oracle Corporation, 2001. All rights reserved.

Scalar Subqueries: Examples

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 or not.

The result of the preceding example follows:

EMPLOYEE_ID	LAST_NAME	LOCATI
100	King	USA
101	Kochhar	USA
102	De Haan	USA
...		
201	Hartstein	Canada
202	Fay	Canada
205	Higgins	USA
206	Gietz	USA

20 rows selected.

Scalar Subqueries: Examples (continued)

The second example in the slide demonstrates that scalar subqueries can be used in the ORDER BY clause. The example orders the output based on the DEPARTMENT_NAME by matching the DEPARTMENT_ID from the EMPLOYEES table with the DEPARTMENT_ID from the DEPARTMENTS table. This comparison is done in a scalar subquery in the ORDER BY clause. The result of the second example follows:

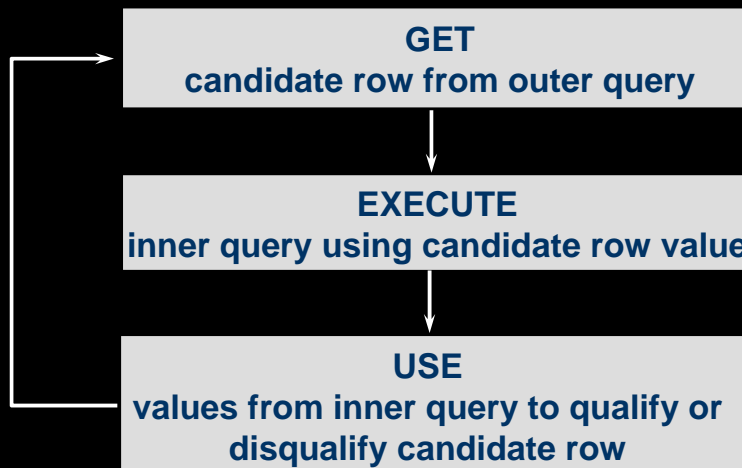
EMPLOYEE_ID	LAST_NAME
205	Higgins
206	Gietz
200	Whalen
100	King
101	Kochhar
102	De Haan
103	Hunold
104	Ernst
107	Lorentz
201	Hartstein
202	Fay
149	Zlotkey
176	Taylor
174	Abel
EMPLOYEE_ID	LAST_NAME
124	Mourgos
141	Rajs
142	Davies
143	Matos
144	Vargas
178	Grant

20 rows selected.

The second example uses a correlated subquery. In a **correlated subquery**, the subquery references a column from a table referred to in the parent statement. Correlated subqueries are explained later in this lesson.

Correlated Subqueries

Correlated subqueries are used for row-by-row processing. Each subquery is executed once for every row of the outer query.



ORACLE

18-14

Copyright © Oracle Corporation, 2001. All rights reserved.

Correlated Subqueries

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. In other words, 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 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

```
SELECT column1, column2, ...  
FROM   table1 outer  
WHERE  column1 operator  
        (SELECT column1, column2  
         FROM   table2  
         WHERE  expr1 =  
                outer.expr2);
```

The subquery references a column from a table in the parent query.

ORACLE

18-15

Copyright © Oracle Corporation, 2001. All rights reserved.

Correlated Subqueries (continued)

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. In other words, you use a correlated subquery to answer a multipart question whose answer depends on the value in each row processed by the parent statement.

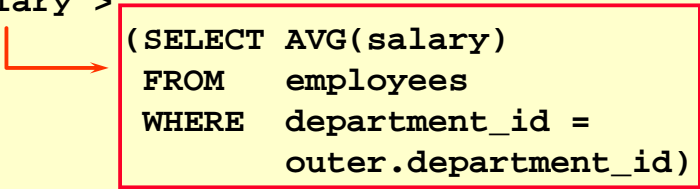
The Oracle Server performs a correlated subquery when the subquery references a column from a table in the parent query.

Note: You can use the ANY and ALL operators in a correlated subquery.

Using Correlated Subqueries

Find all employees who earn more than the average salary in their department.

```
SELECT last_name, salary, department_id
FROM   employees outer
WHERE  salary >
      (SELECT AVG(salary)
       FROM   employees
       WHERE  department_id =
             outer.department_id) ;
```



Each time a row from the outer query is processed, the inner query is evaluated.

ORACLE

18-16

Copyright © Oracle Corporation, 2001. All rights reserved.

Using Correlated Subqueries

The example in the slide determines which employees earn more than the average salary of 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. Not only does the alias make the entire SELECT statement more readable, but 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.

Instructor Note

You may wish to indicate that the aliases used are a syntactical requirement. The alias OUTER used here is mandatory, unlike other cases where an alias is used to add clarity and readability to the SQL statement.

Using Correlated Subqueries

Display details of those employees who have switched jobs at least twice.

```
SELECT e.employee_id, last_name, e.job_id
FROM   employees e
WHERE  2 <= (SELECT COUNT(*)
             FROM   job_history
             WHERE  employee_id = e.employee_id);
```

EMPLOYEE_ID	LAST_NAME	JOB_ID
101	Kochhar	AD_VP
176	Taylor	SA_REP
200	Whalen	AD_ASST

ORACLE

18-17

Copyright © Oracle Corporation, 2001. All rights reserved.

Using Correlated Subqueries

The example in the slide displays the details of those employees who have switched jobs at least twice. 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.EMPLOYEE_ID.)
3. Perform the subquery with its condition referencing the value from the outer query's candidate row. (In the example in the slide, group function COUNT(*) is evaluated based on the value of the E.EMPLOYEE_ID column obtained in step 2.)
4. Evaluate the WHERE clause of the outer query on the basis of results of the subquery performed in step 3. This determines if the candidate row is selected for output. (In the example, the number of times an employee has switched 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, the correlation is established by the statement `EMPLOYEE_ID = E.EMPLOYEE_ID` in which you compare `EMPLOYEE_ID` from the table in the subquery with the `EMPLOYEE_ID` from the table in the outer query.

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

18-18

Copyright © Oracle Corporation, 2001. All rights reserved.

The EXISTS Operator

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

Find employees who have at least one person reporting to them.

```
SELECT employee_id, last_name, job_id, department_id
FROM   employees outer
WHERE  EXISTS ( SELECT 'X'
                  FROM   employees
                  WHERE  manager_id =
                        outer.employee_id);
```

EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
100	King	AD_PRES	90
101	Kochhar	AD_VP	90
102	De Haan	AD_VP	90
103	Hunold	IT_PROG	60
124	Mourgos	ST_MAN	50
149	Zlotkey	SA_MAN	80
201	Hartstein	MK_MAN	20
205	Higgins	AC_MGR	110

8 rows selected.

ORACLE

Using the EXISTS Operator

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. From a performance standpoint, it is faster to select a constant than a column.

Note: Having EMPLOYEE_ID in the SELECT clause of the inner query causes a table scan for that column. Replacing it with the literal X, or any constant, improves performance. This is more efficient than using the IN operator.

A IN construct can be used as an alternative for a **EXISTS operator**, as shown in the following example:

```
SELECT employee_id, last_name, job_id, department_id
FROM   employees
WHERE  employee_id IN (SELECT manager_id
                       FROM   employees
                       WHERE  manager_id IS NOT NULL);
```

Using the NOT EXISTS Operator

Find all departments that do not have any employees.

```
SELECT department_id, department_name
FROM departments d
WHERE NOT EXISTS (SELECT 'X'
                  FROM employees
                  WHERE department_id
                     = d.department_id);
```

DEPARTMENT_ID	DEPARTMENT_NAME
190	Contracting

ORACLE

18-20

Copyright © Oracle Corporation, 2001. All rights reserved.

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);
```

no rows selected

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.

Correlated UPDATE

```
UPDATE table1 alias1
SET    column = (SELECT expression
                    FROM    table2 alias2
                    WHERE   alias1.column =
                        alias2.column);
```

Use a correlated subquery to update rows in one table based on rows from another table.

ORACLE

18-21

Copyright © Oracle Corporation, 2001. All rights reserved.

Correlated UPDATE

In the case of the UPDATE statement, you can use a **correlated subquery to update rows** in one table based on rows from another table.

Correlated UPDATE

- Denormalize the EMPLOYEES table by adding a column to store the department name.
- Populate the table by using a correlated update.

