Introduction to Oracle: SQL and PL/SQL

Student Guide • Volume 1

ORACLE®

Authors

Neena Kochhar
Ellen Gravina
Priya Nathan

Technical Contributors
and Reviewers

Claire Bennet
Christa Miethaner
Tony Hickman
Sherin Nassa
Nancy Greenberg
Hazel Russell
Kenneth Goetz
Piet van Zon
Ulrike Dietrich
Helen Robertson
Thomas Nguyen
Lisa Jansson
Kuljit Jassar

Publisher

Jerry Brosnan

Contents

Preface

Profile

Before You Begin This Course

Before you begin this course, you should be able to use a graphical user interface (GUI). Required prerequisites are familiarity with data processing concepts and techniques.

How This Course Is Organized

Introduction to Oracle: SQL and PL/SQL is an instructor-led course featuring lectures and hands-on exercises. Online demonstrations and written practice sessions reinforce the concepts and skills introduced.

Related Publications

Oracle Publications

Title	Part Number
Oracle8i Server, Release 8.1.5	A68826-01
Oracle8i Concepts, Release 8.1.5	A67781-01
Oracle8i SQL Reference Manual, Release 8.1.5	A67779-01
Oracle8i Server Application Developer's Guide	A68003-01
SQL*Plus User's Guide and Reference, Release 8.1.5	A66736-01
SQL*Plus Quick Reference, Release 8.1.5	A66735-01
PL/SQL User's Guide and Reference, Release 8.1.5	A67842-01

Additional Publications

- System release bulletins
- Installation and user's guides
- *read.me* files
- International Oracle User's Group (IOUG) articles
- *Oracle Magazine*

Typographic Conventions

What follows are two lists of typographical conventions used specifically within text or within code.

Typographic Conventions Within Text

Convention	Object or Term	Example
Uppercase	Commands, functions, column names, table names, PL/SQL objects, schemas	Use the SELECT command to view information stored in the LAST_NAME column of the EMP table.
Lowercase, italic	Filenames, syntax variables, usernames, passwords	**where:** *role* is the name of the role to be created.
Initial cap	Trigger and button names	Assign a When-Validate-Item trigger to the ORD block. Choose Cancel.
Italic	Books, names of courses and manuals, and emphasized words or phrases	For more information on the subject see *Oracle Server SQL Language Reference Manual* Do *not* save changes to the database.
Quotation marks	Lesson module titles referenced within a course	This subject is covered in Lesson 3, "Working with Objects."
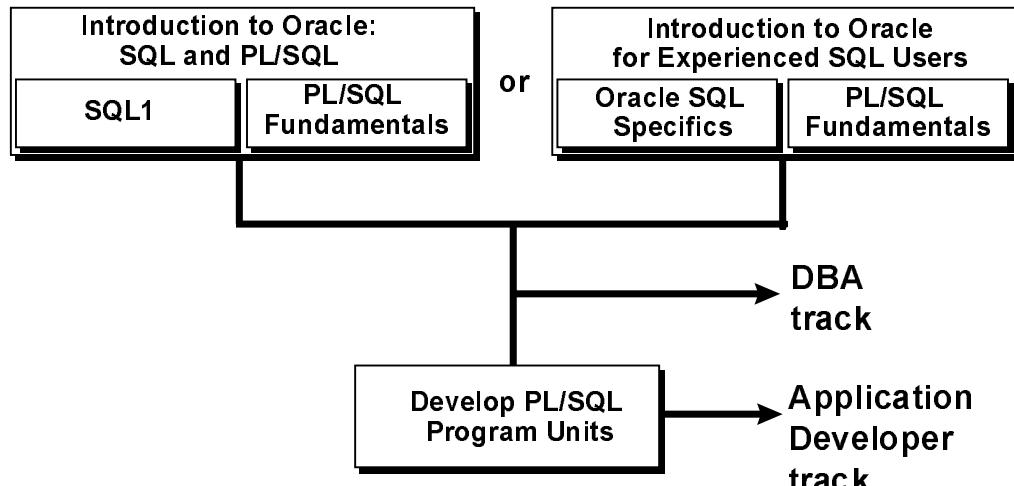
Typographic Conventions (continued)

Typographic Conventions Within Code

Convention	Object or Term	Example
Uppercase	Commands, functions	`SQL>  SELECT userid` `  2    FROM  emp;`
Lowercase, italic	Syntax variables	`SQL>  CREATE ROLE role;`

Initial cap	Forms triggers	`Form module: ORD`
		`Trigger level: S_ITEM.QUANTITY`
		`item`
		`Trigger name: When-Validate-Item`
		`. . .`

Lowercase	Column names, table names, filenames, PL/SQL objects	`. . .`
		`OG_ACTIVATE_LAYER`
		`(OG_GET_LAYER ('prod_pie_layer'))`
		`. . .`
		
		`SQL>  SELECT last_name`
		`  2    FROM emp;`

Bold	Text that must be entered by a user	`SQLDBA>  DROP USER scott` `     2>   IDENTIFIED BY tiger;`

Curriculum Map

Integrated Languages Curriculum: Certification Tracks

Introduction to Oracle: SQL and PL/SQL		or	Introduction to Oracle for Experienced SQL Users	
SQL1	PL/SQL Fundamentals		Oracle SQL Specifics	PL/SQL Fundamentals

→ DBA track

Develop PL/SQL Program Units → Application Developer track
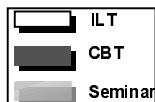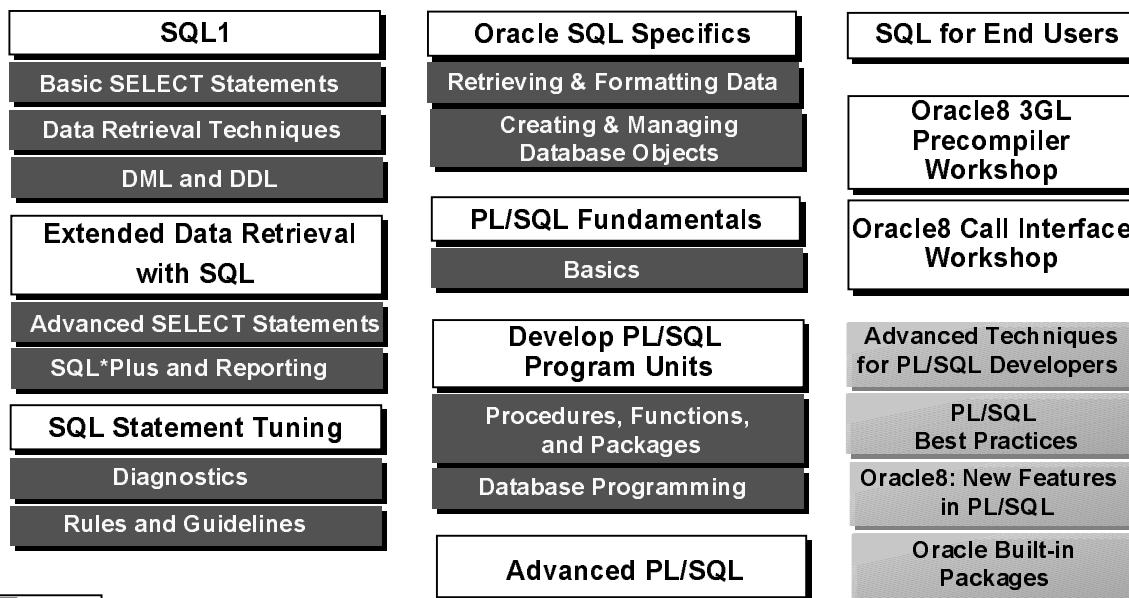
Integrated Languages Curriculum: Certification Tracks

Introduction to Oracle: SQL and PL/SQL consists of two modules, *SQL1* and *PL/SQL Fundamentals*. This is the first course that you take for the DBA or Application Developer track. *SQL1* covers creating database structures and storing, retrieving, and manipulating data in a relational database. *PL/SQL Fundamentals* covers creating PL/SQL blocks of application code.

For people who have worked with other relational databases and have knowledge of SQL, another module called *Oracle SQL Specifics* is offered. This module covers the SQL statements that are not part of ANSI SQL but are specific to Oracle. This module combined with *PL/SQL Fundamentals* forms *Introduction to Oracle for Experienced SQL Users*.

Introduction to Oracle: SQL and PL/SQL and *Introduction to Oracle for Experienced SQL Users* are considered equivalent and after taking one of them, you can move on to the DBA track. For the Application Developer track, you must take the *Develop PL/SQL Program Units* course. This course teaches you how to write PL/SQL procedures, functions, packages, and triggers.

Integrated Languages Curriculum

SQL1
Basic SELECT Statements
Data Retrieval Techniques
DML and DDL

Extended Data Retrieval with SQL
Advanced SELECT Statements
SQL*Plus and Reporting

SQL Statement Tuning
Diagnostics
Rules and Guidelines

Oracle SQL Specifics
Retrieving & Formatting Data
Creating & Managing Database Objects

PL/SQL Fundamentals
Basics

Develop PL/SQL Program Units
Procedures, Functions, and Packages
Database Programming

Advanced PL/SQL

SQL for End Users

Oracle8 3GL Precompiler Workshop

Oracle8 Call Interface Workshop

Advanced Techniques for PL/SQL Developers

PL/SQL Best Practices

Oracle8: New Features in PL/SQL

Oracle Built-in Packages

- ILT
- CBT
- Seminar

ORACLE

Integrated Languages Curriculum

The slide lists various modules and courses that are available in the languages curriculum. For most of these modules and courses, there are equivalent CBTs.

Course or Module	Equivalent CBT
SQL1	Oracle SQL: Basic SELECT Statements Oracle SQL: Data Retrieval Techniques Oracle SQL: DML and DDL
Oracle SQL Specifics	Oracle SQL Specifics: Retrieving and Formatting Data Oracle SQL Specifics: Creating and Managing Database Objects
PL/SQL Fundamentals	PL/SQL: Basics
Extended Data Retrieval with SQL	Oracle SQL and SQL*Plus: Advanced SELECT Statements Oracle SQL and SQL*Plus: SQL*Plus and Reporting
Develop PL/SQL Program Units	PL/SQL: Procedures, Functions, and Packages PL/SQL: Database Programming
SQL Statement Tuning	SQL and PL/SQL Tuning: Diagnostics SQL and PL/SQL Tuning: Rules and Guidelines

Four seminars on PL/SQL are also offered: *Advanced Techniques for PL/SQL Developers, PL/SQL Best Practices, Oracle8: New Features in PL/SQL,* and *Oracle Built-in Packages.*

Introduction

ORACLE®

Objectives

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

- **Discuss the theoretical and physical aspects of a relational database**

- **Describe the Oracle implementation of the RDBMS and ORDBMS**

- **Describe new features of Oracle8*i***

- **Describe how SQL and PL/SQL are used in the Oracle product set**

- **Describe the use and benefits of PL/SQL**
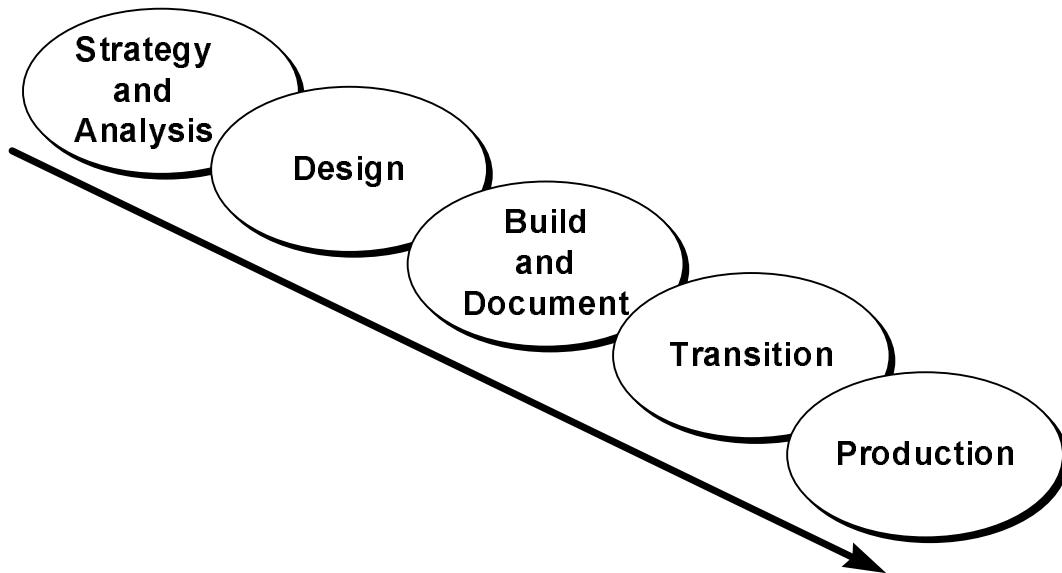
Lesson Aim

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

- SQL statements that are specific to Oracle
- SQL*Plus, which is used for executing SQL and for formatting and reporting purposes
- PL/SQL, which is Oracle's procedural language

System Development Life Cycle

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.
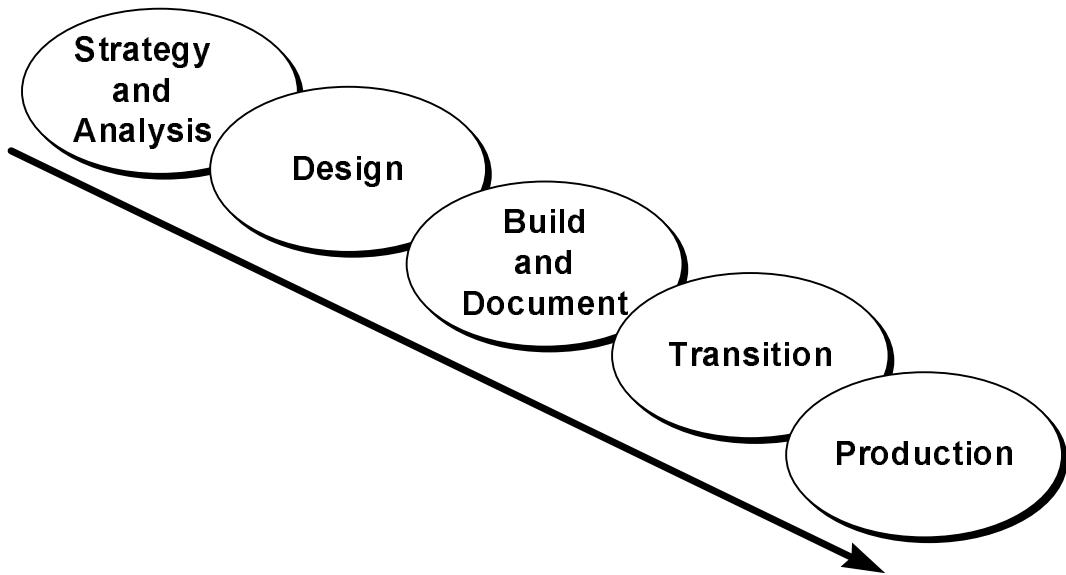
Design

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

System Development Life Cycle (continued)

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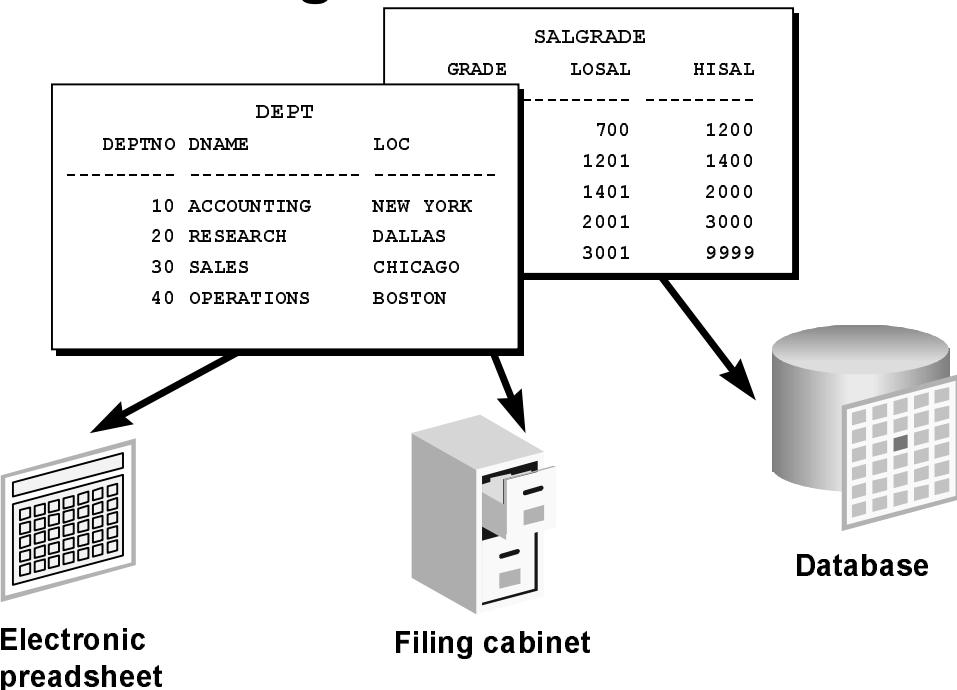
Refine the prototype. Move an application into production with user acceptance testing, conversion of existing data, and parallel operations. Make any modifications required.

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 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

```
                              SALGRADE
                    GRADE     LOSAL      HISAL
              DEPT                     --------- ---------
 DEPTNO DNAME          LOC                700       1200
--------- -------------- ----------      1201      1400
     10 ACCOUNTING    NEW YORK           1401      2000
     20 RESEARCH      DALLAS             2001      3000
     30 SALES         CHICAGO            3001      9999
     40 OPERATIONS    BOSTON
```

Database

**Electronic
spreadsheet**

Filing cabinet

ORACLE®

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—for example, 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*.