```
ALTER TABLE employees
ADD(department_name VARCHAR2(14));
```

```
UPDATE employees e
SET    department_name =
        (SELECT department_name
         FROM   departments d
         WHERE  e.department_id = d.department_id);
```

ORACLE

18-22

Copyright © Oracle Corporation, 2001. All rights reserved.

Correlated UPDATE (continued)

The example in the slide denormalizes the EMPLOYEES table by adding a column to store the department name and then populates the table by using a correlated update.

Here is another example for a **correlated update**.

Problem Statement

Use a correlated subquery to update rows in the EMPLOYEES table based on rows from the REWARDS table:

```
UPDATE employees
SET    salary = (SELECT employees.salary + rewards.pay_raise
                 FROM    rewards
                 WHERE    employee_id = employees.employee_id
                 AND      payraise_date =
                        (SELECT MAX(payraise_date)
                         FROM    rewards
                         WHERE    employee_id = employees.employee_id))
WHERE  employees.employee_id
IN      (SELECT employee_id
        FROM    rewards);
```

Instructor Note

In order to demonstrate the code example in the notes, you must first run the script file `\labs\cre_reward.sql`, which creates the REWARDS table and inserts records into the table. Remember to **rollback** the transaction if you demo the script in the slide or notes page. This is very important as if this is not done, the outputs shown in the practices will not match.

Correlated UPDATE (continued)

This example uses the REWARDS table. The REWARDS table has the columns EMPLOYEE_ID, PAY_RAISE, and PAYRAISE_DATE. Every time an employee gets a pay raise, a record with the details of 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 EMPLOYEES table is updated to reflect the latest pay raise received by the employee. This is done by adding the current salary of the employee with the corresponding pay raise from the REWARDS table.

Correlated DELETE

```
DELETE FROM table1 alias1
WHERE column operator
      (SELECT expression
       FROM table2 alias2
       WHERE alias1.column = alias2.column);
```

Use a correlated subquery to delete rows in one table based on rows from another table.

ORACLE

18-24

Copyright © Oracle Corporation, 2001. All rights reserved.

Correlated DELETE

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, then when an employee transfers to a fifth job, you delete the oldest JOB_HISTORY row by looking up the JOB_HISTORY table for the MIN(START_DATE) for the employee. The following code illustrates how the preceding operation can be performed 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));
```

Correlated DELETE

Use a correlated subquery to delete only those rows from the EMPLOYEES table that also exist in the EMP_HISTORY table.

```
DELETE FROM employees E
WHERE employee_id =
      (SELECT employee_id
       FROM   emp_history
       WHERE  employee_id = E.employee_id);
```

ORACLE

18-25

Copyright © Oracle Corporation, 2001. All rights reserved.

Correlated DELETE (continued)

Example

Two tables are used in this example. They are:

- The EMPLOYEES table, which gives details of all the current employees
- The EMP_HISTORY table, which gives details of previous employees

EMP_HISTORY contains data regarding previous employees, so it would be erroneous if the same employee's record existed in both the EMPLOYEES and EMP_HISTORY tables. You can delete such erroneous records by using the correlated subquery shown in the slide.

Instructor Note

In order to demonstrate the code example in the slide, you must first run the script file `\labs\cre_emphistory.sql`, which creates the EMP_HISTORY table and inserts records into the table.

The 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 improves performance

ORACLE

18-26

Copyright © Oracle Corporation, 2001. All rights reserved.

The WITH clause

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 high cost 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, thereby enhancing performance

WITH Clause: Example

Using the WITH clause, write a query to display the department name and total salaries for those departments whose total salary is greater than the average salary across departments.

ORACLE

18-27

Copyright © Oracle Corporation, 2001. All rights reserved.

WITH Clause: Example

The problem in the slide would require the following intermediate calculations:

1. Calculate the total salary for every department, and store the result using a WITH clause.
2. Calculate the average salary across departments, and store the result using a WITH clause.
3. Compare the total salary calculated in the first step with the average salary calculated in the second step. If the total salary for a particular department is greater than the average salary across departments, display the department name and the total salary for that department.

The solution for the preceding problem is given in the next page.

WITH Clause: Example

WITH

```
dept_costs AS (  
    SELECT d.department_name, SUM(e.salary) AS dept_total  
    FROM   employees e, departments d  
    WHERE  e.department_id = d.department_id  
    GROUP BY d.department_name),  
avg_cost AS (  
    SELECT SUM(dept_total)/COUNT(*) AS dept_avg  
    FROM   dept_costs)  
SELECT *  
FROM   dept_costs  
WHERE  dept_total >  
       (SELECT dept_avg  
        FROM avg_cost)  
ORDER BY department_name;
```

ORACLE

18-28

Copyright © Oracle Corporation, 2001. All rights reserved.

WITH Clause: Example (continued)

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 names DEPT_COSTS and AVG_COST and then uses them in the body of the main query. Internally, the WITH clause is resolved either as an in-line view or a temporary table. The optimizer chooses the appropriate resolution depending on the cost or benefit of temporarily storing the results of the WITH clause.

Note: A subquery in the FROM clause of a SELECT statement is also called an in-line view.

The output generated by the SQL code on the slide will be as follows:

DEPARTMENT_NAME	DEPT_TOTAL
Executive	58000
Sales	30100

The 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) 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, 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.

Summary

In this lesson, you should have learned the following:

- **A multiple-column subquery returns more than one column.**
- **Multiple-column comparisons can be pairwise or nonpairwise.**
- **A multiple-column subquery can also be used in the `FROM` clause of a `SELECT` statement.**
- **Scalar subqueries have been enhanced in Oracle9i.**

ORACLE

18-29

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary

You can use multiple-column subqueries to combine multiple `WHERE` conditions into a single `WHERE` clause. Column comparisons in a multiple-column subquery can be pairwise comparisons or non-pairwise comparisons.

You can use a subquery to define a table to be operated on by a containing query.

Oracle 9i enhances the the uses of scalar subqueries. Scalar subqueries can now be used in:

- Condition and expression part of `DECODE` and `CASE`
- All clauses of `SELECT` except `GROUP BY`
- `SET` clause and `WHERE` clause of `UPDATE` statement

Summary

- **Correlated subqueries are useful whenever a subquery must return a different result for each candidate row.**
- **The `EXISTS` operator is a Boolean operator that tests the presence of a value.**
- **Correlated subqueries can be used with `SELECT`, `UPDATE`, and `DELETE` statements.**
- **You can use the `WITH` clause to use the same query block in a `SELECT` statement when it occurs more than once**

ORACLE

18-30

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary (continued)

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. Using the `WITH` clause, you can reuse the same query when it is costly to reevaluate the query block and it occurs more than once within a complex query.

Practice 18 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

ORACLE

18-31

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 18 Overview

In this practice, you write multiple-column subqueries, correlated and scalar subqueries. You also solve problems by writing the **WITH** clause.

Instructor Note

You might want to recap the **ALL** and **ANY** operators before the students start the practice. This is required for the questions.

ALL: Compares a value to every value in a list or returned by a query. Must be preceded by **=**, **!=**, **>**, **<**, **<=**, **>=**. Evaluates to **TRUE** if the query returns no rows.

```
SELECT * FROM employees
WHERE salary > = ALL ( 1400, 3000 );
```

ANY: Compares a value to each value in a list or returned by a query. Must be preceded by **=**, **!=**, **>**, **<**, **<=**, **>=**. Evaluates to **FALSE** if the query returns no rows.

```
SELECT * FROM employees
WHERE salary = ANY
  (SELECT salary FROM employees
WHERE department_id = 30);
```

Practice 18

- Write a query to display the last name, department number, and salary of any employee whose department number and salary both match the department number and salary of any employee who earns a commission.

LAST_NAME	DEPARTMENT_ID	SALARY
Taylor	80	8600
Zlotkey	80	10500
Abel	80	11000

- Display the last name, department name, and salary of any employee whose salary and commission match the salary and commission of any employee located in location ID 1700.

LAST_NAME	DEPARTMENT_NAME	SALARY
Whalen	Administration	4400
Gietz	Accounting	8300
Higgins	Accounting	12000
Kochhar	Executive	17000
De Haan	Executive	17000
King	Executive	24000

6 rows selected.

- Create a query to display the last name, hire date, and salary for all employees who have the same salary and commission as Kochhar.

Note: Do not display Kochhar in the result set.