Note: Oracle7 is a relational database management system and Oracle8 is an object relational database management system.

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**

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.
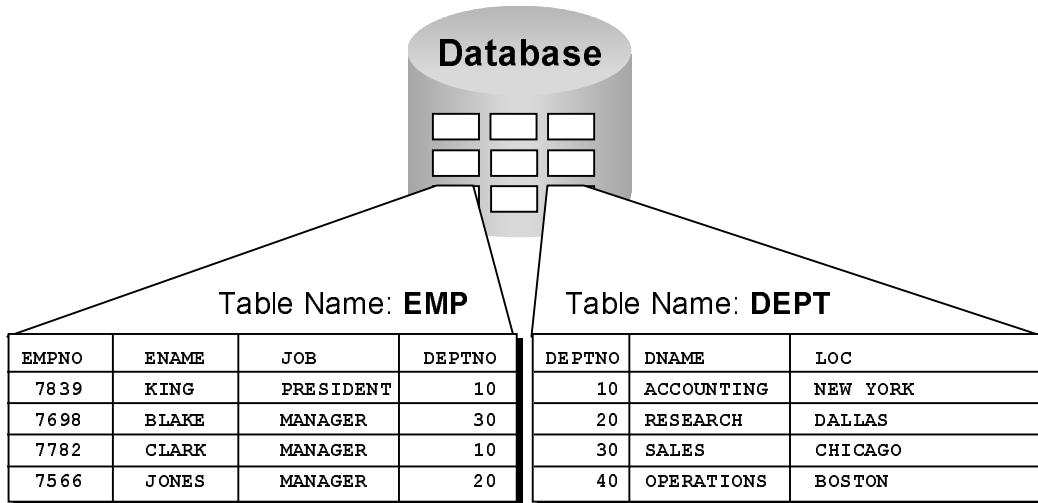
Components of the Relational Model

- 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.

Database

Table Name: **EMP**

EMPNO	ENAME	JOB	DEPTNO
7839	KING	PRESIDENT	10
7698	BLAKE	MANAGER	30
7782	CLARK	MANAGER	10
7566	JONES	MANAGER	20

Table Name: **DEPT**

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

ORACLE®
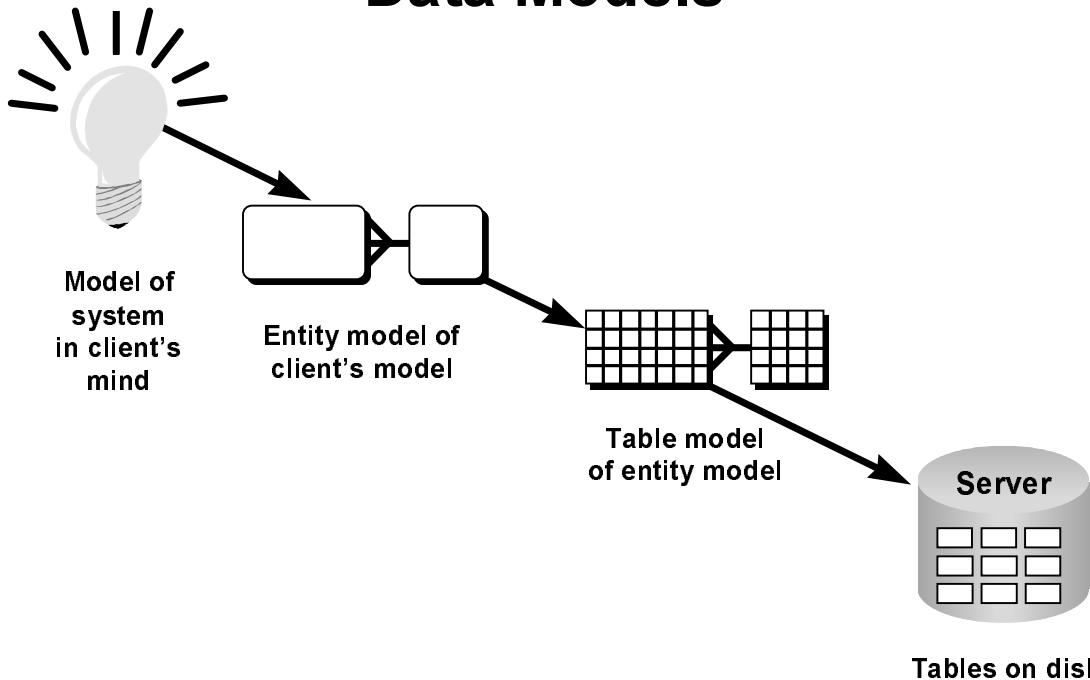
Definition of a Relational Database

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

Model of system in client's mind → **Entity model of client's model** → **Table model of entity model** → **Server** **Tables on disk**

ORACLE®

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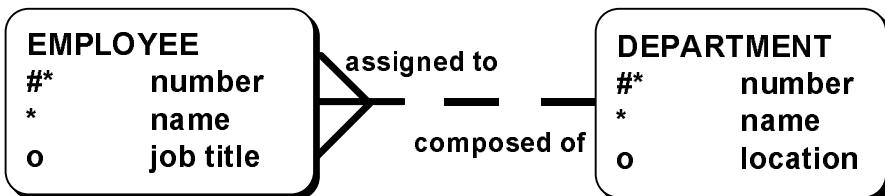
Models help communicate the concepts in people's minds. They can be used to do the following:

- 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**

EMPLOYEE		DEPARTMENT	
#*	number	#*	number
*	name	*	name
o	job title	o	location

assigned to

composed of

- **Scenario**
  - "... Assign one or more employees to a department ..."
  - "... Some departments do not yet have assigned employees ..."

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"

```
EMPLOYEE                    DEPARTMENT
#*      number              #*      number
*       name                *       name
o       job title           o       location
```

assigned to

composed of

Unique Identifier (UID)
Primary marked with "#"
Secondary marked with "(#)"
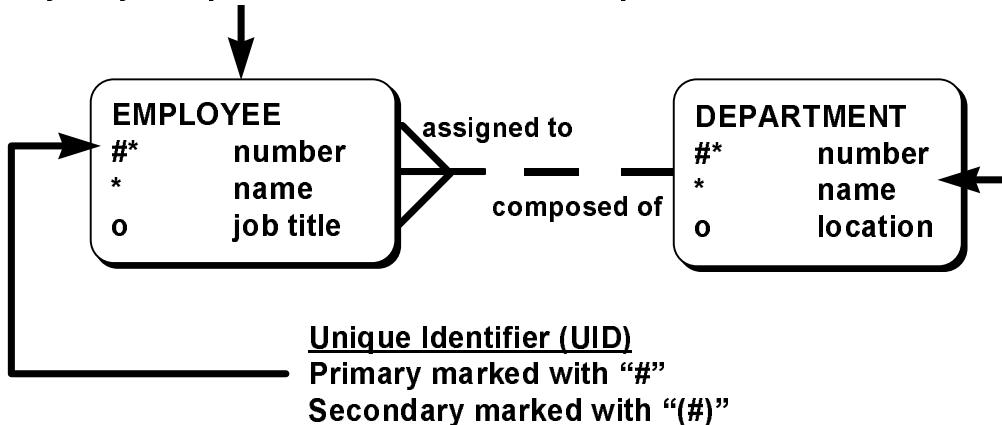
Entities

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

<u>Entity</u>
Soft box
Singular, unique name
Uppercase
Synonym in parentheses

<u>Attribute</u>
Singular name
Lowercase
Mandatory marked with "*"
Optional marked with "o"

```
EMPLOYEE                          DEPARTMENT
#*      number     assigned to    #*      number
*       name                      *       name
o       job title  composed of    o       location
```

<u>Unique Identifier (UID)</u>
Primary marked with "#"
Secondary marked with "(#)"
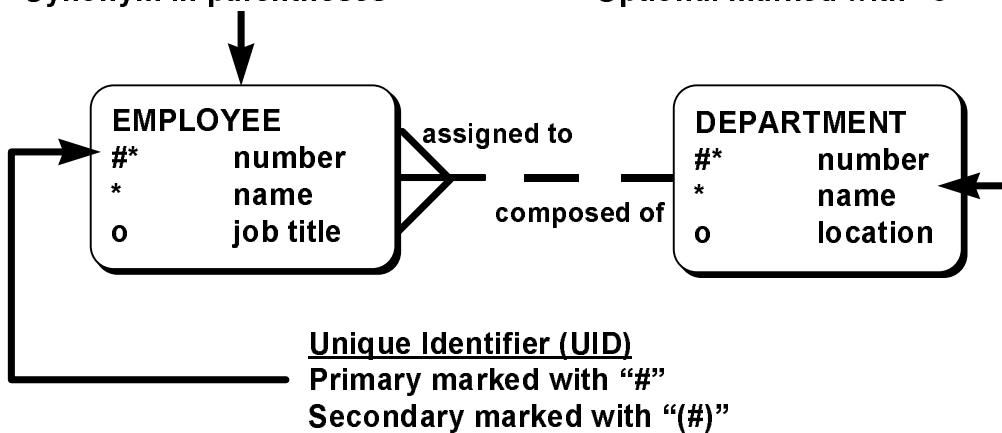
     ORACLE®

Relationships

Each direction of the relationship contains:

- A name, 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: (#)

Relational Database Terminology

	EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
	7839	KING	PRESIDENT		17-NOV-81	5000		10
	7698	BLAKE	MANAGER	7839	01-MAY-81	2850		30
	7782	CLARK	MANAGER	7839	09-JUN-81	2450		10
	7566	JONES	MANAGER	7839	02-APR-81	2975		20
	7654	MARTIN	SALESMAN	7698	28-SEP-81	1250	1400	30
	7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	300	30
	7844	TURNER	SALESMAN	7698	08-SEP-81	1500	0	30
	7900	JAMES	CLERK	7698	03-DEC-81	950		30
	7521	WARD	SALESMAN	7698	22-FEB-81	1250	500	30
	7902	FORD	ANALYST	7566	03-DEC-81	3000		20
	7369	SMITH	CLERK	7902	17-DEC-80	800		20
	7788	SCOTT	ANALYST	7566	09-DEC-82	3000		20
	7876	ADAMS	CLERK	7788	12-JAN-83	1100		20
	7934	MILLER	CLERK	7782	23-JAN-82	1300		10

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—for example, employees, invoices, or customers.

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

1. A single *row* or tuple 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, which is also the primary key. The employee number identifies a *unique* employee in the EMP table. A primary key must contain a value.

3. A column that is not a key value. A column represents one kind of data in a table; in the example, the job title 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 another table. In the example, DEPTNO *uniquely* identifies a department in the DEPT table.

5. A *field* can be found at the intersection of a row and a column. There can be only one value in it.

6. A field may have no value in it. This is called a null value. In the EMP table, only employees who have a role of salesman have a value in the COMM (commission) field.

Note: Null values are covered further in subsequent lessons.

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: **EMP**   Table Name: **DEPT**

EMPNO	ENAME	JOB	DEPTNO
7839	KING	PRESIDENT	10
7698	BLAKE	MANAGER	30
7782	CLARK	MANAGER	10
7566	JONES	MANAGER	20

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

↑ **Primary key**   ↑ **Foreign key**   ↑ **Primary key**

ORACLE®

Relating Multiple Tables

Each table contains data that describes exactly one entity. For example, the EMP 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 EMP table (which contains data about employees) and the DEPT table (which contains information about departments). An RDBMS enables you to 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.

The ability to relate data in one table to data in another enables you 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

- No duplicate values are allowed 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.
- You cannot define foreign keys without existing primary (unique) keys.

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**
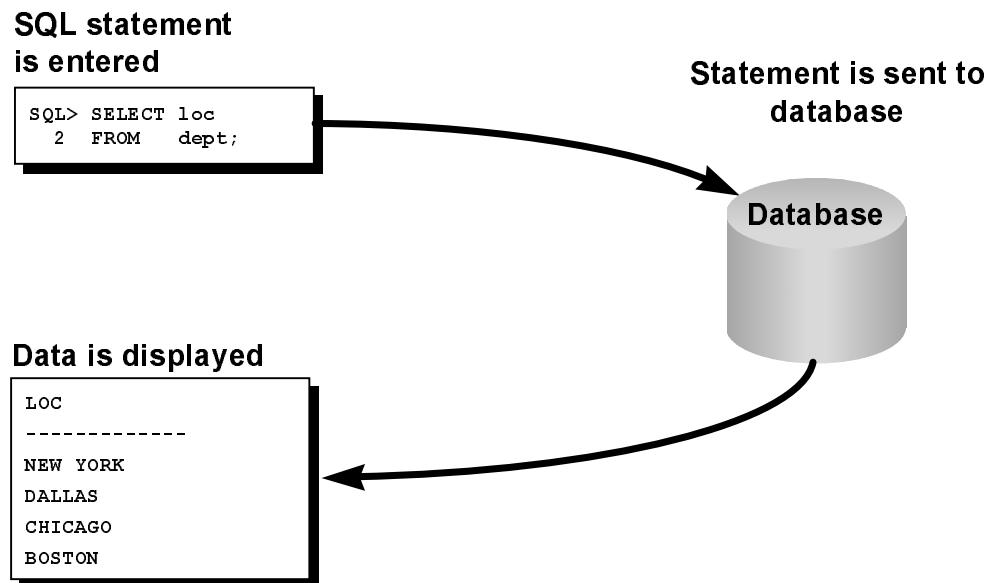
ORACLE®

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 upon 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

```
SQL> SELECT loc
  2  FROM   dept;
```

Statement is sent to database

Database

Data is displayed

```
LOC
-------------
NEW YORK
DALLAS
CHICAGO
BOSTON
```
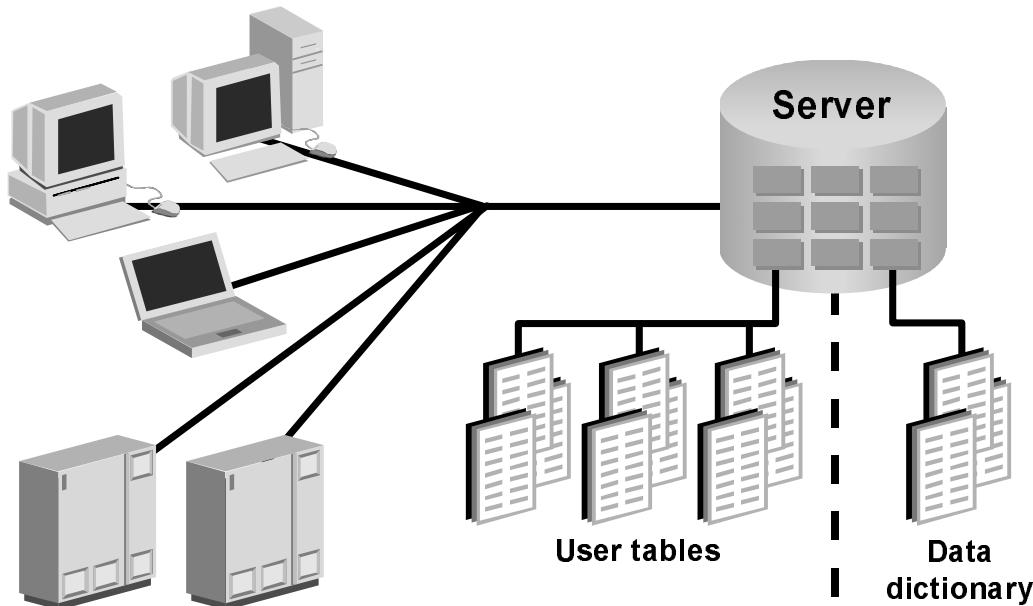
ORACLE®

Structured Query Language

SQL allows you to communicate with the server and has the following advantages:

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

Relational Database Management System

Server

User tables

Data dictionary

ORACLE®

Relational Database Management System

Oracle provides a flexible RDBMS called Oracle7. 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. The 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.

Oracle applications may run on the same computer as the Oracle Server. Alternatively, you can run applications on a system local to the user and run the Oracle Server on another system (client-server architecture). In this client-server environment, a wide range of computing resources can be used. For example, a form-based airline reservation application can run on a client personal computer while accessing flight data that is conveniently managed by an Oracle Server on a central computer.

For more information, see *Oracle Server Concepts Manual,* Release 8.

Oracle8: Object Relational Database Management System

- **User-defined datatypes and objects**
- **Fully compatible with relational database**
- **Support of multimedia and large objects**
- **High-quality database server features**

About Oracle8

Oracle8 is the first object-capable database developed by Oracle. It extends the data modeling capabilities of Oracle7 to support a new object relational database model. Oracle8 provides a new engine that brings object-oriented programming, complex datatypes, complex business objects, and full compatibility with the relational world.

Oracle8 extends Oracle7 in many ways. It includes several features for improved performance and functionality of online transaction processing (OLTP) applications, such as better sharing of runtime 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, Oracle8 supports client-server and Web-based applications that are distributed and multitiered.

Oracle8 can scale tens of thousands of concurrent users, support up to 512 petabytes, 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 *Oracle Server Concepts Manual,* Release 8.

Oracle8*i*: Internet Platform

Database for Internet Computing Features

- **Advanced tools to manage all types of data on Web sites**

- **More than a simple relational data store: *i*FS**

- **Integrated Java VM in the server: JServer**

- **Better performance, stronger security, language improvement**

- **Greater integration with Windows NT environment: AppWizard**

About Oracle8*i*

Oracle8*i*, the database for Internet computing, provides advanced tools to manage all types of data in Web sites.

It is much more than a simple relational data store. The Internet File System (*i*FS) combines the power of Oracle8*i* with the ease of use of a file system. It allows users to move all of their data into the Oracle8*i* database, where it can be stored and managed more efficiently. End users can easily access files and folders in Oracle *i*FS via a variety of protocols, such as HTML, FTP, and IMAP4, giving them universal access to their data.
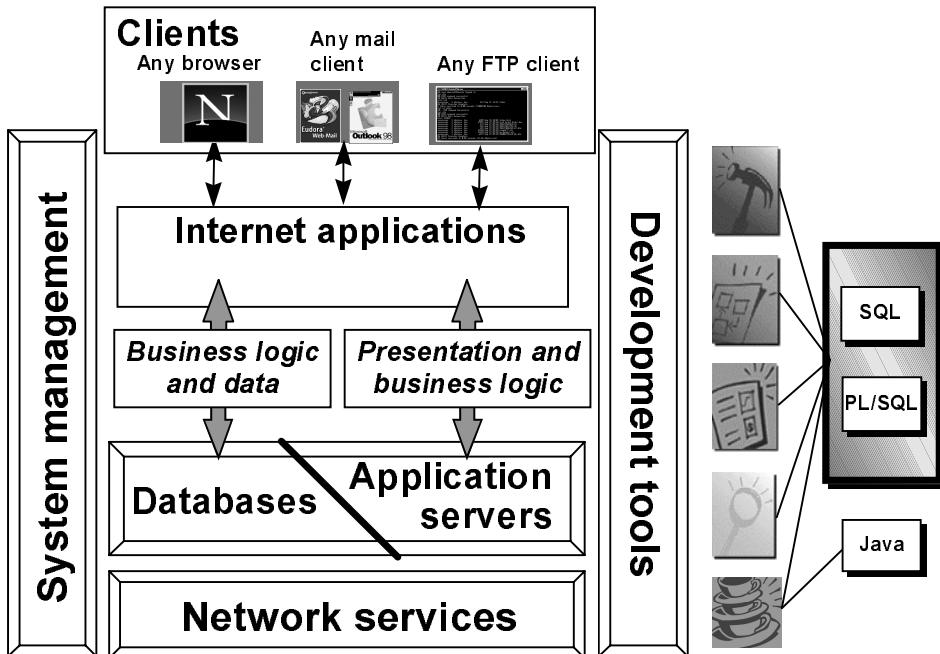
Oracle8*i* *inter*Media allows users to web-enable their multi-media data—including image, text, audio, and video data. Oracle8*i* includes a robust, integrated, and scalable Java Virtual Machine within the server (Jserver), thus supporting Java in all tiers of applications. This eliminates the necessity of recompiling or modifying Java code when it is to be deployed on a different tier.

With the newly introduced resource management, the DBA can choose the best method to fit an application's profile and workload. The extended features of parallel server and networking improves ease of system administration. The extended functionality of advanced replication results in better performance and improved security. Significant new functionalities have been added to languages.

Oracle8*i* provides full, native integration with Microsoft Transaction Server (MTS) in the Windows NT environment. Application development is simplified by the Oracle Application Wizard (AppWizard) for Visual Studio, which provides developers with a GUI tool for creating a Visual C++, Visual Interdev, or Visual Basic applications accessing data in an Oracle database.

For more information, see *Oracle Server Concepts Manual,* Release 8*i*.

Oracle Internet Platform

Clients

Any browser　　Any mail client　　Any FTP client

Internet applications

Business logic and data	Presentation and business logic

Databases　　Application servers

Network services

System management

Development tools

SQL

PL/SQL

Java

ORACLE®

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.

SQL Statements

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

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	Enters new rows, changes existing rows, and removes unwanted rows from tables in the database, respectively. Collectively known as *data manipulation language* (DML).
CREATE ALTER DROP RENAME TRUNCATE	Sets up, changes, and removes data structures from tables. Collectively known as *data definition language* (DDL).
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 *data control language* (DCL).

About PL/SQL

- **PL/SQL is an extension to SQL with design features of programming languages.**

- **Data manipulation and query statements of SQL are included within procedural units of code.**
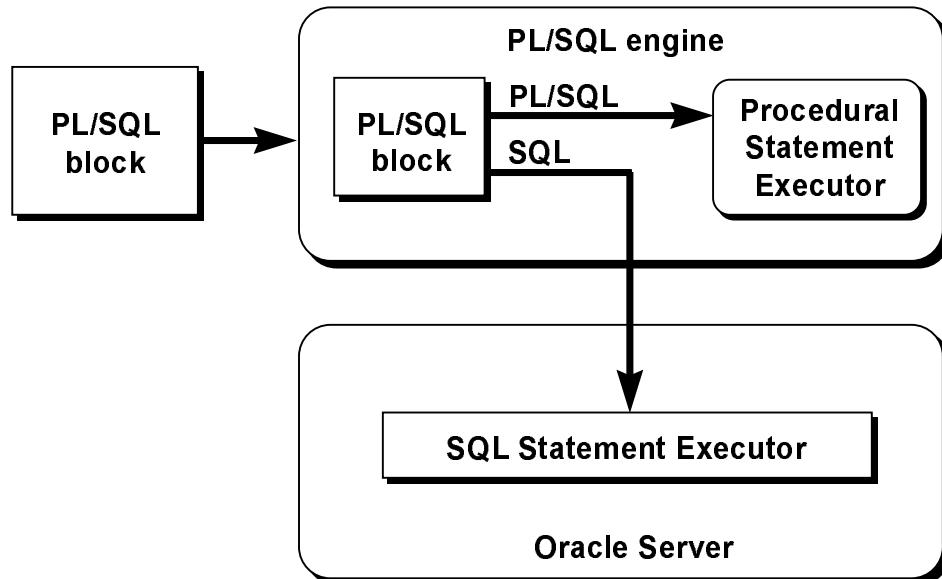
About PL/SQL

Procedural Language/SQL (PL/SQL) is Oracle Corporation's procedural language extension to SQL, the standard data access language for relational databases. PL/SQL offers modern software engineering features such as data encapsulation, exception handling, information hiding, and object orientation, and so brings state-of-the-art programming to the Oracle Server and Toolset.

PL/SQL incorporates many of the advanced features made in programming languages designed during the 1970s and 1980s. It allows the data manipulation and query statements of SQL to be included in block-structured and procedural units of code, making PL/SQL a powerful transaction processing language. With PL/SQL, you can use SQL statements to finesse Oracle data and PL/SQL control statements to process the data.

PL/SQL Environment

```
                            PL/SQL engine

  ┌──────────┐          ┌──────────┐  PL/SQL    ┌──────────────┐
  │  PL/SQL  │─────────▶│  PL/SQL  │───────────▶│  Procedural  │
  │  block   │          │  block   │            │  Statement   │
  └──────────┘          └──────────┘  SQL       │  Executor    │
                                                └──────────────┘
                                        │
                                        ▼
                        ┌────────────────────────────────┐
                        │    SQL Statement Executor       │
                        └────────────────────────────────┘
                               Oracle Server
```

     ORACLE®

PL/SQL Engine and the Oracle Server

PL/SQL is not an Oracle product in its own right; it is a technology employed by the Oracle Server and by certain Oracle tools. Blocks of PL/SQL are passed to and processed by a PL/SQL engine, which may reside within the tool or within the Oracle Server. The engine used depends on where the PL/SQL is being invoked.

When you submit PL/SQL blocks from a Pro* program, user-exit, SQL*Plus, or Server Manager, the PL/SQL engine in the Oracle Server processes them. It separates out the SQL statements and sends them individually to the SQL statements executor.

A single transfer is required to send the block from the application to the Oracle Server, thus improving performance, especially in a client-server network. PL/SQL code can also be stored in the Oracle Server as subprograms that can be referenced by any number of applications connected to the database.

Tables Used in the Course

EMP

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7839	KING	PRESIDENT		17-NOV-81	5000		10
7698	BLAKE	MANAGER	7839	01-MAY-81	2850		30
7782	CLARK	MANAGER	7839	09-JUN-81	2450		10
7566	JONES	MANAGER	7839	02-APR-81	2975		20
7654	MARTIN	SALESMAN	7698	28-SEP-81	1250	1400	30
7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	300	30
7844	TURNER	SALESMAN	7698	08-SEP-81	1500	0	30
7900	JAMES	CLERK	7698	03-DEC-81	950		30
7698				22-FEB-81	1250	500	30
7566				03-DEC-81	3000		20
7902				17-DEC-80			
7566				09-DEC-82			
7788				12-JAN-83			
7782				23-JAN-82			

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

DEPT

GRADE	LOSAL	HISAL
1	700	1200
2	1201	1400
3	1401	2000
4	2001	3000
5	3001	9999

SALGRADE

ORACLE®

Tables Used in the Course

The following three tables will be used in this course:

- EMP table, which gives details of all the employees
- DEPT table, which gives details of all the departments
- SALGRADE table, which gives details of salaries for various grades

The structure and data for all the tables is given in Appendix B.

Summary

- **Relational databases are composed of relations, managed by relational operations, and governed by data integrity constraints.**
- **The Oracle Server allows you to store and manage information by using the SQL language and PL/SQL engine.**
- **Oracle8 is based on the object relational database management system.**
- **The Oracle8*i* Server is the database for Internet computing.**
- **PL/SQL is an extension to SQL with design features of programming languages.**

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 product is the Oracle Server, which enables you to store and manage information by using SQL and the PL/SQL engine for procedural constructs.

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.

PL/SQL

The PL/SQL language extends the SQL language by offering block-structured procedural constructs combined with SQL nonprocedural capabilities.

1

Writing Basic
SQL Statements

ORACLE®

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 SQL*Plus commands**

ORACLE®

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 that you need to perform these actions.

You may want to create SELECT statements that can be used time and time again. This lesson also covers the use of SQL*Plus commands to execute SQL statements.

Capabilities of SQL SELECT Statements

Selection
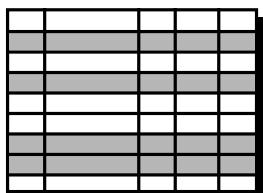
Table 1

Projection
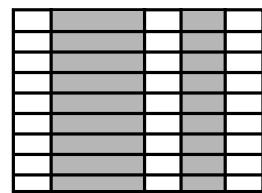
Table 1

Join
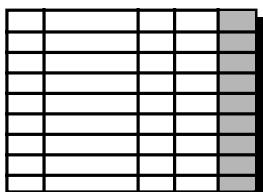
Table 1
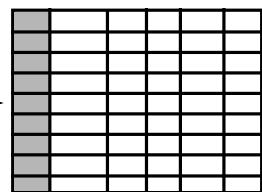
Table 2

ORACLE®

Capabilities of SQL SELECT Statements

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

- 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 selectively restrict the rows that you see.

- 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.

- Join: 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 will learn more about joins in a later lesson.

Basic SELECT Statement

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

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

ORACLE®

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.
DISTINCT	suppresses duplicates.
*	selects all columns.
column	selects the named column.
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.

- 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 empno, ename, ... is a clause.

- A *statement* is a combination of two or more clauses.
  For example, SELECT * FROM emp is a SQL statement.

Introduction to Oracle: SQL and PL/SQL 1-4

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.**
- **Tabs and indents are used to enhance readability.**

     ORACLE®

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.
- Tabs and indents can 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.
- Within SQL*Plus, a SQL statement is entered at the SQL prompt, and the subsequent lines are numbered. This is called the *SQL buffer*. Only one statement can be current at any time within the buffer.

Executing SQL Statements

- Place a semicolon (;) at the end of the last clause.
- Place a slash on the last line in the buffer.
- Place a slash at the SQL prompt.
- Issue a SQL*Plus RUN command at the SQL prompt.

Selecting All Columns

```
SQL> SELECT *
  2  FROM    dept;
```

```
    DEPTNO DNAME           LOC
---------- --------------- -------------
        10 ACCOUNTING      NEW YORK
        20 RESEARCH        DALLAS
        30 SALES           CHICAGO
        40 OPERATIONS      BOSTON
```

Selecting All Columns, 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 three columns: DEPTNO, DNAME, and LOC. The table contains four 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 DEPT table:

```
SQL> SELECT deptno, dname, loc
  2  FROM    dept;
```

Selecting Specific Columns

```
SQL> SELECT deptno, loc
  2  FROM    dept;
```

```
    DEPTNO LOC
---------- --------------
        10 NEW YORK
        20 DALLAS
        30 CHICAGO
        40 BOSTON
```

ORACLE®

Selecting Specific Columns, 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 locations from the DEPT table.

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

```
SQL> SELECT  loc, deptno
  2  FROM    dept;

LOC               DEPTNO
-------------- ---------
NEW YORK              10
DALLAS               20
CHICAGO              30
BOSTON               40
```

Column Heading Defaults

- **Default justification**
    - **Left: Date and character data**
    - **Right: Numeric data**
- **Default display: Uppercase**

 ORACLE®

Column Heading Defaults

Character column heading and data as well as date column heading and data are left-justified within a column width. Number headings and data are right-justified.

```
SQL> SELECT ename, hiredate, sal
  2  FROM   emp;

ENAME      HIREDATE       SAL
---------- --------- ---------
KING       17-NOV-81      5000
BLAKE      01-MAY-81      2850
CLARK      09-JUN-81      2450
JONES      02-APR-81      2975
MARTIN     28-SEP-81      1250
ALLEN      20-FEB-81      1600
...
14 rows selected.
```

Character and date column headings can be truncated, but number headings cannot be truncated. The column headings appear in uppercase by default. You can override the column heading display with an alias. Column aliases are covered later in this lesson.

Arithmetic Expressions

Create expressions on NUMBER and DATE data by using arithmetic operators.

Operator	Description
+	Add
-	Subtract
*	Multiply
/	Divide

Arithmetic Expressions

You may need to modify the way in which data is displayed, perform calculations, or look at what-if scenarios. This is possible using arithmetic expressions. An arithmetic expression may 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 the FROM clause.

Using Arithmetic Operators

```
SQL> SELECT  ename, sal, sal+300
  2  FROM    emp;
```

```
ENAME            SAL    SAL+300
---------- ---------- ----------
KING            5000       5300
BLAKE           2850       3150
CLARK           2450       2750
JONES           2975       3275
MARTIN          1250       1550
ALLEN           1600       1900
...
14 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 SAL+300 column in the output.

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

Note: SQL*Plus 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®

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

```
SQL> SELECT ename, sal, 12*sal+100
  2  FROM    emp;
```

```
ENAME            SAL 12*SAL+100
---------- --------- ----------
KING            5000      60100
BLAKE           2850      34300
CLARK           2450      29500
JONES           2975      35800
MARTIN          1250      15100
ALLEN           1600      19300
...
14 rows selected.
```

ORACLE®

Operator Precedence (continued)

The example on the slide displays the 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 above can be written as (12*sal)+100 with no change in the result.

Using Parentheses

```
SQL> SELECT ename, sal, 12*(sal+100)
  2  FROM    emp;
```

```
ENAME            SAL 12*(SAL+100)
---------- --------- -----------
KING            5000       61200
BLAKE           2850       35400
CLARK           2450       30600
JONES           2975       36900
MARTIN          1250       16200
...
14 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 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.**

```
SQL> SELECT ename, job, sal, comm
  2  FROM    emp;
```

```
ENAME        JOB              SAL     COMM
----------   ----------   ----------  ----------
KING         PRESIDENT       5000
BLAKE        MANAGER         2850
...
TURNER       SALESMAN        1500            0
...
14 rows selected.
```

ORACLE®

Null Values

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

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

Columns of any datatype can contain null values, unless the column was defined as NOT NULL or as PRIMARY KEY when the column was created.

In the COMM column in the EMP table, you notice that only a SALESMAN can earn commission. Other employees are not entitled to earn commission. A null value represents that fact. Turner, who is a salesman, does not earn any commission. Notice that his commission is zero and not null.

Null Values
in Arithmetic Expressions

Arithmetic expressions containing a null value evaluate to null.

```
SQL> select ename, 12*sal+comm
  2  from    emp
  3  WHERE   ename='KING';
```

```
ENAME      12*SAL+COMM
---------- -----------
KING
```

ORACLE®

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 is not in SALESMAN and does not get any commission. Because the COMM column in the arithmetic expression is null, the result is null.

For more information, see *Oracle Server SQL Reference,* Release 8, "Elements of SQL."

Defining a Column Alias

- **Renames a column heading**

- **Is useful with calculations**

- **Immediately follows column name; optional AS keyword between column name and alias**

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

Copyright © Oracle Corporation, 1999. All rights reserved.   ORACLE®

Column Aliases

When displaying the result of a query, SQL*Plus normally uses the name of the selected column as the column heading. In many cases, this heading may not be descriptive and hence is 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, special characters (such as # or $), or is case sensitive, enclose the alias in double quotation marks (" ").

Using Column Aliases

```
SQL> SELECT ename AS name, sal salary
  2  FROM    emp;
```

```
NAME              SALARY
------------ ---------
...
```

```
SQL> SELECT ename    "Name",
  2         sal*12 "Annual Salary"
  3  FROM    emp;
```

```
Name             Annual Salary
------------ -------------
...
```

  ORACLE®

Column Aliases (continued)

The first example displays the name and the monthly salary of all the employees. Notice that the optional AS keyword has been used before the column alias name. The result of the query would be the same whether the AS keyword is used or not. Also notice that the SQL statement has the column aliases, name and salary, in lowercase, whereas the result of the query displays the column headings in uppercase. As mentioned in the last slide, column headings appear in uppercase by default.

The second example displays the name and annual salary of all the employees. Because Annual Salary contains spaces, it has been enclosed in double quotation marks. Notice that the column heading in the output is exactly the same as the column alias.