LAST_NAME	HIRE_DATE	SALARY
De Haan	13-JAN-93	17000

- Create a query to display the employees who earn a salary that is higher than the salary of all of the sales managers (JOB_ID = 'SA_MAN'). Sort the results on salary from highest to lowest.

LAST_NAME	JOB_ID	SALARY
King	AD_PRES	24000
Kochhar	AD_VP	17000
De Haan	AD_VP	17000
Hartstein	MK_MAN	13000
Higgins	AC_MGR	12000
Abel	SA_REP	11000

6 rows selected.

Practice 18 (continued)

5. Display the details of the employee ID, last name, and department ID of those employees who live in cities whose name begins with *T*.

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
201	Hartstein	20
202	Fay	20

6. Write a query to find all employees who earn more than the average salary in their departments. Display last name, salary, department ID, and the average salary for the department. Sort by average salary. Use aliases for the columns retrieved by the query as shown in the sample output.

ENAME	SALARY	DEPTNO	DEPT_AVG
Mourgos	5800	50	3500
Hunold	9000	60	6400
Hartstein	13000	20	9500
Abel	11000	80	10033.3333
Zlotkey	10500	80	10033.3333
Higgins	12000	110	10150
King	24000	90	19333.3333

7 rows selected.

7. Find all employees who are not supervisors.
a. First do this using the NOT EXISTS operator.

LAST_NAME
Ernst
Lorentz
Rajs
Davies
Matos
Vargas
Abel
Taylor
Grant
Whalen
Fay
Gietz

12 rows selected.

- b. Can this be done by using the NOT IN operator? How, or why not?

Practice 18 (continued)

8. Write a query to display the last names of the employees who earn less than the average salary in their departments.

LAST_NAME
Kochhar
De Haan
Ernst
Lorentz
Davies
Matos
Vargas
Taylor
Fay
Gietz

10 rows selected.

9. Write a query to display the last names of the employees who have one or more coworkers in their departments with later hire dates but higher salaries.

LAST_NAME
Rajs
Davies
Matos
Vargas
Taylor

Practice 18 (continued)

10. Write a query to display the employee ID, last names, and department names of all employees.

Note: Use a scalar subquery to retrieve the department name in the SELECT statement.

EMPLOYEE_ID	LAST_NAME	DEPARTMENT
205	Higgins	Accounting
206	Gietz	Accounting
200	Whalen	Administration
100	King	Executive
101	Kochhar	Executive
102	De Haan	Executive
103	Hunold	IT
104	Ernst	IT
107	Lorentz	IT
201	Hartstein	Marketing
202	Fay	Marketing
149	Zlotkey	Sales
176	Taylor	Sales
174	Abel	Sales
EMPLOYEE_ID	LAST_NAME	DEPARTMENT
124	Mourgos	Shipping
141	Rajs	Shipping
142	Davies	Shipping
143	Matos	Shipping
144	Vargas	Shipping
178	Grant	

20 rows selected.

11. Write a query to display the department names of those departments whose total salary cost is above one eighth (1/8) of the total salary cost of the whole company. Use the WITH clause to write this query. Name the query SUMMARY.

DEPARTMENT_NAME	DEPT_TOTAL
Executive	58000
Sales	30100

19

Hierarchical Retrieval

ORACLE®

Copyright © Oracle Corporation, 2001. All rights reserved.

Schedule:	Timing	Topic
	30 minutes	Lecture
	20 minutes	Practice
	50 minutes	Total

Objectives

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

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

ORACLE®

19-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Lesson Aim

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

Sample Data from the EMPLOYEES Table

EMPLOYEE_ID	LAST_NAME	JOB_ID	MANAGER_ID
100	King	AD_PRES	
101	Kochhar	AD_VP	100
102	De Haan	AD_VP	100
103	Hunold	IT_PROG	102
104	Ernst	IT_PROG	103
107	Lorentz	IT_PROG	103
124	Mourgos	ST_MAN	100
141	Rajs	ST_CLERK	124
142	Davies	ST_CLERK	124
143	Matos	ST_CLERK	124
144	Vargas	ST_CLERK	124
149	Zlotkey	SA_MAN	100
174	Abel	SA_REP	149
176	Taylor	SA_REP	149
EMPLOYEE_ID	LAST_NAME	JOB_ID	MANAGER_ID
178	Grant	SA_REP	149
200	Whalen	AD_ASST	101
201	Hartstein	MK_MAN	100
202	Fay	MK_REP	201
205	Higgins	AC_MGR	101
206	Gietz	AC_ACCOUNT	205

20 rows selected.

ORACLE

19-3

Copyright © Oracle Corporation, 2001. All rights reserved.

Sample Data from the EMPLOYEES Table

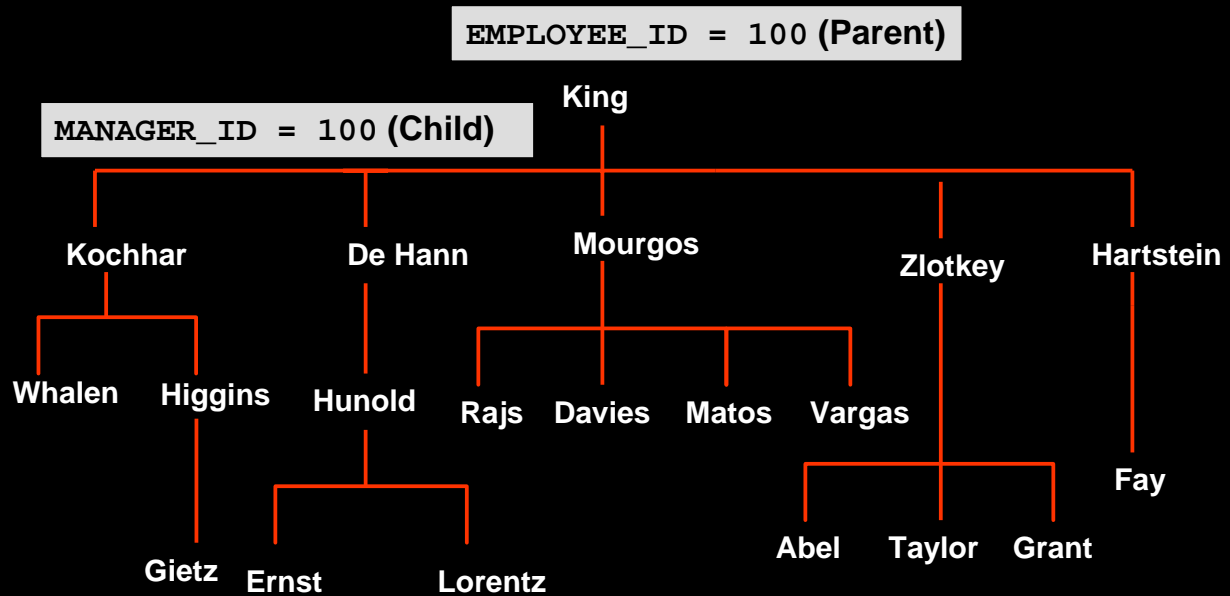
Using **hierarchical queries**, you can retrieve data based on a natural hierarchical relationship between rows in a table. A relational database does not store records in a hierarchical way. However, where a hierarchical relationship exists between the rows of a single table, a process called *tree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

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

A hierarchical query is possible when a relationship exists between rows in a table. For example, in the slide, you see that employees with the job IDs of AD_VP, ST_MAN, SA_MAN, and MK_MAN report directly to the president of the company. We know this because the MANAGER_ID column of these records contain the employee ID 100, which belongs to the president (AD_PRES).

Note: Hierarchical trees are used in various fields such as human genealogy (family trees), livestock (breeding purposes), corporate management (management hierarchies), manufacturing (product assembly), evolutionary research (species development), and scientific research.

Natural Tree Structure



ORACLE

19-4

Copyright © Oracle Corporation, 2001. All rights reserved.

Natural Tree Structure

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

The **parent-child relationship** of a tree structure enables you to control:

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

Note: The slide displays an inverted tree structure of the management hierarchy of the employees in the EMPLOYEES table.

Instructor Note

You can use the data shown in the previous slide to explain the tree structure shown in the slide.

Hierarchical Queries

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

WHERE *condition*:

expr comparison_operator expr

ORACLE

19-5

Copyright © Oracle Corporation, 2001. All rights reserved.

Keywords and Clauses

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

In the syntax:

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

The SELECT statement cannot contain a join or query from a view that contains a join.

Walking the Tree

Starting Point

- Specifies the condition that must be met
- Accepts any valid condition

```
START WITH column1 = value
```

Using the **EMPLOYEES** table, start with the employee whose last name is Kochhar.

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

ORACLE

19-6

Copyright © Oracle Corporation, 2001. All rights reserved.

Walking the Tree

The row or rows to be used as the root of the tree are determined by the **START WITH** clause. The **START WITH** clause can be used in conjunction with any valid condition.

Examples

Using the **EMPLOYEES** table, start with King, the president of the company.

```
... START WITH manager_id IS NULL
```

Using the **EMPLOYEES** table, start with employee Kochhar. A **START WITH** condition can contain a subquery.

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

If the **START WITH** clause is omitted, the tree walk is started with all of the rows in the table as root rows. If a **WHERE** clause is used, the walk is started with all the rows that satisfy the **WHERE** condition. This no longer reflects a true hierarchy.

Note: The clauses **CONNECT BY PRIOR** and **START WITH** are not ANSI SQL standard.

Instructor Note

You may wish to add that multiple hierarchical outputs are generated if more than one row satisfies the **START WITH** condition.

Walking the Tree

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

Walk from the top down, using the **EMPLOYEES** table.

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

Direction

Top down	→	Column1 = Parent Key Column2 = Child Key
Bottom up	→	Column1 = Child Key Column2 = Parent Key

ORACLE

19-7

Copyright © Oracle Corporation, 2001. All rights reserved.

Walking the Tree (continued)

The direction of the query, whether it is from parent to child or from child to parent, is determined by the `CONNECT BY PRIOR` column placement. The **PRIOR operator** refers to the parent row. To find the children of a parent row, the Oracle Server evaluates the `PRIOR` expression for the parent row and the other expressions for each row in the table. Rows for which the condition is true are the children of the parent. The Oracle Server always selects children by evaluating the **CONNECT BY** condition with respect to a current parent row.

Examples

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

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

Walk from the bottom up using the **EMPLOYEES** table.

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

The `PRIOR` operator does not necessarily need to be coded immediately following the `CONNECT BY`. Thus, the following `CONNECT BY PRIOR` clause gives the same result as the one in the preceding example.

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

Note: The `CONNECT BY` clause cannot contain a subquery.

Walking the Tree: From the Bottom Up

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

EMPLOYEE_ID	LAST_NAME	JOB_ID	MANAGER_ID
101	Kochhar	AD_VP	100
100	King	AD_PRES	

ORACLE

19-8

Copyright © Oracle Corporation, 2001. All rights reserved.

Walking the Tree: From the Bottom Up

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

Example

In the following example, EMPLOYEE_ID values are evaluated for the parent row and MANAGER_ID, and SALARY values are evaluated for the child rows. The PRIOR operator applies only to the EMPLOYEE_ID value.

```
... CONNECT BY PRIOR employee_id = manager_id
                AND salary > 15000;
```

To qualify as a child row, a row must have a MANAGER_ID value equal to the EMPLOYEE_ID value of the parent row and must have a SALARY value greater than \$15,000.

Instructor Note

In the context of the first paragraph, you may wish to include here that the hierarchy will be established to the furthest extremity before the next parent row is evaluated.

In the context of the second paragraph, you may wish to include that additional conditions added to the CONNECT BY PRIOR clause potentially eliminated the whole of the branch, hence the EMPLOYEE_ID AND SALARY are evaluated for the parent row to determine if it is to be part of the output.

Walking the Tree: From the Top Down

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

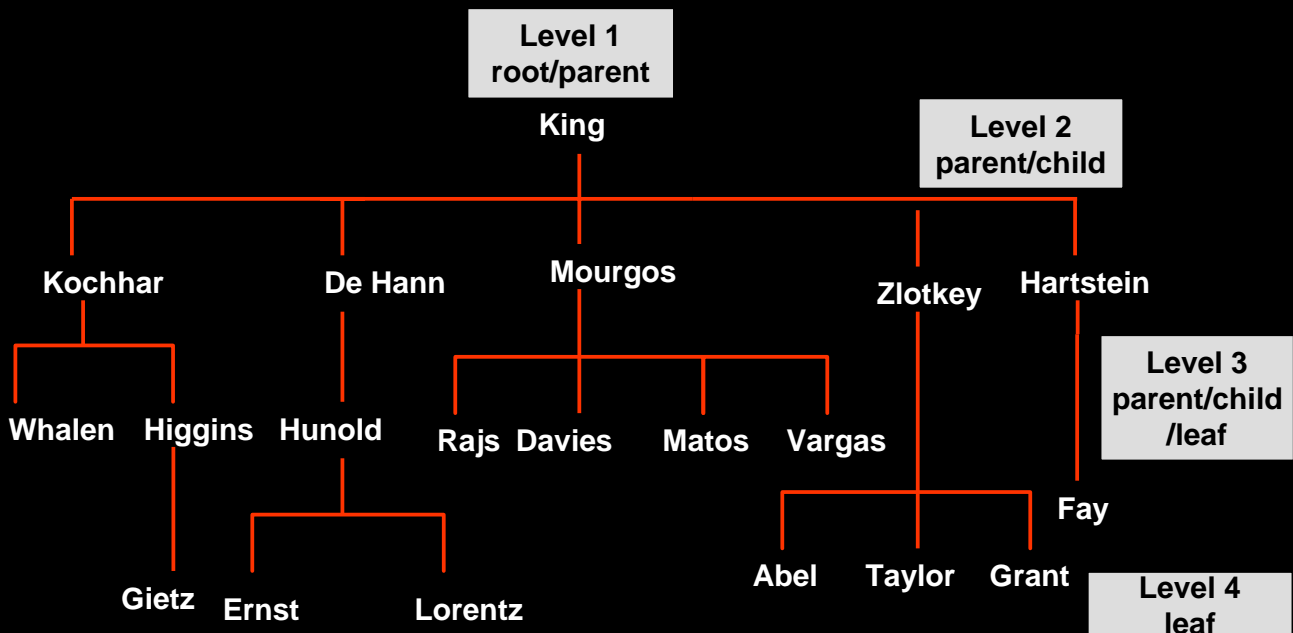
Walk Top Down	
King reports to	
Kochhar reports to King	
Whalen reports to Kochhar	
Higgins reports to Kochhar	
...	
Zlotkey reports to King	
Abel reports to Zlotkey	
Taylor reports to Zlotkey	
Grant reports to Zlotkey	
Hartstein reports to King	
Fay reports to Hartstein	
20 rows selected.	

ORACLE

Walking the Tree: From the Top Down

Walking from the top down, display the names of the employees and their manager. Use employee King as the starting point. Print only one column.

Ranking Rows with the LEVEL Pseudocolumn



ORACLE

19-10

Copyright © Oracle Corporation, 2001. All rights reserved.

Ranking Rows with the LEVEL Pseudocolumn

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

The LEVEL Pseudocolumn

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

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

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

Formatting Hierarchical Reports Using LEVEL and LPAD

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

```
COLUMN org_chart FORMAT A12
SELECT LPAD(last_name, LENGTH(last_name)+(LEVEL*2)-2, '_')
       AS org_chart
FROM   employees
START WITH last_name='King'
CONNECT BY PRIOR employee_id=manager_id
```

ORACLE

19-11

Copyright © Oracle Corporation, 2001. All rights reserved.

Formatting Hierarchical Reports Using LEVEL

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

In the example on the slide:

- `LPAD(char1, n [, char2])` returns `char1`, left-padded to length `n` with the sequence of characters in `char2`. The argument `n` is the total length of the return value as it is displayed on your terminal screen.
- `LPAD(last_name, LENGTH(last_name)+(LEVEL*2)-2, '_')` defines the display format.
- `char1` is the `LAST_NAME`, `n` the total length of the return value, is length of the `LAST_NAME` + $(LEVEL * 2) - 2$, and `char2` is `'_'`.

In other words, this tells SQL to take the `LAST_NAME` and left-pad it with the `'_'` character till the length of the resultant string is equal to the value determined by $LENGTH(last_name) + (LEVEL * 2) - 2$.

For King, $LEVEL = 1$. Hence, $(2 * 1) - 2 = 2 - 2 = 0$. So King does not get padded with any `'_'` character and is displayed in column 1.