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®

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

```
SQL> SELECT   ename||job AS "Employees"
  2  FROM     emp;
```

```
Employees
--------------------
KINGPRESIDENT
BLAKEMANAGER
CLARKMANAGER
JONESMANAGER
MARTINSALESMAN
ALLENSALESMAN
...
14 rows selected.
```

Concatenation Operator (continued)

In the example, ENAME and JOB are concatenated, and they are given the alias Employees. Notice that the employee number and job 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®

Literal Character Strings

A literal is character, a number, or a date included in the SELECT list 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 must not.

Using Literal Character Strings

```
SQL> SELECT ename ||' is a '||job
  2                AS "Employee Details"
  3  FROM    emp;

Employee Details
-------------------------
KING is a PRESIDENT
BLAKE is a MANAGER
CLARK is a MANAGER
JONES is a MANAGER
MARTIN is a SALESMAN
...
14 rows selected.
```

     ORACLE®

Literal Character Strings (continued)

The example on the slide displays names and jobs 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 name and salary for each employee is concatenated with a literal to give the returned rows more meaning.

```
SQL> SELECT ename ||': '||'1'||' Month salary = '||sal Monthly
  2  FROM    emp;


MONTHLY
----------------------------------------------------------------
KING: 1 Month salary = 5000
BLAKE: 1 Month salary = 2850
CLARK: 1 Month salary = 2450
JONES: 1 Month salary = 2975
MARTIN: 1 Month salary = 1250
ALLEN: 1 Month salary = 1600
TURNER: 1 Month salary = 1500
...
14 rows selected.
```

Duplicate Rows

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

```
SQL> SELECT deptno
  2  FROM    emp;
```

```
   DEPTNO
---------
       10
       30
       10
       20
...
14 rows selected.
```

ORACLE®

Duplicate Rows

Unless you indicate otherwise, SQL*Plus displays the results of a query without eliminating duplicate rows. The example on the slide displays all the department numbers from the EMP table. Notice that the department numbers are repeated.

Eliminating Duplicate Rows

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

```
SQL> SELECT DISTINCT deptno
  2  FROM     emp;
```

```
    DEPTNO
---------
        10
        20
        30
```

ORACLE®

Duplicate Rows (continued)
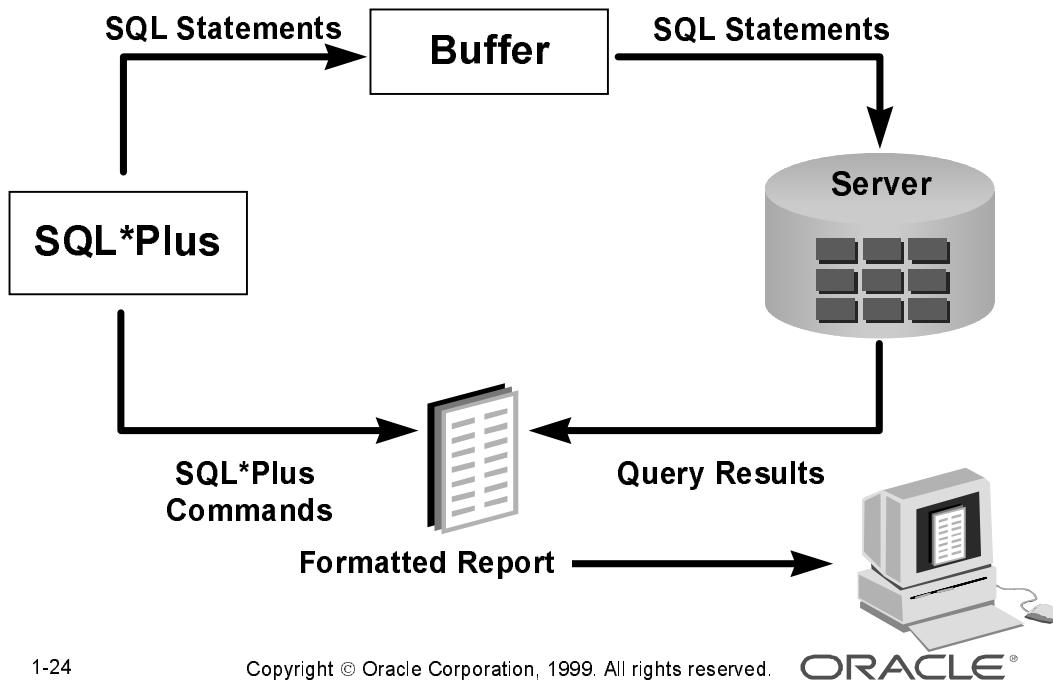
To eliminate duplicate rows in the result, include the DISTINCT keyword in the SELECT clause immediately after the SELECT keyword. In the example on the slide, the EMP table actually contains fourteen rows but there are only three unique department numbers in the table.

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

```
SQL> SELECT DISTINCT deptno, job
  2  FROM     emp;

DEPTNO JOB
------ ---------
    10 CLERK
    10 MANAGER
    10 PRESIDENT
    20 ANALYST
...
9 rows selected.
```

SQL and SQL*Plus Interaction

SQL Statements → **Buffer** → **SQL Statements**

SQL*Plus

Server

SQL*Plus Commands

Query Results

Formatted Report →

ORACLE®

SQL and SQL*Plus

SQL is a command language for communication with the Oracle Server from any tool or application. Oracle SQL contains many extensions. When you enter a SQL statement, it is stored in a part of memory called the *SQL buffer* and remains there until you enter a new statement.

*SQL*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
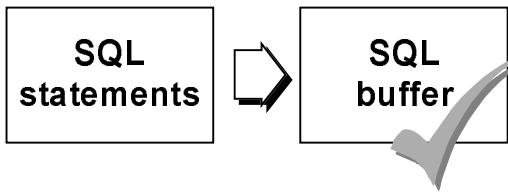
Features of SQL*Plus

- Accepts ad hoc entry of statements
- Accepts SQL input from files
- Provides a line editor for modifying SQL statements
- Controls environmental settings
- Formats query results into a basic report
- Accesses local and remote databases

SQL Statements Versus SQL*Plus Commands
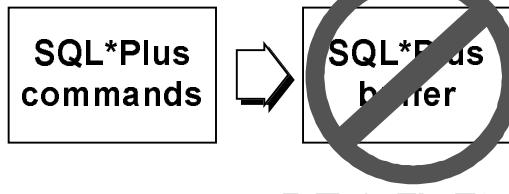
SQL

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

| SQL statements | ⇒ | SQL buffer |

SQL*Plus

- **An environment**
- **Oracle proprietary**
- **Keywords can be abbreviated**
- **Commands do not allow manipulation of values in the database**

| SQL*Plus commands | ⇒ | SQL*Plus buffer |

ORACLE®

SQL and SQL*Plus (continued)

The following table compares SQL and SQL*Plus:

SQL	SQL*Plus
Is a language for communicating with the Oracle Server to access data	Recognizes SQL statements and sends them to the Server
Is based on American National Standards Institute (ANSI) standard SQL	Is the Oracle proprietary interface for executing SQL statements
Manipulates data and table definitions in the database	Does not allow manipulation of values in the database
Is entered into the SQL buffer on one or more lines	Is entered one line at a time; not stored in the SQL buffer
Does not have a continuation character	Has a dash (-) as a continuation character if the command is longer than one line
Cannot be abbreviated	Can be abbreviated
Uses a termination character to execute command immediately	Does not require termination characters; commands are executed immediately
Uses functions to perform some formatting	Uses commands to format data

Overview of SQL*Plus

- **Log in to SQL*Plus.**
- **Describe the table structure.**
- **Edit your SQL statement.**
- **Execute SQL from SQL*Plus.**
- **Save SQL statements to files and append SQL statements to files.**
- **Execute saved files.**
- **Load commands from file to buffer to edit.**

 ORACLE®

SQL*Plus

SQL*Plus is an environment in which you can do the following:
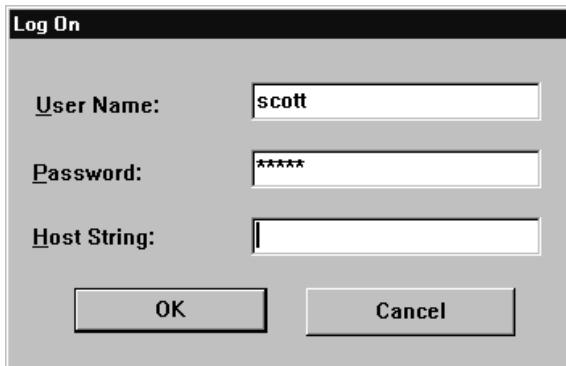
- Execute SQL statements to retrieve, modify, add, and remove data from the database
- Format, perform calculations on, store, and print query results in the form of reports
- Create script files to store SQL statements for repetitive use in the future

SQL*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, loads, and runs script files
Execution	Sends SQL statements from SQL buffer to Oracle8 Server
Edit	Modifies SQL statements in the buffer
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 SQL*Plus environment, and display column definitions

Logging In to SQL*Plus

• From Windows environment:

```
Log On

User Name:      scott

Password:       *****

Host String:    |

        OK              Cancel
```

• From command line:

> **sqlplus [*username*[/*password*
> [*@database*]]]**

     ORACLE®

Logging In to SQL*Plus

How you invoke SQL*Plus depends on which type of operating system or Windows environment you are running.

To log in through a Windows environment:

1. Click Start—>Programs—>Oracle for Windows NT—>SQL*Plus 8.0.

2. Fill in username, password, and database.

To log in through a command-line environment:

1. Log on to your machine.

2. Enter the SQL*Plus command as shown in the slide.

In the command:

username	is your database username
password	is your database password (if you enter your password here, it is visible)
@database	is the database connect string

Note: To ensure the integrity of your password, do not enter it at the operating system prompt. Instead, enter only your username. Enter your password at the Password prompt.

Once you are successfully logged in to SQL*Plus, you see the following message:

```
SQL*Plus : Release 8.0.3.0.0 - Production on Tue Jun 22 16:03:43 1999
(c) Copyright 1999 Oracle Corporation. All rights reserved.
```

Introduction to Oracle: SQL and PL/SQL 1-27

Displaying Table Structure

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

```
DESC[RIBE] tablename
```

ORACLE®

Displaying Table Structure

In SQL*Plus, you can display the structure of a table using the DESCRIBE command. The result of the command is to see the column names and datatypes 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

```
SQL> DESCRIBE dept
```

```
Name                    Null?      Type
-----------------       --------   ------------
DEPTNO                  NOT NULL   NUMBER(2)
DNAME                              VARCHAR2(14)
LOC                                VARCHAR2(13)
```

ORACLE®

Displaying Table Structure (continued)

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

In the result:

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

Type   displays the datatype for a column

The datatypes are described in the following table:

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

SQL*Plus Editing Commands

- **A[PPEND]** *text*
- **C[HANGE]** / *old* / *new*
- **C[HANGE]** / *text* /
- **CL[EAR] BUFF[ER]**
- **DEL**
- **DEL** *n*
- **DEL** *m n*

     ORACLE®

SQL*Plus Editing Commands

SQL*Plus commands are entered one line at a time and are not stored in the SQL buffer.

Command	Description
A[PPEND] *text*	Adds text to the end of the current line
C[HANGE] / *old* / *new*	Changes *old* text to *new* in the current line
C[HANGE] / *text* /	Deletes *text* from the current line
CL[EAR] BUFF[ER]	Deletes all lines from the SQL buffer
DEL	Deletes current line

Guidelines

- If you press [Return] before completing a command, SQL*Plus prompts you with a line number.
- You terminate the SQL buffer by either entering one of the terminator characters (semicolon or slash) or pressing [Return] twice. You then see the SQL prompt.

SQL*Plus Editing Commands

- **I[NPUT]**
- **I[NPUT]** *text*
- **L[IST]**
- **L[IST]** *n*
- **L[IST]** *m n*
- **R[UN]**
- *n*
- *n text*
- **0** *text*

SQL*Plus Editing Commands (continued)

Command	Description
I[NPU T]	Inserts an indefinite number of lines
I[NPUT] *text*	Inserts a line consisting of *text*
L[IST]	Lists all lines in the SQL buffer
L[IST] *n*	Lists one line (specified by *n*)
L[IST] *m n*	Lists a range of lines (*m* to *n*)
R[UN]	Displays and runs the current SQL statement in the buffer
n	Specifies the line to make the current line
n text	Replaces line *n* with *text*
0 *text*	Inserts a line before line 1

You can enter only one SQL*Plus command per SQL prompt. SQL*Plus commands are not stored in the buffer. To continue a SQL*Plus command on the next line, end the current line with a hyphen (-).

SQL*Plus File Commands

- **SAVE** *filename*
- **GET** *filename*
- **START** *filename*
- **@** *filename*
- **EDIT** *filename*
- **SPOOL** *filename*
- **EXIT**

ORACLE®

SQL*Plus File Commands

SQL statements communicate with the Oracle Server. SQL*Plus commands control the environment, format query results, and manage files. You can use the commands identified in the following table:

Command	Description
SAV[E] *filename* [.ext] [REP[LACE]APP[END]]	Saves current contents of SQL buffer to a file. Use APPEND to add to an existing file; use REPLACE to overwrite an existing file. The default extension is `.sql`.
GET *filename* [.ext]	Writes the contents of a previously saved file to the SQL buffer. The default extension for the filename is `.sql`.
STA[RT] *filename* [.ext]	Runs a previously saved command file.
@ *filename*	Runs a previously saved command file (same as START).
ED[IT]	Invokes the editor and saves the buffer contents to a file named `afiedt.buf.`
ED[IT] [*filename*[.ext]]	Invokes editor to edit contents of a saved file.
SPO[OL] [*filename*[.ext]| OFF|OUT]	Stores query results in a file. OFF closes the spool file. OUT closes the spool file and sends the file results to the system printer.
EXIT	Leaves SQL*Plus.

Summary

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

Use SQL*Plus as an environment to:
- ## Execute SQL statements
- ## Edit SQL statements

   ORACLE®

SELECT Statement

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

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

where:	SELECT	is a list of at least one column
	DISTINCT	suppresses duplicates
	*	selects all columns
	column	selects the named column
	alias	gives selected column a different heading
	FROM table	specifies the table containing the columns

SQL*Plus

SQL*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.

Practice Overview

- **Selecting all data from different tables**
- **Describing the structure of tables**
- **Performing arithmetic calculations and specifying column names**
- **Using SQL*Plus editor**

ORACLE®

Practice 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 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.

Take the practice 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.

Practice 1

1. Initiate a SQL*Plus session using the user ID and password provided by the instructor.

2. SQL*Plus commands access the database.
   True/False

3. Will the SELECT statement execute successfully?
   True/False

   ```
   SQL> SELECT      ename, job, sal Salary
     2  FROM        emp;
   ```

4. Will the SELECT statement execute successfully?
   True/False

   ```
   SQL> SELECT      *
     2  FROM        salgrade;
   ```

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

   ```
   SQL> SELECT      empno, ename
     2              salary x 12 ANNUAL SALARY
     3  FROM        emp;
   ```

6. Show the structure of the DEPT table. Select all data from the DEPT table.

   ```
   Name            Null?       Type
   -----------    --------    ------------
   DEPTNO         NOT NULL    NUMBER(2)
   DNAME                      VARCHAR2(14)
   LOC                        VARCHAR2(13)


   DEPTNO   DNAME          LOC
   ------   ----------    -------------
       10   ACCOUNTING    NEW YORK
       20   RESEARCH      DALLAS
       30   SALES         CHICAGO
       40   OPERATIONS    BOSTON
   ```

Practice 1 (continued)

7. Show the structure of the EMP table. Create a query to display the name, job, hire date, and employee number for each employee, with employee number appearing first. Save your SQL statement to a file named `p1q7.sql`.

```
Name            Null?       Type
--------------- ----------- --------------
EMPNO           NOT NULL    NUMBER(4)
ENAME                       VARCHAR2(10)
JOB                         VARCHAR2(9)
MGR                         NUMBER(4)
HIREDATE                    DATE
SAL                         NUMBER(7,2)
COMM                        NUMBER(7,2)
DEPTNO          NOT NULL    NUMBER(2)
```

8. Run your query in the file `p1q7.sql`.

```
EMPNO ENAME   JOB          HIREDATE
----- ------- ----------- ---------
 7839 KING    PRESIDENT    17-NOV-81
 7698 BLAKE   MANAGER      01-MAY-81
 7782 CLARK   MANAGER      09-JUN-81
 7566 JONES   MANAGER      02-APR-81
 7654 MARTIN  SALESMAN     28-SEP-81
 7499 ALLEN   SALESMAN     20-FEB-81
 7844 TURNER  SALESMAN     08-SEP-81
 7900 JAMES   CLERK        03-DEC-81
 7521 WARD    SALESMAN     22-FEB-81
 7902 FORD    ANALYST      03-DEC-81
 7369 SMITH   CLERK        17-DEC-80
 7788 SCOTT   ANALYST      09-DEC-82
 7876 ADAMS   CLERK        12-JAN-83
 7934 MILLER  CLERK        23-JAN-82
14 rows selected.
```

Practice 1 (continued)

9. Create a query to display unique jobs from the EMP table.

```
JOB
-----------
ANALYST
CLERK
MANAGER
PRESIDENT
SALESMAN
```

If you have time, complete the following exercises:

10. Load p1q7.sql into the SQL buffer. Name the column headings Emp #, Employee, Job, and Hire Date, respectively. Rerun your query.

```
Emp # Employee   Job         Hire Date
----- ---------- ----------  ----------------
 7839 KING       PRESIDENT   17-NOV-81
 7698 BLAKE      MANAGER     01-MAY-81
 7782 CLARK      MANAGER     09-JUN-81
 7566 JONES      MANAGER     02-APR-81
 7654 MARTIN     SALESMAN    28-SEP-81
 7499 ALLEN      SALESMAN    20-FEB-81
 7844 TURNER     SALESMAN    08-SEP-81
 7900 JAMES      CLERK       03-DEC-81
 7521 WARD       SALESMAN    22-FEB-81
 7902 FORD       ANALYST     03-DEC-81
 7369 SMITH      CLERK       17-DEC-80
 7788 SCOTT      ANALYST     09-DEC-82
 7876 ADAMS      CLERK       12-JAN-83
 7934 MILLER     CLERK       23-JAN-82
14 rows selected.
```

Practice 1 (continued)

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

```
Employee and Title
-------------------
KING, PRESIDENT
BLAKE, MANAGER
CLARK, MANAGER
JONES, MANAGER
MARTIN, SALESMAN
ALLEN, SALESMAN
TURNER, SALESMAN
JAMES, CLERK
WARD, SALESMAN
FORD, ANALYST
SMITH, CLERK
SCOTT, ANALYST
ADAMS, CLERK
MILLER, CLERK
14 rows selected.
```

If you want extra challenge, complete the following exercise:

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

```
THE_OUTPUT
--------------------------------------------------
7839,KING,PRESIDENT,,17-NOV-81,5000,,10
7698,BLAKE,MANAGER,7839,01-MAY-81,2850,,30
7782,CLARK,MANAGER,7839,09-JUN-81,2450,,10
7566,JONES,MANAGER,7839,02-APR-81,2975,,20
7654,MARTIN,SALESMAN,7698,28-SEP-81,1250,1400,30
7499,ALLEN,SALESMAN,7698,20-FEB-81,1600,300,30
7844,TURNER,SALESMAN,7698,08-SEP-81,1500,0,30
7900,JAMES,CLERK,7698,03-DEC-81,950,,30
7521,WARD,SALESMAN,7698,22-FEB-81,1250,500,30
7902,FORD,ANALYST,7566,03-DEC-81,3000,,20
7369,SMITH,CLERK,7902,17-DEC-80,800,,20
7788,SCOTT,ANALYST,7566,09-DEC-82,3000,,20
7876,ADAMS,CLERK,7788,12-JAN-83,1100,,20
7934,MILLER,CLERK,7782,23-JAN-82,1300,,10
14 rows selected.
```

2

Restricting and Sorting Data

ORACLE®

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®

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 will use to perform these actions.