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

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

Formatting Hierarchical Reports Using LEVEL (continued)

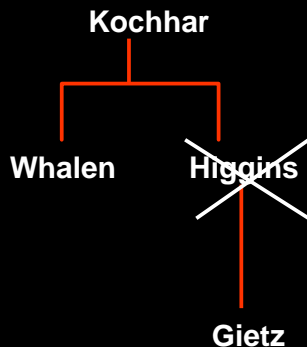
ORG_CHART	
King	
_Kochhar	
_Whalen	
_Higgins	
_Gietz	
_De Haan	
_Hunold	
_Ernst	
_Lorentz	
_Mourgos	
_Rajs	
_Davies	
_Matos	
_Vargas	
ORG_CHART	
_Zlotkey	
_Abel	
_Taylor	
_Grant	
_Hartstein	
_Fay	

20 rows selected.

Pruning Branches

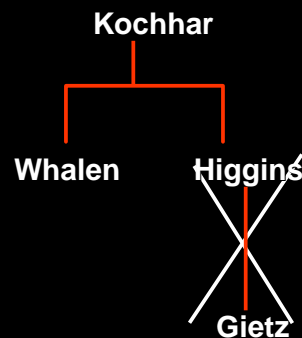
Use the **WHERE** clause
to eliminate a node.

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



Use the **CONNECT BY** clause
to eliminate a branch.

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



ORACLE

19-13

Copyright © Oracle Corporation, 2001. All rights reserved.

Pruning Branches

You can use the **WHERE** and **CONNECT BY** clauses to **prune the tree**; that is, to control which nodes or rows are displayed. The predicate you use acts as a Boolean condition.

Examples

Starting at the root, walk from the top down, and eliminate employee Higgins in the result, but process the child rows.

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

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

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

Instructor Note

You may wish to add here that using a **WHERE** clause to restrict a node could result in the hierarchy not being reflected truly by the output.

Introduction to Oracle9i: SQL 19-13

Summary

In this lesson, you should have learned the following:

- **You can use hierarchical queries to view a hierarchical relationship between rows in a table.**
- **You specify the direction and starting point of the query.**
- **You can eliminate nodes or branches by pruning.**

ORACLE

19-14

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary

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

Practice 19 Overview

This practice covers the following topics:

- **Distinguishing hierarchical queries from nonhierarchical queries**
- **Walking through a tree**
- **Producing an indented report by using the `LEVEL` pseudocolumn**
- **Pruning the tree structure**
- **Sorting the output**

ORACLE

19-15

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 19 Overview

In this practice, you gain practical experience in producing hierarchical reports.

Paper-Based Questions

Question 1 is a paper-based question.

Practice 19

1. Look at the following outputs. Are these outputs the result of a hierarchical query? Explain why or why not.

Exhibit 1:

EMPLOYEE_ID	LAST_NAME	MANAGER_ID	SALARY	DEPARTMENT_ID
100	King		24000	90
101	Kochhar	100	17000	90
102	De Haan	100	17000	90
201	Hartstein	100	13000	20
205	Higgins	101	12000	110
174	Abel	149	11000	80
149	Zlotkey	100	10500	80
103	Hunold	102	9000	60
■ ■ ■				
200	Whalen	101	4400	10
107	Lorentz	103	4200	60
141	Rajs	124	3500	50
142	Davies	124	3100	50
143	Matos	124	2600	50
144	Vargas	124	2500	50

20 rows selected.

Exhibit 2:

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
205	Higgins	110	Accounting
206	Gietz	110	Accounting
100	King	90	Executive
101	Kochhar	90	Executive
102	De Haan	90	Executive
149	Zlotkey	80	Sales
174	Abel	80	Sales
176	Taylor	80	Sales
103	Hunold	60	IT
104	Ernst	60	IT
107	Lorentz	60	IT

11 rows selected.

Practice 19 (continued)

Exhibit 3:

RANK	LAST_NAME
1	King
2	Kochhar
2	De Haan
3	Hunold
4	Ernst

2. Produce a report showing an organization chart for Mourgos's department. Print last names, salaries, and department IDs.

LAST_NAME	SALARY	DEPARTMENT_ID
Mourgos	5800	50
Rajs	3500	50
Davies	3100	50
Matos	2600	50
Vargas	2500	50

3. Create a report that shows the hierarchy of the managers for the employee Lorentz. Display his immediate manager first.

LAST_NAME
Hunold
De Haan
King

Practice 19 (continued)

4. Create an indented report showing the management hierarchy starting from the employee whose LAST_NAME is Kochhar. Print the employee's last name, manager ID, and department ID. Give alias names to the columns as shown in the sample output.

NAME	MGR	DEPTNO
Kochhar	100	90
_Whalen	101	10
_Higgins	101	110
_Gietz	205	110

If you have time, complete the following exercise:

5. Produce a company organization chart that shows the management hierarchy. Start with the person at the top level, exclude all people with a job ID of IT_PROG, and exclude De Haan and those employees who report to De Haan.

LAST_NAME	EMPLOYEE_ID	MANAGER_ID
King	100	
Kochhar	101	100
Whalen	200	101
Higgins	205	101
Gietz	206	205
Mourgos	124	100
Rajs	141	124
Davies	142	124
Matos	143	124
Vargas	144	124
Zlotkey	149	100
Abel	174	149
Taylor	176	149
Grant	178	149
LAST_NAME	EMPLOYEE_ID	MANAGER_ID
Hartstein	201	100
Fay	202	201

16 rows selected.

20

Oracle9i Extensions to DML and DDL Statements

ORACLE

Copyright © Oracle Corporation, 2001. All rights reserved.

Schedule:

Timing

40 minutes

30 minutes

70 minutes

Topic

Lecture

Practice

Total

Objectives

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

- Describe the features of multitable inserts
- Use the following types of multitable inserts
 - Unconditional `INSERT`
 - Pivoting `INSERT`
 - Conditional `ALL INSERT`
 - Conditional `FIRST INSERT`
- Create and use external tables
- Name the index at the time of creating a primary key constraint

ORACLE

20-2

Copyright © Oracle Corporation, 2001. All rights reserved.

Lesson Aim

This lesson addresses the Oracle9i extensions to DDL and DML statements. It focuses on **multitable `INSERT` statements**, types of multitable `INSERT` statements, **external tables**, and the provision to **name the index** at the time of creating a primary key constraint.

Review of the INSERT Statement

- Add new rows to a table by using the INSERT statement.

```
INSERT INTO  table [(column [, column...])]  
VALUES      (value [, value...]);
```

- Only one row is inserted at a time with this syntax.

```
INSERT INTO departments(department_id, department_name,  
                        manager_id, location_id)  
VALUES      (70, 'Public Relations', 100, 1700);  
1 row created.
```

ORACLE

Review of the INSERT Statement

You can add new rows to a table by issuing the INSERT statement.

In the syntax:

<i>table</i>	is the name of the table
<i>column</i>	is the name of the column in the table to populate
<i>value</i>	is the corresponding value for the column

Note: This statement with the VALUES clause adds only one row at a time to a table.

Instructor Note

You can skip this slide if the students are already familiar with these concepts.

Review of the UPDATE Statement

- **Modify existing rows with the UPDATE statement.**

```
UPDATE      table
SET         column = value [, column = value, ...]
[WHERE      condition];
```

- **Update more than one row at a time, if required.**
- **Specific row or rows are modified if you specify the WHERE clause.**

```
UPDATE employees
SET    department_id = 70
WHERE  employee_id = 142;
1 row updated.
```

ORACLE

20-4

Copyright © Oracle Corporation, 2001. All rights reserved.

Review of the UPDATE Statement

You can modify existing rows by using the UPDATE statement.

In the syntax:

<i>table</i>	is the name of the table
<i>column</i>	is the name of the column in the table to populate
<i>value</i>	is the corresponding value or subquery for the column
<i>condition</i>	identifies the rows to be updated and is composed of column names expressions, constants, subqueries, and comparison operators

Confirm the update operation by querying the table to display the updated rows.

Instructor Note

You can skip this slide if the students are already familiar with these concepts.

Overview of Multitable `INSERT` Statements

- The `INSERT . . . SELECT` statement can be used to insert rows into multiple tables as part of a single DML statement.
- Multitable `INSERT` statements can be used in data warehousing systems to transfer data from one or more operational sources to a set of target tables.
- They provide significant performance improvement over:
 - Single DML versus multiple `INSERT . . . SELECT` statements
 - Single DML versus a procedure to do multiple inserts using `IF . . . THEN` syntax

ORACLE

20-5

Copyright © Oracle Corporation, 2001. All rights reserved.

Overview of Multitable `INSERT` Statements

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 can play a very useful role 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 needs to 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 has to be identified and extracted from many different sources, such as database systems and applications. After extraction, the data has to 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.

Once data is loaded into an Oracle9i database, data transformations can be executed using SQL operations. With Oracle9i multitable **`INSERT`** statements is one of the techniques for implementing SQL data transformations.

Overview of Multitable Insert Statements (continued)

Multitable INSERTS statement offer the benefits of the INSERT . . . SELECT statement when multiple tables are involved as targets. Using functionality prior to Oracle9i, 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 more relational database table environment. To implement this functionality before Oracle9i, you had to write multiple INSERT statements.

Types of Multitable INSERT Statements

Oracle9i introduces the following types of multitable insert statements:

- Unconditional INSERT
- Conditional ALL INSERT
- Conditional FIRST INSERT
- Pivoting INSERT

ORACLE

20-7

Copyright © Oracle Corporation, 2001. All rights reserved.

Types of Multitable INSERT Statements

Oracle 9i introduces the following types of **multitable INSERT statements**:

- Unconditional INSERT
- Conditional ALL INSERT
- Conditional FIRST INSERT
- Pivoting INSERT

You use different clauses to indicate the type of INSERT to be executed.

Multitable INSERT Statements

Syntax

```
INSERT [ALL] [conditional_insert_clause]  
[insert_into_clause values_clause] (subquery)
```

conditional_insert_clause

```
[ALL] [FIRST]  
[WHEN condition THEN] [insert_into_clause values_clause]  
[ELSE] [insert_into_clause values_clause]
```

ORACLE

20-8

Copyright © Oracle Corporation, 2001. All rights reserved.

Multitable INSERT Statements

The slide displays the generic format for multitable INSERT statements. There are four types of multitable insert statements.

- Unconditional INSERT
- Conditional ALL INSERT
- Conditional FIRST INSERT
- Pivoting INSERT

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.

Multitable INSERT Statements (continued)

Conditional FIRST: INSERT

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 inserts only on tables, not on views or materialized views.
- You cannot perform a multitable insert into a remote table.
- You cannot specify a table collection expression when performing a multitable insert.
- In a multitable insert, all of the `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 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 created.
```

ORACLE

20-10

Copyright © Oracle Corporation, 2001. All rights reserved.

Unconditional INSERT ALL

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`, as 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 have to 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.

The feedback `8 rows created` can be interpreted to mean that a total of eight insertions were performed on the base tables `SAL_HISTORY` and `MGR_HISTORY`.

Instructor Note

In order to demonstrate the code example in the slide, you must first run the script files `lab\cre_sal_history.sql` and `lab\cre_mgr_history.sql`, which create the `SAL_HISTORY` and `MGR_HISTORY` tables.

Conditional 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.
- If the `SALARY` is greater than \$10,000, insert these values into the `SAL_HISTORY` table using a conditional multitable `INSERT` statement.
- If the `MANAGER_ID` is greater than 200, insert these values into the `MGR_HISTORY` table using a conditional multitable `INSERT` statement.

ORACLE

20-11

Copyright © Oracle Corporation, 2001. All rights reserved.

Conditional INSERT ALL

The problem statement for a conditional `INSERT ALL` statement is specified in the slide. The solution to the preceding problem is shown in the next page.

Conditional INSERT ALL

```
INSERT ALL
  WHEN SAL > 10000 THEN
    INTO sal_history VALUES(EMPID,HIREDATE,SAL)
  WHEN MGR > 200 THEN
    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;
4 rows created.
```

ORACLE

20-12

Copyright © Oracle Corporation, 2001. All rights reserved.

Conditional INSERT ALL (continued)

The example in the slide is similar to the example on the previous slide as it 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 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 a conditional ALL INSERT, as 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 value of the SAL column is more than 10000 are inserted in the SAL_HISTORY table, and similarly only those rows where the value of the MGR column is more than 200 are inserted in the MGR_HISTORY table.

Observe that unlike the previous example, where eight rows were inserted into the tables, in this example only four rows are inserted.

The feedback 4 rows created can be interpreted to mean that a total of four inserts were performed on the base tables, SAL_HISTORY and MGR_HISTORY.

Conditional FIRST INSERT

- Select the DEPARTMENT_ID , SUM(SALARY) and MAX(HIRE_DATE) from the EMPLOYEES table.
- If the SUM(SALARY) is greater than \$25,000 then insert these values into the SPECIAL_SAL, using a conditional FIRST multitable INSERT.
- If the first WHEN clause evaluates to true, the subsequent WHEN clauses for this row should be skipped.
- For the rows that do not satisfy the first WHEN condition, insert into the HIREDATE_HISTORY_00, or HIREDATE_HISTORY_99, or HIREDATE_HISTORY tables, based on the value in the HIRE_DATE column using a conditional multitable INSERT.

ORACLE

20-13

Copyright © Oracle Corporation, 2001. All rights reserved.

Conditional FIRST INSERT

The problem statement for a conditional FIRST INSERT statement is specified in the slide. The solution to the preceding problem is shown on the next page.

Conditional FIRST INSERT

```
INSERT FIRST
  WHEN SAL > 25000 THEN
    INTO special_sal VALUES(DEPTID, SAL)
  WHEN HIREDATE like ('%00%') THEN
    INTO hiredate_history_00 VALUES(DEPTID, HIREDATE)
  WHEN HIREDATE like ('%99%') THEN
    INTO hiredate_history_99 VALUES(DEPTID, HIREDATE)
  ELSE
    INTO hiredate_history VALUES(DEPTID, HIREDATE)
  SELECT department_id DEPTID, SUM(salary) SAL,
    MAX(hire_date) HIREDATE
  FROM employees
  GROUP BY department_id;
8 rows created.
```

ORACLE

20-14

Copyright © Oracle Corporation, 2001. All rights reserved.

Conditional FIRST INSERT (continued)

The example in the slide inserts rows into more than one table, using one single INSERT statement. The SELECT statement retrieves the details of department ID, total salary, and maximum hire date for every department in the EMPLOYEES table.

This INSERT statement is referred to as a **conditional FIRST INSERT**, as an exception is made for the departments whose total salary is more than \$25,000. The condition WHEN ALL > 25000 is evaluated first. If the total salary for a department is more than \$25,000, then the record is inserted into the SPECIAL_SAL table irrespective of the hire date. If this first WHEN clause evaluates to true, the Oracle Server executes the corresponding INTO clause and skips subsequent WHEN clauses for this row.

For the rows that do not satisfy the first WHEN condition (WHEN SAL > 25000), the rest of the conditions are evaluated just as a conditional INSERT statement, and the records retrieved by the SELECT statement are inserted into the HIREDATE_HISTORY_00, or HIREDATE_HISTORY_99, or HIREDATE_HISTORY tables, based on the value in the HIREDATE column.

The feedback 8 rows created can be interpreted to mean that a total of eight INSERT statements were performed on the base tables SPECIAL_SAL, HIREDATE_HISTORY_00, HIREDATE_HISTORY_99, and HIREDATE_HISTORY.

Instructor Note

In order to demonstrate the code example in the slide, you must first run the script files lab\cre_special_sal.sql, lab\cre_hiredate_history_99.sql, lab\cre_hiredate_history_00.sql and lab\cre_hiredate_history.sql

Pivoting INSERT

- 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 would want to store these records in the `SALES_INFO` table in a more typical relational format:
`EMPLOYEE_ID, WEEK, SALES`
- Using a **pivoting** INSERT, convert the set of sales records from the nonrelational database table to relational format.

ORACLE

20-15

Copyright © Oracle Corporation, 2001. All rights reserved.

Pivoting INSERT

Pivoting is an operation in which you need to 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.

In order to solve the problem mentioned in the slide, you need to 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 problem statement for a pivoting INSERT statement is specified in the slide. The solution to the preceding problem is shown in 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 created.
```

ORACLE

20-16

Copyright © Oracle Corporation, 2001. All rights reserved.