Limiting Rows Using a Selection

EMP

EMPNO	ENAME	JOB	. . .	DEPTNO
7839	KING	PRESIDENT		10
7698	BLAKE	MANAGER		30
7782	CLARK	MANAGER		10
7566	JONES	MANAGER		20
. . .				

"...retrieve all employees in department 10"

EMP

EMPNO	ENAME	JOB	. . .	DEPTNO
7839	KING	PRESIDENT		10
7782	CLARK	MANAGER		10
7934	MILLER	CLERK		10

ORACLE®

Limiting Rows Using a Selection

In the example on the slide, assume that you want to display all the employees in department 10. The highlighted set of rows with a value of 10 in DEPTNO column are the only ones returned. This method of restriction is the basis of the WHERE clause in SQL.

Limiting Rows Selected

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

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

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

 ORACLE®

Limiting 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.

In the syntax:

WHERE	restricts the query to rows that meet a condition
condition	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. The WHERE clause consists of three elements:

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

Using the WHERE Clause

```
SQL> SELECT ename, job, deptno
  2  FROM    emp
  3  WHERE   job='CLERK';
```

```
ENAME       JOB           DEPTNO
----------  ----------  ---------
JAMES       CLERK            30
SMITH       CLERK            20
ADAMS       CLERK            20
MILLER      CLERK            10
```

ORACLE®

Using the WHERE Clause

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

Note that the job title CLERK has been specified in uppercase to ensure that the match is made with the job column in the EMP 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-YY.**

```
SQL>  SELECT    ename, job, deptno
  2   FROM      emp
  3   WHERE     ename = 'JAMES';
```

     ORACLE®

Character Strings and Dates

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

All character searches are case sensitive. In the following example, no rows are returned because the EMP table stores all the data in uppercase:

```
SQL>  SELECT  ename, empno, job, deptno
  2   FROM    emp
  3   WHERE   job='clerk';
```

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

Note: Changing default date format will be covered in a subsequent lesson.

Number values are not enclosed within quotation marks.

Comparison Operators

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

 ORACLE®

Comparison Operators

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

Syntax

```
…  WHERE expr operator value
```

Examples

```
…  WHERE hiredate='01-JAN-95'
…  WHERE sal>=1500
…  WHERE ename='SMITH'
```

Using the Comparison Operators

```
SQL> SELECT ename, sal, comm
  2  FROM    emp
  3  WHERE   sal<=comm;
```

```
ENAME            SAL      COMM
---------- --------- ---------
MARTIN          1250  ←→ 1400
```

Using the Comparison Operators

In the example, the SELECT statement retrieves name, salary, and commission from the EMP table, where the employee salary is less than or equal to the commission amount. Note that there is no explicit value supplied to the WHERE clause. The two values being compared are taken from the SAL and COMM columns in the EMP table.

Other Comparison Operators

Operator	Meaning
BETWEEN ...AND...	Between two values (inclusive)
IN(list)	Match any of a list of values
LIKE	Match a character pattern
IS NULL	Is a null value

ORACLE®

Using the BETWEEN Operator

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

```
SQL> SELECT    ename, sal
  2  FROM      emp
  3  WHERE     sal BETWEEN 1000 AND 1500;

ENAME              SAL              Lower      Higher
----------   ----------            limit      limit
MARTIN            1250
TURNER            1500
WARD              1250
ADAMS             1100
MILLER            1300
```

ORACLE®

The BETWEEN Operator

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

The SELECT statement on the slide returns rows from the EMP table for any employee whose salary is between $1000 and $1500.

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

Using the IN Operator

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

```
SQL> SELECT    empno, ename, sal, mgr
  2  FROM      emp
  3  WHERE     mgr IN (7902, 7566, 7788);
```

```
    EMPNO ENAME            SAL        MGR
--------- ---------- --------- ----------
     7902 FORD            3000       7566
     7369 SMITH            800       7902
     7788 SCOTT           3000       7566
     7876 ADAMS           1100       7788
```

The IN Operator

To test for values in a specified list, use the IN operator.

The slide example displays employee number, name, salary, and manager's employee number of all the employees whose manager's employee number is 7902, 7566, or 7788.

The IN operator can be used with any datatype. The following example returns a row from the EMP table for any employee whose name is included in the list of names in the WHERE clause:

```
SQL> SELECT    empno,  ename,  mgr, deptno
  2  FROM      emp
  3  WHERE     ename IN ('FORD' , 'ALLEN');
```

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

Using the LIKE Operator

- **Use the LIKE operator 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.**

```
SQL> SELECT    ename
  2  FROM      emp
  3  WHERE     ename LIKE 'S%';
```

   ORACLE®

The LIKE Operator

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

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

The SELECT statement above returns the employee name from the EMP table for any employee whose name begins with an "S." Note the uppercase "S." Names beginning with an "s" will not be returned.

The LIKE operator can be used as a shortcut for some BETWEEN comparisons. The following example displays names and hire dates of all employees who joined between January 1981 and December 1981:

```
SQL>  SELECT    ename, hiredate
  2   FROM      emp
  3   WHERE     hiredate LIKE '%1981';
```

Using the LIKE Operator

- **You can combine pattern-matching characters.**

```
SQL> SELECT    ename
  2  FROM      emp
  3  WHERE     ename LIKE '_A%';
```

```
ENAME
----------
MARTIN
JAMES
WARD
```

- **You can use the ESCAPE identifier to search for "%" or "_".**

  ORACLE®

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 name has an "A" 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 have HEAD_QUARTERS as a department name, you would search for it using the following SQL statement:

```
SQL> SELECT * FROM dept
  2  WHERE dname LIKE '%\_%' ESCAPE '\';

    DEPTNO    DNAME           LOC
    --------- --------------- -------------
        50    HEAD_QUARTERS   ATLANTA
```

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 IS NULL Operator

Test for null values with the IS NULL operator.

```
SQL> SELECT   ename, mgr
  2  FROM     emp
  3  WHERE    mgr IS NULL;
```

```
ENAME                   MGR
----------     ---------
KING
```

 ORACLE®

The IS NULL Operator

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

For example, to display name, job title, and commission for all employees who are not entitled to get a commission, use the following SQL statement:

```
SQL> SELECT      ename,  job, comm
  2  FROM        emp
  3  WHERE       comm  IS  NULL;


ENAME      JOB            COMM
--------  -----------  ------
KING       PRESIDENT
BLAKE      MANAGER
CLARK      MANAGER
...
```

Logical Operators

Operator	Meaning
AND	Returns TRUE if *both* component conditions are TRUE
OR	Returns TRUE if *either* component condition is TRUE
NOT	Returns TRUE if the following condition is FALSE

 ORACLE®

Logical Operators

A logical operator combines the result of two component conditions to produce a single result based on them or to invert the result of a single condition. 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.

```
SQL> SELECT empno, ename, job, sal
  2  FROM    emp
  3  WHERE   sal>=1100
  4  AND     job='CLERK';
```

```
    EMPNO ENAME      JOB              SAL
--------- ---------- ---------  ---------
     7876 ADAMS      CLERK           1100
     7934 MILLER     CLERK           1300
```

ORACLE®

The AND Operator

In the example, both conditions must be true for any record to be selected. Therefore, an employee who has a job title of CLERK *and* earns more than $1100 will be selected.

All character searches are case sensitive. No rows are returned if CLERK 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

Using the OR Operator

OR requires either condition to be TRUE.

```
SQL> SELECT empno, ename, job, sal
  2  FROM    emp
  3  WHERE   sal>=1100
  4  OR      job='CLERK';
```

```
    EMPNO ENAME      JOB              SAL
--------- ---------- -------- ---------
     7839 KING       PRESIDENT      5000
     7698 BLAKE      MANAGER        2850
     7782 CLARK      MANAGER        2450
     7566 JONES      MANAGER        2975
     7654 MARTIN     SALESMAN       1250
     ...
     7900 JAMES      CLERK           950
     ...
14 rows selected.
```

 ORACLE®

The OR Operator

In the example, either condition can be true for any record to be selected. Therefore, an employee who has a job title of CLERK *or* earns more than $1100 will be 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

Using the NOT Operator

```
SQL> SELECT  ename, job
  2  FROM    emp
  3  WHERE   job NOT IN ('CLERK','MANAGER','ANALYST');
```

```
ENAME       JOB
----------  ----------
KING        PRESIDENT
MARTIN      SALESMAN
ALLEN       SALESMAN
TURNER      SALESMAN
WARD        SALESMAN
```

     ORACLE®

The NOT Operator

The slide example displays name and job title of all the employees whose job title *is not* CLERK, MANAGER, or ANALYST.

The NOT Truth Table

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

NOT	TRUE	FALSE	NULL
TRUE	FALSE	TRUE	NULL

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

```
... WHERE   job   NOT  IN ('CLERK', 'ANALYST')
... WHERE   sal   NOT  BETWEEN  1000 AND  1500
... WHERE   ename NOT  LIKE '%A%'
... WHERE   comm  IS   NOT  NULL
```

Rules of Precedence

Order Evaluated	Operator
1	All comparison operators
2	NOT
3	AND
4	OR

Override rules of precedence by using parentheses.

ORACLE®

Rules of Precedence

```
SQL> SELECT  ename, job, sal
  2  FROM    emp
  3  WHERE   job='SALESMAN'
  4  OR      job='PRESIDENT'
  5  AND     sal>1500;
```

```
ENAME        JOB          SAL
----------   ----------   ---------
KING         PRESIDENT       5000
MARTIN       SALESMAN        1250
ALLEN        SALESMAN        1600
TURNER       SALESMAN        1500
WARD         SALESMAN        1250
```

ORACLE®

Example of Precedence of AND Operator

In the slide example, there are two conditions:

- The first condition is that job is PRESIDENT *and* salary is greater than 1500.
- The second condition is that job is SALESMAN.

Therefore, the SELECT statement reads as follows:

"Select the row if an employee is a PRESIDENT *and* earns more than $1500 *or* if the employee is a SALESMAN."

Rules of Precedence

Use parentheses to force priority.

```
SQL> SELECT    ename, job, sal
  2  FROM      emp
  3  WHERE     (job='SALESMAN'
  4  OR        job='PRESIDENT')
  5  AND       sal>1500;
```

```
ENAME        JOB           SAL
---------- ---------- ----------
KING         PRESIDENT     5000
ALLEN        SALESMAN      1600
```

Using Parentheses

In the example, there are two conditions:

- The first condition is that job is PRESIDENT *or* SALESMAN.
- The second condition is that salary is greater than 1500.

Therefore, the SELECT statement reads as follows:

"Select the row if an employee is a PRESIDENT or a SALESMAN and if the employee earns more than $1500."

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.**

```
SQL> SELECT     ename, job, deptno, hiredate
  2  FROM       emp
  3  ORDER BY hiredate;
```

```
ENAME       JOB          DEPTNO HIREDATE
----------  ----------  --------- ---------
SMITH       CLERK            20 17-DEC-80
ALLEN       SALESMAN         30 20-FEB-81
...
14 rows selected.
```

   ORACLE®

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, you must place last. You can specify an expression or an alias to sort.

Syntax

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

where:	ORDER BY	specifies the order in which the retrieved rows are displayed
	ASC	orders the rows in ascending order (this is the default order)
	DESC	orders the rows in descending order

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

Sorting in Descending Order

```
SQL> SELECT     ename, job, deptno, hiredate
  2  FROM       emp
  3  ORDER BY hiredate DESC;
```

```
ENAME        JOB          DEPTNO HIREDATE
---------- ---------- --------- ---------
ADAMS        CLERK            20 12-JAN-83
SCOTT        ANALYST          20 09-DEC-82
MILLER       CLERK            10 23-JAN-82
JAMES        CLERK            30 03-DEC-81
FORD         ANALYST          20 03-DEC-81
KING         PRESIDENT        10 17-NOV-81
MARTIN       SALESMAN         30 28-SEP-81
...
14 rows selected.
```

ORACLE®

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 keyword DESC after the column name in the ORDER BY clause. The slide example sorts the result by the most recently hired employee.

Sorting by Column Alias

```
SQL> SELECT    empno, ename, sal*12 annsal
  2  FROM      emp
  3  ORDER BY annsal;
```

```
    EMPNO ENAME         ANNSAL
--------- ---------- ---------
     7369 SMITH           9600
     7900 JAMES          11400
     7876 ADAMS          13200
     7654 MARTIN         15000
     7521 WARD           15000
     7934 MILLER         15600
     7844 TURNER         18000
...
14 rows selected.
```

Sorting by Column Aliases

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

Sorting by Multiple Columns

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

```
SQL> SELECT    ename, deptno, sal
  2  FROM      emp
  3  ORDER BY  deptno, sal DESC;
```

```
ENAME           DEPTNO          SAL
----------   ----------   ----------
KING                10         5000
CLARK               10         2450
MILLER              10         1300
FORD                20         3000
...
14 rows selected.
```

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

 ORACLE®

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 order by columns that are not included in the SELECT clause.

Example

Display name and salary of all employees. Order the result by department number and then descending order by salary.

```
SQL> SELECT    ename,  sal
  2  FROM      emp
  3  ORDER BY  deptno, sal DESC;
```

Summary

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

 ORACLE®

Summary

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

Practice Overview

- **Selecting data and changing the order of rows displayed**
- **Restricting rows by using the WHERE clause**
- **Using the double quotation marks in column aliases**

     ORACLE®

Practice 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 name and salary of employees earning more than $2850.
   Save your SQL statement to a file named `p2q1.sql`. Run your query.

   ```
   ENAME       SAL
   --------    ----
   KING        5000
   JONES       2975
   FORD        3000
   SCOTT       3000
   ```

2. Create a query to display the employee name and department number for employee number 7566.

   ```
   ENAME   DEPTNO
   ------  ------
   JONES       20
   ```

3. Modify `p2q1.sql` to display the name and salary for all employees whose salary is not in the range of $1500 and $2850. Resave your SQL statement to a file named `p2q3.sql`. Rerun your query.

   ```
   ENAME       SAL
   -------    -----
   KING        5000
   JONES       2975
   MARTIN      1250
   JAMES        950
   WARD        1250
   FORD        3000
   SMITH        800
   SCOTT       3000
   ADAMS       1100
   MILLER      1300
   10 rows selected.
   ```

Practice 2 (continued)

4. Display the employee name, job, and start date of employees hired between February 20, 1981, and May 1, 1981. Order the query in ascending order by start date.

```
ENAME    JOB        HIREDATE
-------  ---------  -----------------
ALLEN    SALESMAN   20-FEB-81
WARD     SALESMAN   22-FEB-81
JONES    MANAGER    02-APR-81
BLAKE    MANAGER    01-MAY-81
```

5. Display the employee name and department number of all employees in departments 10 and 30 in alphabetical order by name.

```
ENAME     DEPTNO
--------  ------
ALLEN         30
BLAKE         30
CLARK         10
JAMES         30
JONES         10
MARTIN        30
MILLER        10
TURNER        30
WARD          30
9 rows selected.
```

6. Modify p2q3.sql to list the name and salary of employees who earn more than $1500 and are in department 10 or 30. Label the columns Employee and Monthly Salary, respectively. Resave your SQL statement to a file named p2q6.sql. Rerun your query.

```
Employee  Monthly Salary
--------  --------------
KING                5000
BLAKE               2850
CLARK               2450
ALLEN               1600
```

Practice 2 (continued)

7. Display the name and hire date of every employee who was hired in 1982.

```
ENAME  HIREDATE
------ ---------
SCOTT  09-DEC-82
MILLER 23-JAN-82
```

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

```
ENAME JOB
----- ---------
KING  PRESIDENT
```

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

```
ENAME    SAL   COMM
-------  ----  ----
ALLEN    1600   300
TURNER   1500     0
MARTIN   1250  1400
WARD     1250   500
```

If you have time, complete the following exercises:

10. Display the names of all employees where the third letter of their name is an *A*.

```
ENAME
-----------
BLAKE
CLARK
ADAMS
```

11. Display the name of all employees who have two *L*s in their name and are in department 30 or their manager is 7782.

```
ENAME
----------
ALLEN
MILLER
```

Practice 2 (continued)

If you want extra challenge, complete the following exercises:

12. Display the name, job, and salary for all employees whose job is Clerk or Analyst and their salary is not equal to $1000, $3000, or $5000.

```
ENAME   JOB       SAL
------  -------   -----
JAMES   CLERK      950
SMITH   CLERK      800
ADAMS   CLERK     1100
MILLER  CLERK     1300
```

13. Modify p2q6.sql to display the name, salary, and commission for all employees whose commission amount is greater than their salary increased by 10%. Rerun your query. Resave your query as p2q13.sql.

```
Employee    Monthly Salary  COMM
----------  --------------- -----
MARTIN                 1250  1400
```

3

Single-Row Functions

ORACLE®

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**
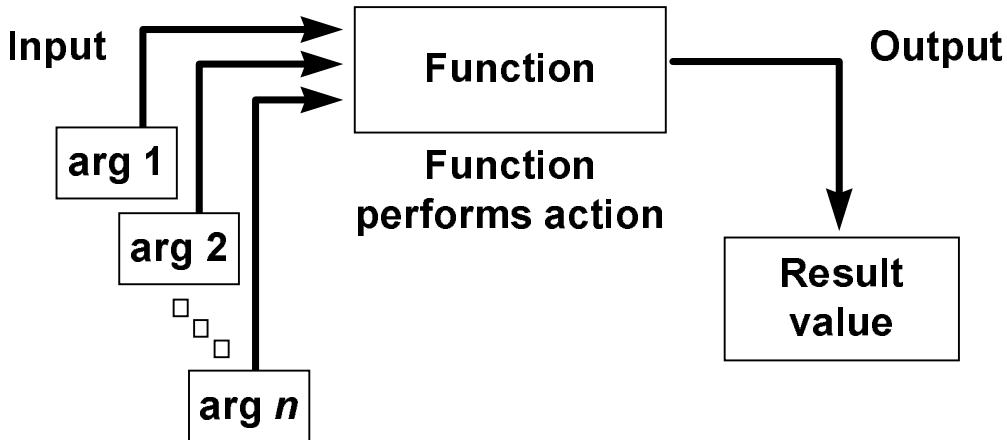
Lesson Aim

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. You will focus 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.

SQL Functions

ORACLE®
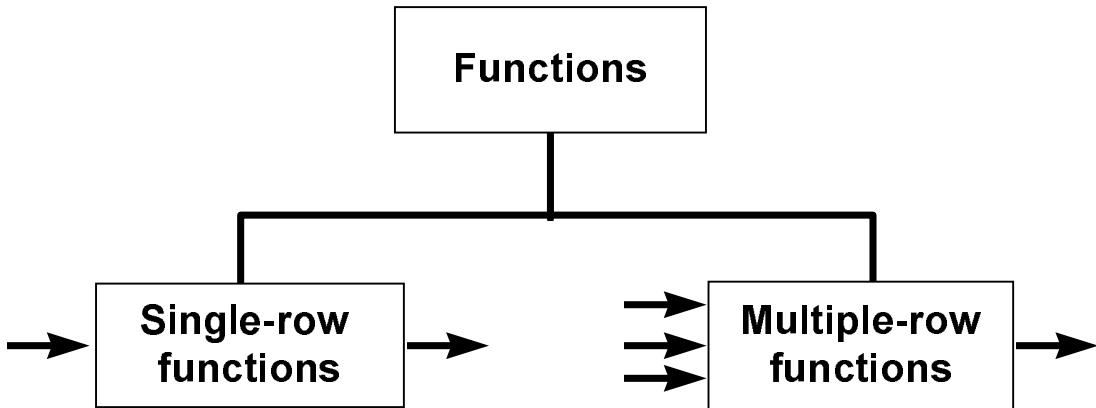
SQL Functions

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 datatypes

SQL functions may accept arguments and always return a value.

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

Two Types of SQL Functions

```
              ┌─────────────┐
              │  Functions  │
              └──────┬──────┘
         ┌───────────┴───────────┐
  ┌──────────────┐        ┌──────────────┐
→ │  Single-row  │   →  ══│ Multiple-row │ →
  │  functions   │   →  ══│  functions   │
  └──────────────┘   →  ══└──────────────┘
```

  ORACLE®

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

These functions manipulate groups of rows to give one result per group of rows.

For more information, see *Oracle Server SQL Reference*, Release 8, for the complete list of available functions and syntax.

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 datatype

- ## • Can be nested

```
function_name (column|expression, [arg1, arg2,...])
```

ORACLE®

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
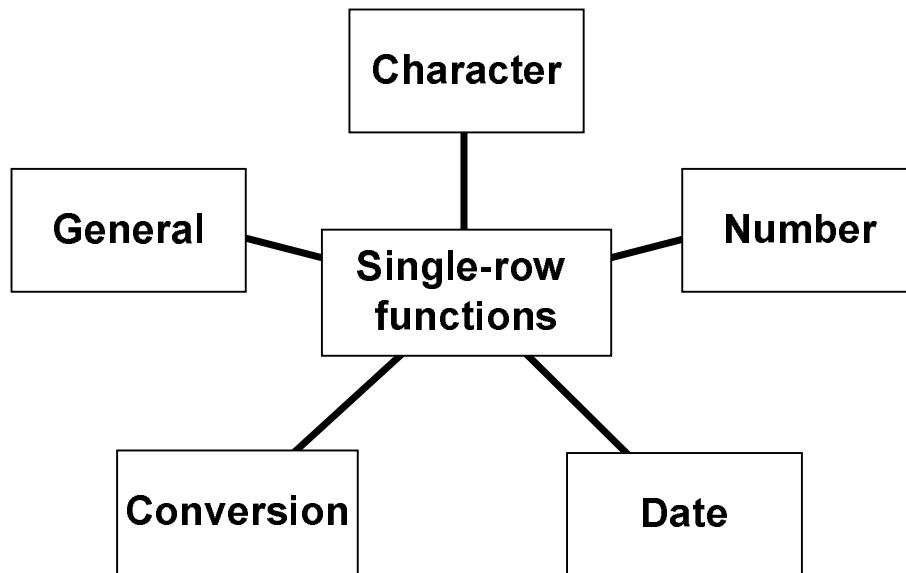
Features of single-row functions:

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

In the syntax:

function_name	is the name of the function
column	is any named database column
expression	is any character string or calculated expression
arg1, arg2	is any argument to be used by the function

Single-Row Functions
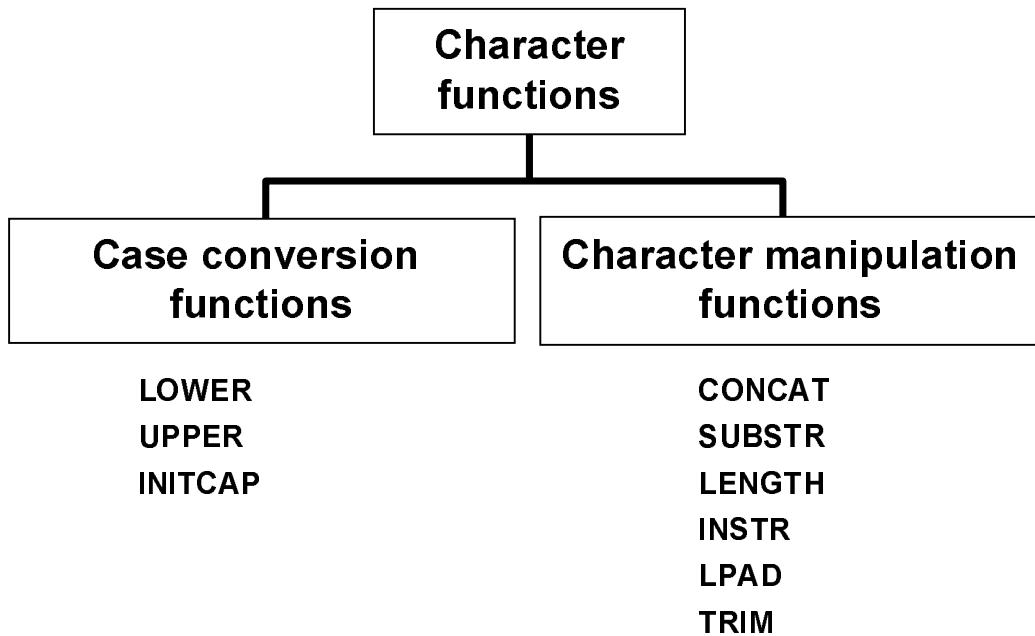
ORACLE®

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 datatype (All date functions return a value of date datatype except the MONTHS_BETWEEN function, which returns a number.)
- Conversion functions: Convert a value from one datatype to another
- General functions:
    - NVL function
    - DECODE function

Character Functions

Character functions

Case conversion functions

LOWER
UPPER
INITCAP

Character manipulation functions

CONCAT
SUBSTR
LENGTH
INSTR
LPAD
TRIM

ORACLE®

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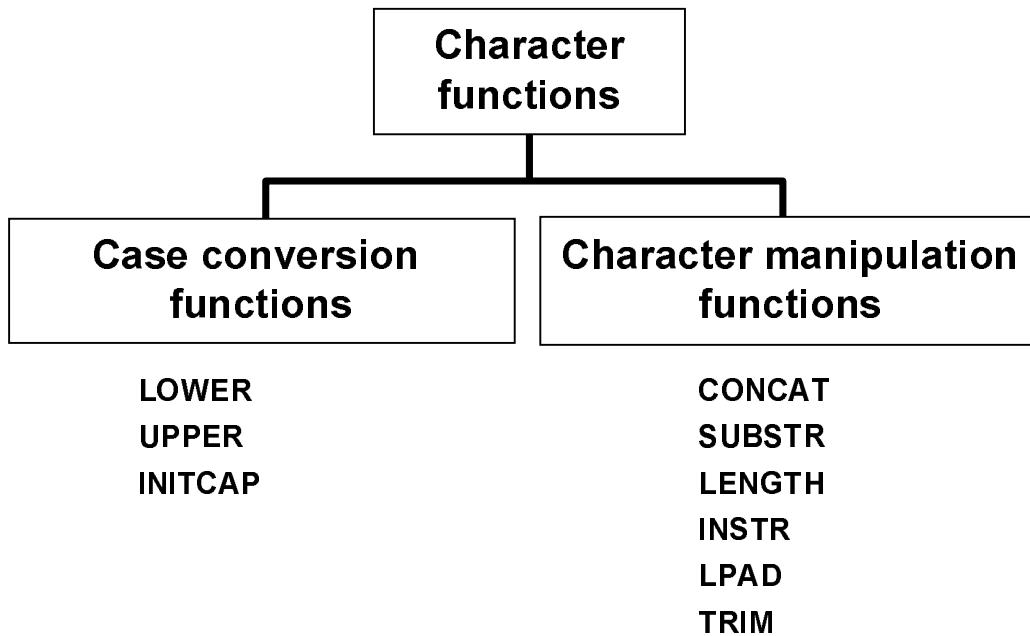
Single-row character functions accept character data as input and can return both character and number values. Character functions can be divided into the following:

- Case conversion functions
- Character manipulation functions

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

Note: The functions discussed in this lesson is a subset of the available functions.

Character Functions

```
                    ┌─────────────┐
                    │  Character  │
                    │  functions  │
                    └──────┬──────┘
              ┌────────────┴────────────┐
    ┌─────────┴─────────┐     ┌─────────┴──────────────┐
    │  Case conversion  │     │ Character manipulation │
    │     functions     │     │       functions        │
    └───────────────────┘     └────────────────────────┘
```

Case conversion functions	Character manipulation functions
LOWER	CONCAT
UPPER	SUBSTR
INITCAP	LENGTH
	INSTR
	LPAD
	TRIM

ORACLE®

Character Functions (continued)

Function	Purpose
LENGTH(*column*|*expression*)	Returns the number of characters in value
INSTR(*column*|*expression,m*)	Returns the numeric position of a named character
LPAD(*column*|*expression, n,* '*string*')	Pads the character value right-justified to a total width of *n* character positions
TRIM(*leading*|*trailing*|*both, trim_character FROM trim_source*)	Enables you to trim 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. This is a feature available from Oracle8*i* onward.

Case Conversion Functions

Convert case for character strings

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

ORACLE®

Case Conversion Functions

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

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

```
SQL> SELECT 'The job title for '||INITCAP(ename)||' is '
  2   ||LOWER(job) AS "EMPLOYEE DETAILS"
  3   FROM emp;

EMPLOYEE DETAILS
--------------------------------------------
The job title for King is president
The job title for Blake is manager
The job title for Clark is manager
...
14 rows selected.
```

Using Case Conversion Functions

Display the employee number, name, and department number for employee Blake.

```
SQL> SELECT    empno, ename, deptno
  2  FROM      emp
  3  WHERE     ename = 'blake';
no rows selected
```

```
SQL> SELECT    empno, ename, deptno
  2  FROM      emp
  3  WHERE     ename = UPPER('blake');
```

```
    EMPNO ENAME            DEPTNO
--------- ---------- ---------
     7698 BLAKE                30
```

Copyright © Oracle Corporation, 1999. All rights reserved.   **ORACLE**®

Case Conversion Functions (continued)

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

The WHERE clause of the first SQL statement specifies the employee name as 'blake.' Since all the data in the EMP table is stored in uppercase, the name 'blake' does not find a match in the EMP table and as a result no rows are selected.

The WHERE clause of the second SQL statement specifies that the employee name in the EMP table is compared to 'blake', converted to upper case. Since both the names are in uppercase 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  ename = 'BLAKE'
```

The name in the output appears as it was stored in the database. To display the name with the first letter capitalized, use the INITCAP function in the SELECT statement.

```
SQL> SELECT    empno, INITCAP(ename), deptno
  2  FROM      emp
  3  WHERE     ename = UPPER('blake');
```

Character Manipulation Functions

Manipulate character strings

Function	Result
CONCAT('Good', 'String')	GoodString
SUBSTR('String',1,3)	Str
LENGTH('String')	6
INSTR('String', 'r')	3
LPAD(sal,10,'*')	******5000
TRIM('S' FROM 'SSMITH')	MITH

ORACLE®

Character Manipulation Functions

CONCAT, SUBSTR, LENGTH, INSTR, LPAD, and TRIM are the six 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

Note: RPAD character manipulation function 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.

Using the Character Manipulation Functions

```
SQL> SELECT ename, CONCAT (ename, job), LENGTH(ename),
  2          INSTR(ename, 'A')
  3  FROM    emp
  4  WHERE   SUBSTR(job,1,5) = 'SALES';
```

ENAME	CONCAT(ENAME,JOB)	LENGTH(ENAME)	INSTR(ENAME,'A')
MARTIN	MARTINSALESMAN	6	2
ALLEN	ALLENSALESMAN	5	1
TURNER	TURNERSALESMAN	6	0
WARD	WARDSALESMAN	4	2

ORACLE®

Character Manipulation Functions (continued)

The slide example displays employee name and job joined together, length of the employee name, and the numeric position of the letter A in the employee name, for all employees who are in sales.

Example

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

```
SQL> SELECT   ename, CONCAT(ename, job), LENGTH(ename),
  2           INSTR(ename, 'A')
  3  FROM     emp
  4  WHERE    SUBSTR(ename, -1, 1) = 'N';
```

```
ENAME    CONCAT(ENAME,JOB)    LENGTH(ENAME)  INSTR(ENAME,'A')
-------- -------------------- -------------- -----------------
MARTIN   MARTINSALESMAN                    6                 2
ALLEN    ALLENSALESMAN                     5                 1
```

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®

Number Functions

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

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

Note: This list is a subset of the available number functions.

For more information, see *Oracle Server SQL Reference,* Release 8, "Number Functions."

Using the ROUND Function

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

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

Copyright © Oracle Corporation, 1999. All rights reserved.   ORACLE®

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 is a dummy table. More about this will be covered later.

Using the TRUNC Function

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

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

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 the ratio of salary to commission for all employees whose job title is salesman.

```
SQL> SELECT    ename, sal, comm, MOD(sal, comm)
  2  FROM      emp
  3  WHERE     job = 'SALESMAN';
```

```
ENAME             SAL       COMM MOD(SAL,COMM)
---------- --------- --------- --------------
MARTIN           1250      1400          1250
ALLEN            1600       300           100
TURNER           1500         0          1500
WARD             1250       500           250
```

MOD Function

The MOD function finds the remainder of value1 divided by value2. The slide example calculates the remainder of the ratio of salary to commission for all employees whose job title is salesman.

Working with Dates

- **Oracle stores dates in an internal numeric format: century, year, month, day, hours, minutes, seconds.**

- **The default date format is DD-MON-YY.**

- **SYSDATE is a function returning date and time.**

- **DUAL is a dummy table used to view SYSDATE.**

Oracle Date Format

Oracle 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-YY. Valid Oracle dates are between January 1, 4712 B.C., and December 31, 9999 A.D.

SYSDATE

SYSDATE is a date function that returns the current 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.

DUAL

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.

Example

Display the current date using the DUAL table.