Pivoting INSERT (continued)

In the example in the slide, the sales data is received from the nonrelational database table SALES_SOURCE_DATA, which is the details of the sales performed by a sales representative on each day of a week, for a week with a particular week ID.

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)

Instructor Note

In order to demonstrate the code example in the slide, you must first run the script files lab\cre_sales_source_data.sql, lab\cre_sales_info.sql and lab\popul_sales_source_data.sql.

Pivoting INSERT (continued)

```
SELECT * FROM SALES_SOURCE_DATA;
```

EMPLOYEE_ID	WEEK_ID	SALES_MON	SALES_TUE	SALES_WED	SALES_THUR	SALES_FRI
176	6	2000	3000	4000	5000	6000

```
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
176	6	2000
176	6	3000
176	6	4000
176	6	5000
176	6	6000

Observe in the preceding example that using a pivoting INSERT, one row from the SALES_SOURCE_DATA table is converted into five records for the relational table, SALES_INFO.

External Tables

- **External tables are read-only tables in which the data is stored outside the database in flat files.**
- **The metadata for an external table is created using a `CREATE TABLE` statement.**
- **With the help of external tables, Oracle data can be stored or unloaded as flat files.**
- **The data can be queried using SQL, but you cannot use DML and no indexes can be created.**

ORACLE

20-18

Copyright © Oracle Corporation, 2001. All rights reserved.

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. Using the Oracle9i external table feature, you can use external data as a virtual table. This data can be queried and joined directly and in parallel without requiring the external data to be first loaded in the database. You can use SQL, PL/SQL, and Java to query the data in an external table.

The main difference between external tables and regular tables is that externally organized tables are read-only. No DML operations (UPDATE, INSERT, or DELETE) are possible, and no indexes can be created on them.

The means of defining the metadata for external tables is through the `CREATE TABLE . . . ORGANIZATION EXTERNAL statement`. 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 Oracle Server provides two major access drivers for external tables. One, the loader access driver, or `ORACLE_LOADER`, is used for reading of data from external files using the Oracle loader technology. This access driver allows the Oracle Server to access data from any data source whose format can be interpreted by the `SQL*Loader` utility. The other Oracle provided access driver, the import/export access driver, or `ORACLE_INTERNAL`, can be used for both the importing and exporting of data using a platform independent format.

Creating an External Table

- Use the `external_table_clause` along with the `CREATE TABLE` syntax to create an external table.
- Specify `ORGANIZATION AS EXTERNAL` to indicate that the table is located outside the database.
- The `external_table_clause` consists of the access driver `TYPE`, `external_data_properties`, and the `REJECT LIMIT`.
- The `external_data_properties` consist of the following:
 - `DEFAULT DIRECTORY`
 - `ACCESS PARAMETERS`
 - `LOCATION`

ORACLE

20-19

Copyright © Oracle Corporation, 2001. All rights reserved.

Creating an External Table

You create external tables 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. The `ORGANIZATION clause` lets you 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.

`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 `REJECT LIMIT` clause lets you 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.

`DEFAULT DIRECTORY` lets you specify one or more default directory objects corresponding to directories on the file system where the external data sources may reside. Default directories can also be used by the access driver to store auxiliary files such as error logs. Multiple default directories are permitted to facilitate load balancing on multiple disk drives.

The optional `ACCESS PARAMETERS` clause lets you assign values to the parameters of the specific access driver for this external table. Oracle does not interpret anything in this clause. It is up to the access driver to interpret this information in the context of the external data.

The `LOCATION clause` lets you specify one external locator for each external data source. Usually the `location_specifier` is a file, but it need not be. Oracle does not interpret this clause. It is up to the access driver to interpret this information in the context of the external data.

Example of Creating an External Table

Create a DIRECTORY object that corresponds to the directory on the file system where the external data source resides.

```
CREATE DIRECTORY emp_dir AS '/flat_files' ;
```

ORACLE

20-20

Copyright © Oracle Corporation, 2001. All rights reserved.

Example of Creating an External Table

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 pathname, for greater file management flexibility.

You must have `CREATE ANY DIRECTORY` system privileges to create directories. When you create a directory, you are automatically granted the `READ` object privilege and can grant `READ` privileges to other users and roles. The DBA can also grant this privilege to other users and roles.

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 regranteeing database object privileges previously granted on the directory. Users who had previously been granted privileges on a redefined directory can still access the directory without being regranteeing the privileges.

directory Specify the name of the directory object to be created. The maximum length of directory is 30 bytes. You cannot qualify a directory object with a schema name.

'path_name' Specify the full pathname of the operating system directory on the result that the path name is case sensitive.

Example of Creating an External Table

```
CREATE TABLE oldemp (  
    empno NUMBER, empname CHAR(20), birthdate DATE)  
    ORGANIZATION EXTERNAL  
    (TYPE ORACLE_LOADER  
    DEFAULT DIRECTORY emp_dir  
    ACCESS PARAMETERS  
    (RECORDS DELIMITED BY NEWLINE  
    BADFILE 'bad_emp'  
    LOGFILE 'log_emp'  
    FIELDS TERMINATED BY ','  
    (empno CHAR,  
    empname CHAR,  
    birthdate CHAR date_format date mask "dd-mon-yyyy"))  
    LOCATION ('emp1.txt'))  
    PARALLEL 5  
    REJECT LIMIT 200;  
Table created.
```

ORACLE

20-21

Copyright © Oracle Corporation, 2001. All rights reserved.

Example of Creating an External Table (continued)

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, and the fields are all terminated by a comma (,). The name of the file is: /flat_files/emp1.txt

To convert this file as the data source for an external table, whose metadata will reside in the database, you need to perform the following steps:

1. Create a directory object emp_dir as follows:

```
CREATE DIRECTORY emp_dir AS '/flat_files' ;
```

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:

```
/flat_files/emp1.txt
```

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 provide values to parameters of the specific access driver and are interpreted by the access driver, not by the Oracle Server.

The PARALLEL clause enables five parallel execution servers to simultaneously scan the external data sources (files) when executing the INSERT INTO TABLE statement. For example, if PARALLEL=5 were specified, then more than one parallel execution server could be working on a data source. Because external tables can be very large, for performance reasons it is advisable to specify the PARALLEL clause, or a parallel hint for the query.

Introduction to Oracle9i: SQL 20-21

Example of Defining External Tables

The `REJECT LIMIT` clause specifies that if more than 200 conversion errors occur during a query of the external data, the query is aborted and an error returned. These conversion errors can arise when the access driver tries to transform the data in the data file to match the external table definition. Once the `CREATE TABLE` command executes successfully, the external table `OLDEMP` can be described and queried like a relational table.

```
DESC oldemp
```

Name	Null?	Type
EMPNO		NUMBER
EMPNAME		CHAR(20)
BIRTHDATE		DATE

In the following example, the `INSERT INTO TABLE` statement generates a dataflow from the external data source to the Oracle SQL engine where data is processed. As data is extracted from the external table, it is transparently converted by the `ORACLE_ LOADER` access driver from its external representation into an equivalent Oracle native representation. The `INSERT` statement inserts data from the external table `OLDEMP` into the `BIRTHDAYS` table:

```
INSERT INTO birthdays(empno, empname, birthdate)
      SELECT empno, empname, birthdate
      FROM   oldemp;
```

2 rows created.

We can now select from the `BIRTHDAYS` table.