```
SQL> SELECT   SYSDATE
  2  FROM     DUAL;
```

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®

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

```
SQL> SELECT ename, (SYSDATE-hiredate)/7 WEEKS
  2  FROM    emp
  3  WHERE   deptno = 10;
```

```
ENAME           WEEKS
----------   ---------
KING         830.93709
CLARK        853.93709
MILLER       821.36566
```

         ORACLE®

Arithmetic with Dates (continued)

The example on the slide displays the name and the number of weeks employed for all employees in department 10. It subtracts the current date (SYSDATE) from the date on which the employee was hired 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.

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®

Date Functions

Date functions operate on Oracle dates. All date functions return a value of DATE datatype 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-SEP-95')** ➙ **'30-SEP-95'**

Date Functions (continued)

For all employees employed for fewer than 200 months, display the employee number, hire date, number of months employed, six-month review date, first Friday after hire date, and last day of the month when hired.

```
SQL> SELECT  empno, hiredate,
  2          MONTHS_BETWEEN(SYSDATE, hiredate) TENURE,
  3          ADD_MONTHS(hiredate, 6) REVIEW,
  4          NEXT_DAY(hiredate, 'FRIDAY'), LAST_DAY(hiredate)
  5  FROM    emp
  6  WHERE   MONTHS_BETWEEN (SYSDATE, hiredate)<200;


    EMPNO HIREDATE      TENURE REVIEW     NEXT_DAY( LAST_DAY(  --
    ----- --------- --------- --------- --------- ---------
    7839 17-NOV-81 192.24794 17-MAY-82 20-NOV-81 30-NOV-81
    7698 01-MAY-81 198.76407 01-NOV-81 08-MAY-81 31-MAY-81
    ...
    11 rows selected.
```

Using Date Functions

- ROUND('25-JUL-95','MONTH') $\longrightarrow$ 01-AUG-95

- ROUND('25-JUL-95','YEAR') $\longrightarrow$ 01-JAN-96

- TRUNC('25-JUL-95','MONTH') $\longrightarrow$ 01-JUL-95

- TRUNC('25-JUL-95','YEAR') $\longrightarrow$ 01-JAN-95

 ORACLE®

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.
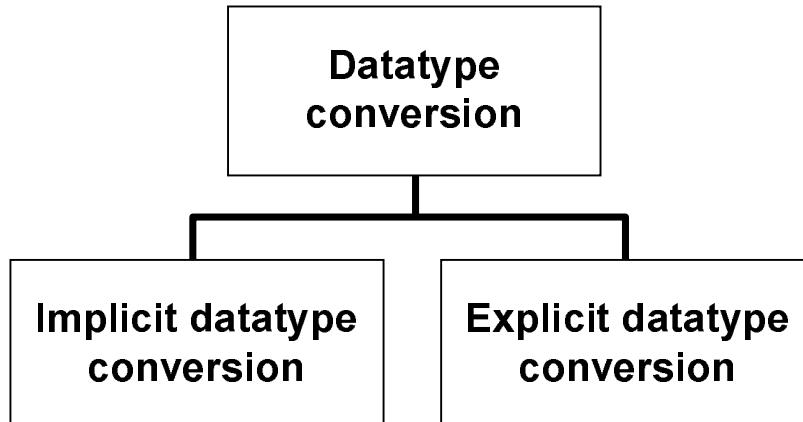
Example

Compare the hire dates for all employees who started in 1982. Display the employee number, hire date, and month started using the ROUND and TRUNC functions.

```
SQL> SELECT    empno, hiredate,
  2         ROUND(hiredate, 'MONTH'), TRUNC(hiredate, 'MONTH')
  3  FROM  emp
  4  WHERE hiredate like '%1982';

   EMPNO HIREDATE   ROUND(HIR TRUNC(HIR
--------- ---------- --------- ---------
    7788 09-DEC-82  01-DEC-82 01-DEC-82
    7934 23-JAN-82  01-FEB-82 01-JAN-82
```

Conversion Functions

```
          ┌─────────────────┐
          │    Datatype     │
          │   conversion    │
          └────────┬────────┘
         ┌─────────┴─────────┐
┌────────┴────────┐ ┌────────┴────────┐
│ Implicit datatype│ │ Explicit datatype│
│    conversion    │ │    conversion    │
└─────────────────┘ └─────────────────┘
```

 ORACLE®

Conversion Functions

In addition to Oracle datatypes, columns of tables in an Oracle8 database can be defined using ANSI, DB2, and SQL/DS datatypes. However, the Oracle Server internally converts such datatypes to Oracle8 datatypes.

In some cases, Oracle Server allows data of one datatype where it expects data of a different datatype. This is allowed when Oracle Server can automatically convert the data to the expected datatype. This datatype conversion can be done *implicitly* by Oracle Server or *explicitly* by the user.

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

Explicit datatype conversions are done by using the conversion functions. Conversion functions convert a value from one datatype to another. Generally, the form of the function names follows the convention *datatype* TO *datatype*. The first datatype is the input datatype; the last datatype is the output.

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

Implicit Datatype 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®

Implicit datatype Conversion

The assignment succeeds if the Oracle Server can convert the datatype of the value used in the assignment to that of the assignment target.

Implicit Datatype Conversion

For expression evaluation, the Oracle Server can automatically convert the following:
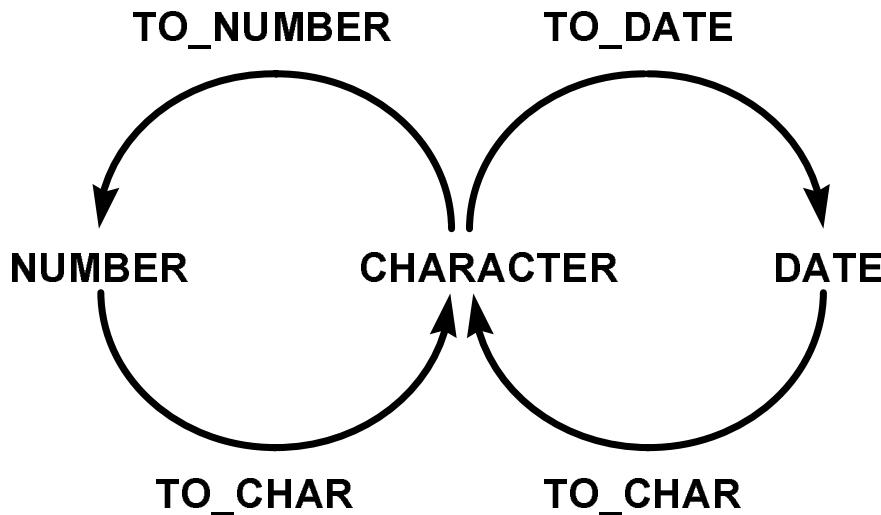
From	To
VARCHAR2 or CHAR	NUMBER
VARCHAR2 or CHAR	DATE

ORACLE®

Implicit Datatype Conversion

In general, the Oracle Server uses the rule for expression when a datatype 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. CHAR to DATE conversions succeed only if the character string has the default format DD-MON-YY.

Explicit Datatype Conversion

```
      TO_NUMBER              TO_DATE


   NUMBER        CHARACTER           DATE


      TO_CHAR                TO_CHAR
```
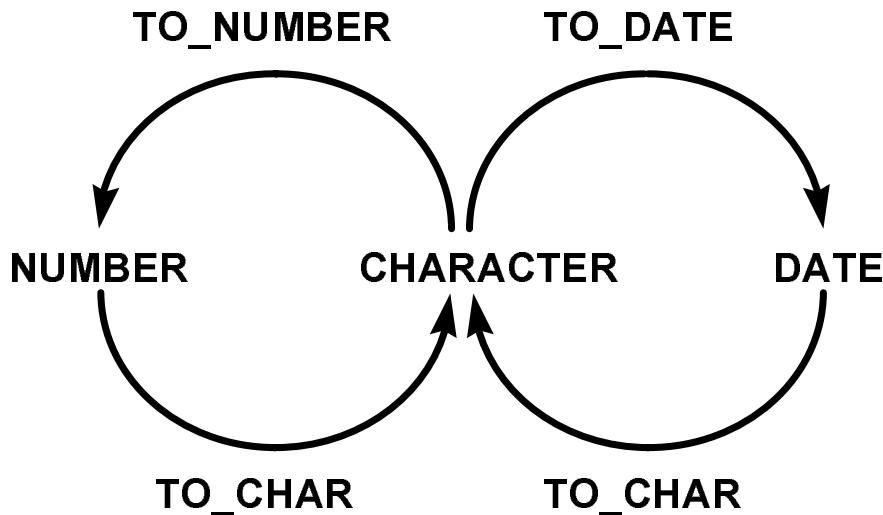
  ORACLE®

Explicit Datatype Conversion

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

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

Explicit Datatype Conversion
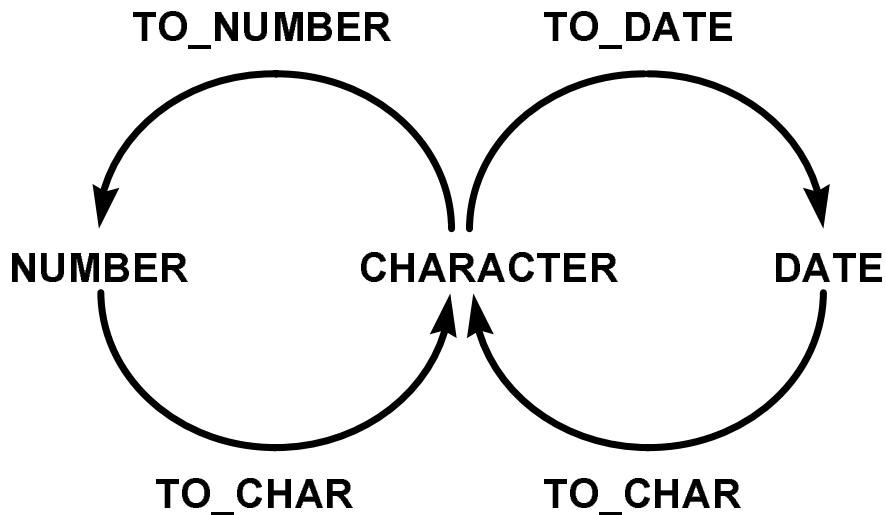
TO_NUMBER   **TO_DATE**

NUMBER   **CHARACTER**   **DATE**

TO_CHAR   **TO_CHAR**

     ORACLE®

Explicit Datatype Conversion (continued)

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

Explicit Datatype Conversion

TO_NUMBER   TO_DATE

NUMBER   CHARACTER   DATE

TO_CHAR   TO_CHAR

   ORACLE®

Explicit Datatype Conversion (continued)

Note: The list of functions mentioned in this lesson are a subset of the available conversion functions.

For more information, see *Oracle8 Server SQL Reference, Release 8.0,* "Conversion Functions."

TO_CHAR Function with Dates

```
TO_CHAR(date, 'fmt')
```

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**

Copyright © Oracle Corporation, 1999. All rights reserved. ORACLE®

Displaying a Date in a Specific Format

Previously, all Oracle date values were displayed in the DD-MON-YY format. The TO_CHAR function allows you 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 resize the display width of the resulting character field with the SQL*Plus COLUMN command.
- The resultant column width is 80 characters by default.

```
SQL> SELECT  empno, TO_CHAR(hiredate, 'MM/YY') Month_Hired
  2  FROM    emp
  3  WHERE   ename = 'BLAKE';
```

Elements of Date Format Model

YYYY	**Full year in numbers**
YEAR	**Year spelled out**
MM	**Two-digit value for month**
MONTH	**Full name of the month**
DY	**Three-letter abbreviation of the day of the week**
DAY	**Full name of the day**

Sample Elements of Valid Date Formats

Element	Description
SCC or CC	Century; S prefixes BC date with -
Years in dates YYYY or SYYYY	Year; S prefixes BC 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; S prefixes BC date with -
BC or AD	BC/AD indicator
B.C. or A.D.	BC/AD 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 length of 9 characters
DY	Name of day; three-letter abbreviation
J	Julian day; the number of days since 31 December 4713 BC

Elements of 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

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, DDSP for FOUR)
SPTH or THSP	Spelled-out ordinal numbers (for example, DDSPTH for FOURTH)

Using TO_CHAR Function
with Dates

```
SQL> SELECT ename,
  2           TO_CHAR(hiredate, 'fmDD Month YYYY') HIREDATE
  3    FROM   emp;
```

```
ENAME        HIREDATE
---------- ------------------
KING         17 November 1981
BLAKE        1 May 1981
CLARK        9 June 1981
JONES        2 April 1981
MARTIN       28 September 1981
ALLEN        20 February 1981
...
14 rows selected.
```

TO_CHAR Function with Dates

The SQL statement on the slide displays the name and hire dates for all the employees. The hire date appears as 17 November 1981.

Example

Modify the slide example to display the dates in a format that appears as Seventh of February 1981 08:00:00 AM.

```
SQL> SELECT  ename,
  2            TO_CHAR(hiredate, 'fmDdspth "of" Month YYYY fmHH:MI:SS AM')
  3            HIREDATE
  4    FROM    emp;
```

```
ENAME        HIREDATE
---------- ------------------------------------------------------
KING         Seventeenth of November 1981 12:00:00 AM
BLAKE        First of May 1981 12:00:00 AM
...
14 rows selected.
```

Notice that the month follows the format model specified, in other words, the first letter is capitalized and the rest in lower case.

TO_CHAR Function with Numbers

```
TO_CHAR(number, 'fmt')
```

Use these formats 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®

TO_CHAR Function with Numbers

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

Number Format Elements

If you are converting a number to character datatype, you can use the following 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 n times (n = number of 9s after V)	9999V99	123400
B	Display zero values as blank, not 0	B9999.99	1234.00

Introduction to Oracle: SQL and PL/SQL 3-33

Using TO_CHAR Function
with Numbers

```
SQL> SELECT   TO_CHAR(sal,'$99,999') SALARY
  2  FROM     emp
  3  WHERE    ename = 'SCOTT';
```

```
SALARY
--------
  $3,000
```

     ORACLE®

Guidelines

- 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.
- The Oracle Server rounds the stored decimal value to the number of decimal spaces provided in the format model.

TO_NUMBER and TO_DATE Functions

- **Convert a character string to a number format using the TO_NUMBER function**

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

- **Convert a character string to a date format using the TO_DATE function**

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

ORACLE®

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, you use the TO_NUMBER or TO_DATE functions. The format model you choose will be based on the previously demonstrated format elements.

Example

Display the names and hire dates of all the employees who joined on February 22, 1981.

```
SQL> SELECT  ename, hiredate
  2  FROM    emp
  3  WHERE   hiredate = TO_DATE('February 22, 1981', 'Month dd,  YYYY');


ENAME      HIREDATE
---------- --------
WARD       22-FEB-81
```

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®

The RR Date Format Element

The RR date format is similar to the YY element, but it allows you 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

NVL Function

Converts null to an actual value

- **Datatypes that can be used are date, character, and number.**
- **Datatypes must match**
  - **NVL(comm,0)**
  - **NVL(hiredate,'01-JAN-97')**
  - **NVL(job,'No Job Yet')**

 ORACLE®

The NVL Function

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

Syntax

```
NVL (expr1, expr2)
```

where: *expr1* is the source value or expression that may contain null

expr2 is the target value for converting null

You can use the NVL function to convert any datatype, but the return value is always the same as the datatype of *expr1*.

NVL Conversions for Various Datatypes

Data type	Conversion Example
NUMBER	NVL(*number_column*,9)
DATE	NVL(*date_column*, '01-JAN-95')
CHAR or VARCHAR2	NVL(*character_column*, 'Unavailable')

Using the NVL Function

```
SQL> SELECT ename, sal, comm, (sal*12)+NVL(comm,0)
  2  FROM    emp;
```

```
ENAME             SAL      COMM (SAL*12)+NVL(COMM,0)
---------- --------- --------- ---------------------
KING             5000                           60000
BLAKE            2850                           34200
CLARK            2450                           29400
JONES            2975                           35700
MARTIN           1250      1400                 16400
ALLEN            1600       300                 19500
...
14 rows selected.
```

ORACLE®

NVL Function

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

```
SQL> SELECT ename, sal, comm, (sal*12)+comm
  2  FROM    emp;
```

```
ENAME             SAL      COMM (SAL*12)+COMM
---------- --------- --------- --------------
KING             5000
BLAKE            2850
CLARK            2450
JONES            2975
MARTIN           1250      1400          16400
...
14 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 to is used to convert null values to zero.

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®

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

```
SQL> SELECT job, sal,
  2              DECODE(job, 'ANALYST', SAL*1.1,
  3                          'CLERK',   SAL*1.15,
  4                          'MANAGER', SAL*1.20,
  5                                     SAL)
  6              REVISED_SALARY
  7   FROM    emp;
```

```
JOB             SAL REVISED_SALARY
--------- --------- ---------------
PRESIDENT      5000            5000
MANAGER        2850            3420
MANAGER        2450            2940
...
14 rows selected.
```

ORACLE®

Using the DECODE Function

In the SQL statement above, the value of JOB is decoded. If JOB is ANALYST, the salary increase is 10%; if JOB is CLERK, the salary increase is 15%; if JOB is MANAGER, the salary increase is 20%. For all other job roles, there is no increase in salary.