```
SELECT * FROM birthdays;
```

EMPNO	EMPNAME	BIRTHDATE
10	jones	11-DEC-34
20	smith	12-JUN-97

Instructor Note

To run the code example in the slide, do the following:

1. Login to unix teach account and type the following:

```
cd FLAT_FILES
```

```
pwd
```

The output should resemble `/home#/teach#/FLAT_FILES`

2. Open the file `cre_dir.sql` from the lab folder and replace the last command in the file with the output from the unix `pwd`.

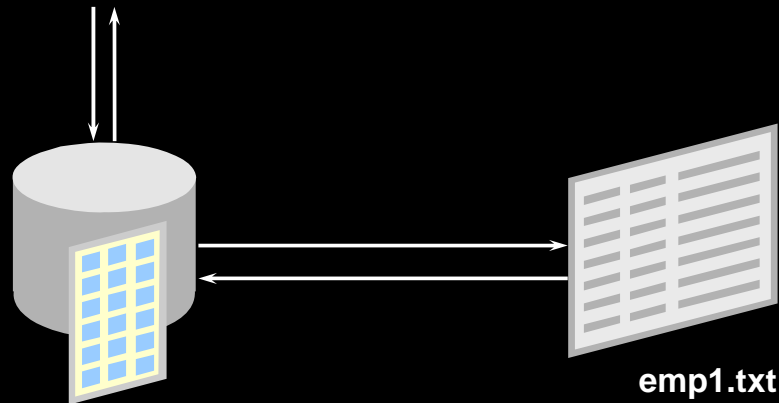
The last command in the file will now look like this:

```
CREATE OR REPLACE emp_dir as '<output from unix pwd>';
```

3. Save the file `cre_dir.sql` and execute this file in *iSQL*Plus*
4. Run the `cre_birthdays.sql` script to create the `BIRTHDAYS` table.

Querying External Tables

```
SELECT *  
FROM oldemp
```



ORACLE

20-23

Copyright © Oracle Corporation, 2001. All rights reserved.

Querying External Table

An external table does not describe any data that is stored in the database. Nor does it describe how data is stored in the external source. Instead, it describes how the external table layer needs to 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 needs to access data in an external source, it calls the appropriate access driver to get the data from an external source in a form that the database server expects.

It is important to remember that the description of the 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 the data from the data source is processed so that it matches the definition of the external table.

CREATE INDEX with 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 created.
```

```
SELECT INDEX_NAME, TABLE_NAME
FROM    USER_INDEXES
WHERE   TABLE_NAME = 'NEW_EMP';
```

INDEX_NAME	TABLE_NAME
EMP_ID_IDX	NEW_EMP

ORACLE

20-24

Copyright © Oracle Corporation, 2001. All rights reserved.

CREATE INDEX with CREATE TABLE Statement

In the example in the slide, the **CREATE INDEX clause** is used with the **CREATE TABLE** statement to create a primary key index explicitly. This is an enhancement provided with Oracle9i. You can now name your indexes at the time of **PRIMARY** key creation, unlike before where the Oracle Server would create an index, but you did not have any control over the name of the index. The following example illustrates this:

```
CREATE TABLE EMP_UNNAMED_INDEX
(employee_id NUMBER(6) PRIMARY KEY ,
first_name VARCHAR2(20),
last_name VARCHAR2(25));
```

Table created.

```
SELECT INDEX_NAME, TABLE_NAME
FROM    USER_INDEXES
WHERE   TABLE_NAME = 'EMP_UNNAMED_INDEX';
```

INDEX_NAME	TABLE_NAME
SYS_C002835	EMP_UNNAMED_INDEX

Observe that the Oracle Server gives a name to the Index that it creates for the **PRIMARY KEY** column. But this name is cryptic and not easily understood. With Oracle9i, you can name your **PRIMARY KEY** column indexes, as you create the table with the **CREATE TABLE** statement. However, prior to Oracle9i, if you named your primary key constraint at the time of constraint creation, the index would also be created with the same name as the constraint name.

Introduction to Oracle9i: SQL 20-24

Summary

In this lesson, you should have learned how to:

- Use the `INSERT...SELECT` statement to insert rows into multiple tables as part of a single DML statement
- Create external tables
- Name indexes using the `CREATE INDEX` statement along with the `CREATE TABLE` statement

ORACLE

20-25

Copyright © Oracle Corporation, 2001. All rights reserved.

Summary

Oracle 9i introduces the following types of multitable `INSERT` statements.

- Unconditional `INSERT`
- Conditional `ALL INSERT`
- Conditional `FIRST INSERT`
- Pivoting `INSERT`

Use the `external_table_clause` to create an external table, which is a read-only table whose metadata is stored in the database but whose data is stored outside the database. You can use external tables to query data without first loading it into the database.

With Oracle9i, you can name your `PRIMARY KEY` column indexes as you create the table with the `CREATE TABLE` statement.

Practice 20 Overview

This practice covers the following topics:

- **Writing unconditional `INSERT` statements**
- **Writing conditional `ALL INSERT` statements**
- **Pivoting `INSERT` statements**
- **Creating indexes along with the `CREATE TABLE` command**

ORACLE

20-26

Copyright © Oracle Corporation, 2001. All rights reserved.

Practice 20 Overview

In this practice, you write multitable inserts and use the `CREATE INDEX` command at the time of table creation, along with the `CREATE TABLE` command.

Practice 20

1. Run the `cre_sal_history.sql` script in the lab folder to create the `SAL_HISTORY` table.
2. Display the structure of the `SAL_HISTORY` table.

Name	Null?	Type
EMPLOYEE_ID		NUMBER(6)
HIRE_DATE		DATE
SALARY		NUMBER(8,2)

3. Run the `cre_mgr_history.sql` script in the lab folder to create the `MGR_HISTORY` table.
4. Display the structure of the `MGR_HISTORY` table.

Name	Null?	Type
EMPLOYEE_ID		NUMBER(6)
MANAGER_ID		NUMBER(6)
SALARY		NUMBER(8,2)

5. Run the `cre_special_sal.sql` script in the lab folder to create the `SPECIAL_SAL` table.
6. Display the structure of the `SPECIAL_SAL` table.

Name	Null?	Type
EMPLOYEE_ID		NUMBER(6)
SALARY		NUMBER(8,2)

7. a. Write a query to do the following:
 - Retrieve the details of the employee ID, hire date, salary, and manager ID of those employees whose employee ID is less than 125 from the `EMPLOYEES` table.
 - If the salary is more than \$20,000, insert the details of employee ID and salary into the `SPECIAL_SAL` table.
 - Insert the details of employee ID, hire date, salary into the `SAL_HISTORY` table.
 - Insert the details of the employee ID, manager ID, and salary into the `MGR_HISTORY` table.

Practice 20 (continued)

b. Display the records from the SPECIAL_SAL table.

EMPLOYEE_ID	SALARY
100	24000

c. Display the records from the SAL_HISTORY table.

EMPLOYEE_ID	HIRE_DATE	SALARY
101	21-SEP-89	17000
102	13-JAN-93	17000
103	03-JAN-90	9000
104	21-MAY-91	6000
107	07-FEB-99	4200
124	16-NOV-99	5800

6 rows selected.

d. Display the records from the MGR_HISTORY table.

EMPLOYEE_ID	MANAGER_ID	SALARY
101	100	17000
102	100	17000
103	102	9000
104	103	6000
107	103	4200
124	100	5800

6 rows selected.

Practice 20 (continued)

- 8.a. Run the `cre_sales_source_data.sql` script in the lab folder to create the `SALES_SOURCE_DATA` table.
- b. Run the `ins_sales_source_data.sql` script in the lab folder to insert records into the `SALES_SOURCE_DATA` table.
- c. Display the structure of the `SALES_SOURCE_DATA` table.

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)

- d. Display the records from the `SALES_SOURCE_DATA` table.

EMPLOYEE_ID	WEEK_ID	SALES_MON	SALES_TUE	SALES_WED	SALES_THUR	SALES_FRI
178	6	1750	2200	1500	1500	3000

- e. Run the `cre_sales_info.sql` script in the lab folder to create the `SALES_INFO` table.
- f. Display the structure of the `SALES_INFO` table.

Name	Null?	Type
EMPLOYEE_ID		NUMBER(6)
WEEK		NUMBER(2)
SALES		NUMBER(8,2)

Practice 20 (continued)

g. Write a query to do the following:

Retrieve the details of employee ID, week ID, sales on Monday, sales on Tuesday, sales on Wednesday, sales on Thursday, and sales on Friday from the SALES_SOURCE_DATA table.

Build a transformation such that each record retrieved from the SALES_SOURCE_DATA table is converted into multiple records for the SALES_INFO table.

Hint: Use a pivoting INSERT statement.

h. Display the records from the SALES_INFO table.

EMPLOYEE_ID	WEEK	SALES
178	6	1750
178	6	2200
178	6	1500
178	6	1500
178	6	3000

9. a. Create the DEPT_NAMED_INDEX table based on the following table instance chart. Name the index for the PRIMARY KEY column as DEPT_PK_IDX.

COLUMN Name	Deptno	Dname
Primary Key	Yes	
Datatype	Number	VARCHAR2
Length	4	30

b. Query the USER_INDEXES table to display the INDEX_NAME for the DEPT_NAMED_INDEX table.

INDEX_NAME	TABLE_NAME
DEPT_PK_IDX	DEPT_NAMED_INDEX