The same statement can be written as an IF-THEN-ELSE statement:

```
IF job = 'ANALYST'    THEN   sal = sal*1.1
IF job = 'CLERK'      THEN   sal = sal*1.15
IF job = 'MANAGER'    THEN   sal = sal*1.20
ELSE sal = sal
```

Using the DECODE Function

Display the applicable tax rate for each employee in department 30.

```
SQL> SELECT ename, sal,
  2         DECODE(TRUNC(sal/1000, 0),
  3                          0, 0.00,
  4                          1, 0.09,
  5                          2, 0.20,
  6                          3, 0.30,
  7                          4, 0.40,
  8                          5, 0.42,
  9                          6, 0.44,
 10                              0.45) TAX_RATE
 11  FROM    emp
 12  WHERE   deptno = 30;
```

     ORACLE®

Example

The slide shows another example using the DECODE function. In this example, we determine the tax rate for each employee in department 30 based on the monthly salary. The tax rates are as per the values mentioned in the following table.
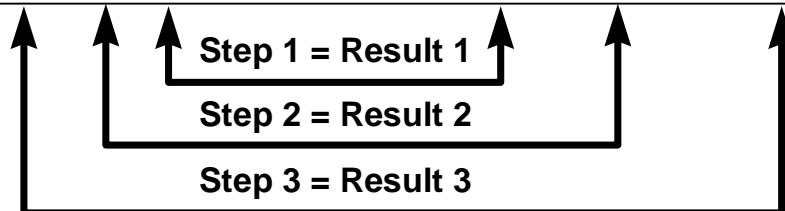
Monthly Salary Range	Rate
$0.00 - 999.99	0%
$1,000.00 - 1,999.99	9%
$2,000.00 - 2,999.99	20%
$3,000.00 - 3,999.99	30%
$4,000.00 - 4,999.99	40%
$5,000.00 - 2,999.99	42%
$6,000.00 - 6,999.99	44%
$7,000.00 or greater	45%

```
ENAME             SAL  TAX_RATE
---------- ---------- ---------
BLAKE            2850        .2
MARTIN           1250       .09
ALLEN            1600       .09
TURNER           1500       .09
...
6 rows selected.
```

Introduction to Oracle: SQL and PL/SQL 3-41

Nesting Functions

- **Single-row functions can be nested to any level.**

- **Nested functions are evaluated from deepest level to the least-deep level.**

```
F3(F2(F1(col,arg1),arg2),arg3)
```

Step 1 = Result 1

Step 2 = Result 2

Step 3 = Result 3

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

```
SQL> SELECT    ename,
  2            NVL(TO_CHAR(mgr),'No Manager')
  3   FROM     emp
  4   WHERE    mgr IS NULL;
```

```
ENAME         NVL(TO_CHAR(MGR),'NOMANAGER')
----------    ---------------------------
KING          No Manager
```

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(mgr)
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, March 12th, 1982. Order the results by hire date.

```
SQL> SELECT    TO_CHAR(NEXT_DAY(ADD_MONTHS
  2            (hiredate, 6), 'FRIDAY'),
  3            'fmDay, Month ddth, YYYY')
  4            "Next 6 Month Review"
  5   FROM     emp
  6   ORDER BY hiredate;
```

Summary

Use functions to do the following:

- **Perform calculations on data**
- **Modify individual data items**
- **Manipulate output for groups of rows**
- **Alter date formats for display**
- **Convert column datatypes**

ORACLE®

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

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 Overview

- **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**

Practice Overview

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

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

Practice 3

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

```
Date
---------
28-OCT-97
```

2. Display the employee number, name, salary, and salary increase by 15% expressed as a whole number. Label the column New Salary. Save your SQL statement to a file named p3q2.sql.

3. Run your query in the file p3q2.sql.

```
EMPNO ENAME      SAL New Salary
----- -------  ----- ----------
 7839 KING     5000        5750
 7698 BLAKE    2850        3278
 7782 CLARK    2450        2818
 7566 JONES    2975        3421
 7654 MARTIN   1250        1438
 7499 ALLEN    1600        1840
 7844 TURNER   1500        1725
 7900 JAMES     950        1093
 7521 WARD     1250        1438
 7902 FORD     3000        3450
 7369 SMITH     800         920
 7788 SCOTT    3000        3450
 7876 ADAMS    1100        1265
 7934 MILLER   1300        1495
14 rows selected.
```

4. Modify your query p3q2.sql to add a column that will subtract the old salary from the new salary. Label the column Increase. Rerun your query.

```
EMPNO ENAME      SAL New Salary Increase
----- -------  ----- ---------- --------
 7839 KING     5000        5750      750
 7698 BLAKE    2850        3278      428
 7782 CLARK    2450        2818      368
 7566 JONES    2975        3421      446
 7654 MARTIN   1250        1438      188
 7499 ALLEN    1600        1840      240
 7844 TURNER   1500        1725      225
 7900 JAMES     950        1093      143
 . . .
14 rows selected.
```

Practice 3 (continued)

5. Display the employee's 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 "Sunday, the Seventh of September, 1981."

```
ENAME   HIREDATE  REVIEW
-----   --------  ----------------------------------------
KING    17-NOV-81 Monday, the Twenty-Fourth of May, 1982
BLAKE   01-MAY-81 Monday, the Second of November, 1981
CLARK   09-JUN-81 Monday, the Fourteenth of December, 1981
JONES   02-APR-81 Monday, the Fifth of October, 1981
MARTIN  28-SEP-81 Monday, the Twenty-Ninth of March, 1982
ALLEN   20-FEB-81 Monday, the Twenty-Fourth of August, 1981
TURNER  08-SEP-81 Monday, the Fifteenth of March, 1982
JAMES   03-DEC-81 Monday, the Seventh of June, 1982
WARD    22-FEB-81 Monday, the Twenty-Fourth of August, 1981
FORD    03-DEC-81 Monday, the Seventh of June, 1982
SMITH   17-DEC-80 Monday, the Twenty-Second of June, 1981
SCOTT   09-DEC-82 Monday, the Thirteenth of June, 1983
ADAMS   12-JAN-83 Monday, the Eighteenth of July, 1983
MILLER  23-JAN-82 Monday, the Twenty-Sixth of July, 1982

14 rows selected.
```

6. For each employee display the employee 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.

```
ENAME      MONTHS_WORKED
-------    -------------
ADAMS                177
SCOTT                178
MILLER               188
JAMES                190
FORD                 190
KING                 191
MARTIN               192
TURNER               193
CLARK                196
BLAKE                197
JONES                198
WARD                 199
ALLEN                199
SMITH                202

14 rows selected
```

Practice 3 (continued)

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

```
Dream Salaries
---------------------------------------------------
KING earns $5,000.00 monthly but wants $15,000.00.
BLAKE earns $2,850.00 monthly but wants $8,550.00.
CLARK earns $2,450.00 monthly but wants $7,350.00.
JONES earns $2,975.00 monthly but wants $8,925.00.
MARTIN earns $1,250.00 monthly but wants $3,750.00.
ALLEN earns $1,600.00 monthly but wants $4,800.00
TURNER earns $1,500.00 monthly but wants $4,500.00.
JAMES earns $950.00 monthly but wants $2,850.00.
WARD earns $1,250.00 monthly but wants $3,750.00.
FORD earns $3,000.00 monthly but wants $9,000.00.
SMITH earns $800.00 monthly but wants $2,400.00.
SCOTT earns $3,000.00 monthly but wants $9,000.00.
ADAMS earns $1,100.00 monthly but wants $3,300.00
MILLER earns $1,300.00 monthly but wants $3,900.00.

14 rows selected.
```

If you have time, complete the following exercises:

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

```
ENAME      SALARY
--------   ---------------
SMITH      $$$$$$$$$$$$800
ALLEN      $$$$$$$$$$$1600
WARD       $$$$$$$$$$$1250
JONES      $$$$$$$$$$$2975
MARTIN     $$$$$$$$$$$1250
BLAKE      $$$$$$$$$$$2850
CLARK      $$$$$$$$$$$2450
SCOTT      $$$$$$$$$$$3000
KING       $$$$$$$$$$$5000
TURNER     $$$$$$$$$$$1500
ADAMS      $$$$$$$$$$$1100
JAMES      $$$$$$$$$$$$950
FORD       $$$$$$$$$$$3000
MILLER     $$$$$$$$$$$1300
14 rows selected.
```

Practice 3 (continued)

9. Write a query that will display the employee's name with the first letter capitalized and all other letters lowercase and the length of their name, for all employees whose name starts with J, A, or M. Give each column an appropriate label.

```
Name     Length
-------  ------
Jones         5
Martin        6
Allen         5
James         5
Adams         5
Miller        6
6 rows selected.
```

10. Display the 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.

```
ENAME    HIREDATE        DAY
------  ---------       ----------
MARTIN 28-SEP-81        MONDAY
CLARK  09-JUN-81        TUESDAY
KING   17-NOV-81        TUESDAY
TURNER 08-SEP-81        TUESDAY
SMITH  17-DEC-80        WEDNESDAY
ADAMS  12-JAN-83        WEDNESDAY
JONES  02-APR-81        THURSDAY
FORD   03-DEC-81        THURSDAY
SCOTT  09-DEC-82        THURSDAY
JAMES  03-DEC-81        THURSDAY
ALLEN  20-FEB-81        FRIDAY
BLAKE  01-MAY-81        FRIDAY
MILLER 23-JAN-82        SATURDAY
WARD   22-FEB-81        SUNDAY
14 rows selected.
```

Practice 3 (continued)

If you want extra challenge, complete the following exercises:

11. Create a query that will display the employee name and commission amount. If the employee does not earn commission, put "No Commission." Label the column COMM.

```
ENAME   COMM
------  -----------
SMITH   No Commission
ALLEN   300
WARD    500
JONES   No Commission
MARTIN  1400
BLAKE   No Commission
CLARK   No Commission
SCOTT   No Commission
KING    No Commission
TURNER  0
ADAMS   No Commission
JAMES   No Commission
FORD    No Commission
MILLER  No Commission

14 rows selected.
```

12. Create a query that displays the employees' names and indicates the amounts of their salaries through asterisks. Each asterisk signifies a hundred dollars. Sort the data in descending order of salary. Label the column EMPLOYEE_AND_THEIR_SALARIES.

```
EMPLOYEE_AND_THEIR_SALARIES
------------------------------------------------------------------
KING    **************************************************
FORD    ****************************
SCOTT   ****************************
JONES   **************************
BLAKE   *************************
CLARK   **********************
ALLEN   ****************
TURNER  ***************
MILLER  *************
MARTIN  ************
WARD    ************
ADAMS   ***********
JAMES   *********
SMITH   ********
14 rows selected.
```

Practice 3 (continued)

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

13. Write a query that displays the grade of all employees based on the value of the column JOB, as per the table shown below:

JOB	*GRADE*
PRESIDENT	A
MANAGER	B
ANALYST	C
SALESMAN	D
CLERK	E
None of the above	O

```
JOB        GRADE
---------  -----
CLERK      E
SALESMAN   D
SALESMAN   D
MANAGER    B
SALESMAN   D
MANAGER    B
MANAGER    B
ANALYST    C
PRESIDENT  A
SALESMAN   D
CLERK      E
CLERK      E
ANALYST    C
CLERK      E

14 rows selected.
```

4

Displaying Data
from Multiple Tables

ORACLE®

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**

   ORACLE®

Lesson Aim

This lesson covers how to obtain data from more than one table, using the different methods available.

Obtaining Data from Multiple Tables

EMP

EMPNO	ENAME	...	DEPTNO
7839	KING	...	10
7698	BLAKE	...	30
...			
7934	MILLER	...	10

DEPT

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

EMPNO	DEPTNO	LOC
7839	10	NEW YORK
7698	30	CHICAGO
7782	10	NEW YORK
7566	20	DALLAS
7654	30	CHICAGO
7499	30	CHICAGO

...
14 rows selected.

4-3

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.

- EMPNO exists in the EMP table.
- DEPTNO exists in both the EMP and DEPT tables.
- LOC exists in the DEPT table.

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

What Is a Join?

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®

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. Therefore, to join four tables, a minimum of three joins are 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 *Oracle Server SQL Reference Manual,* Release 8, "SELECT."

Cartesian Product

- **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®

Cartesian Product

When a join condition is invalid or omitted completely, the result is a *Cartesian product* in which all combinations of rows will be 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 its 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.

Generating a Cartesian Product

EMP (14 rows)

```
 EMPNO ENAME   ... DEPTNO
------ -----   ... ------
  7839 KING    ...     10
  7698 BLAKE   ...     30
  ...
  7934 MILLER  ...     10
```

DEPT (4 rows)

```
DEPTNO DNAME       LOC
------ ----------  --------
    10 ACCOUNTING  NEW YORK
    20 RESEARCH    DALLAS
    30 SALES       CHICAGO
    40 OPERATIONS  BOSTON
```

"Cartesian product: ➡ 14*4=56 rows"

```
ENAME       DNAME
------      ----------
KING        ACCOUNTING
BLAKE       ACCOUNTING
...
KING        RESEARCH
BLAKE       RESEARCH
...
56 rows selected.
```

ORACLE®

Cartesian Product (continued)
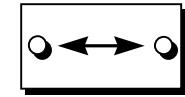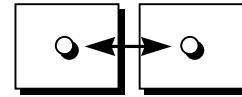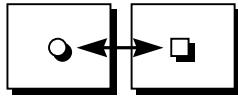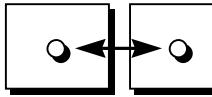
A Cartesian product is generated if a join condition is omitted. The example on the slide displays employee name and department name from EMP and DEPT tables. Because no WHERE clause has been specified, all rows (14 rows) from the EMP table are joined with all rows (4 rows) in the DEPT table, thereby generating 56 rows in the output.

```
SQL> SELECT  ename, dname
  2  FROM    emp, dept;

ENAME       DNAME
----------  ---------------
KING        ACCOUNTING
BLAKE       ACCOUNTING
...
KING        RESEARCH
BLAKE       RESEARCH
...
56 rows selected.
```

Types of Joins

Equijoin   Non-equijoin   Outer join   Self join

Types of Joins

There are two main types of join conditions:

- Equijoins
- Non-equijoins

Additional join methods include the following:

- Outer joins
- Self joins
- Set operators

Note: Set operators are not covered in this course. They are covered in another SQL course.

What Is an Equijoin?

EMP

```
EMPNO  ENAME    DEPTNO
------ -------- --------
  7839 KING          10
  7698 BLAKE         30
  7782 CLARK         10
  7566 JONES         20
  7654 MARTIN        30
  7499 ALLEN         30
  7844 TURNER        30
  7900 JAMES         30
  7521 WARD          30
  7902 FORD          20
  7369 SMITH         20
...
14 rows selected.
```

DEPT

```
DEPTNO DNAME      LOC
------- ---------- --------
     10 ACCOUNTING NEW YORK
     30 SALES      CHICAGO
     10 ACCOUNTING NEW YORK
     20 RESEARCH   DALLAS
     30 SALES      CHICAGO
     30 SALES      CHICAGO
     30 SALES      CHICAGO
     30 SALES      CHICAGO
     30 SALES      CHICAGO
     20 RESEARCH   DALLAS
     20 RESEARCH   DALLAS
...
14 rows selected.
```

Foreign key   Primary key

ORACLE®

Equijoins

To determine the name of an employee's department, you compare the value in the DEPTNO column in the EMP table with the DEPTNO values in the DEPT table. The relationship between the EMP and DEPT tables is an *equijoin*—that is, values in the DEPTNO 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*.

Retrieving Records
with Equijoins

```
SQL> SELECT    emp.empno,    emp.ename, emp.deptno,
  2            dept.deptno, dept.loc
  3  FROM      emp, dept
  4  WHERE     emp.deptno=dept.deptno;
```

```
EMPNO ENAME    DEPTNO DEPTNO LOC
----- ------   ------ ------ ---------
 7839 KING         10     10 NEW YORK
 7698 BLAKE        30     30 CHICAGO
 7782 CLARK        10     10 NEW YORK
 7566 JONES        20     20 DALLAS
...
14 rows selected.
```

   ORACLE®

Retrieving Records with Equijoins

In the slide example:

- The SELECT clause specifies the column names to retrieve:
    - employee name, employee number, and department number, which are columns in the EMP table
    - department number, department name, and location, which are columns in the DEPT table
- The FROM clause specifies the two tables that the database must access:
    - EMP table
    - DEPT table
- The WHERE clause specifies how the tables are to be joined:
    - EMP.DEPTNO=DEPT.DEPTNO

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

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.**

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 DEPTNO column could be from either the DEPT table or the EMP 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, you will gain improved performance by using the table prefix because you tell the Oracle Server exactly where to go to find 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.

Additional Search Conditions Using the AND Operator

EMP

```
 EMPNO  ENAME       DEPTNO
 ------ ---------  -------
  7839  KING            10
  7698  BLAKE           30
  7782  CLARK           10
  7566  JONES           20
  7654  MARTIN          30
  7499  ALLEN           30
  7844  TURNER          30
  7900  JAMES           30
  7521  WARD            30
  7902  FORD            20
  7369  SMITH           20
...
14 rows selected.
```

DEPT

```
 DEPTNO  DNAME       LOC
 ------  ---------   --------
     10  ACCOUNTING  NEW YORK
     30  SALES       CHICAGO
     10  ACCOUNTING  NEW YORK
     20  RESEARCH    DALLAS
     30  SALES       CHICAGO
     30  SALES       CHICAGO
     30  SALES       CHICAGO
     30  SALES       CHICAGO
     30  SALES       CHICAGO
     20  RESEARCH    DALLAS
     20  RESEARCH    DALLAS
...
14 rows selected.
```

ORACLE®

Additional Search Conditions

In addition to the join, you may have criteria for your WHERE clause. For example, to display employee King's employee number, name, department number, and department location, you need an additional condition in the WHERE clause.

```
SQL> SELECT empno, ename, emp.deptno, loc
  2  FROM    emp, dept
  3  WHERE   emp.deptno = dept.deptno
  4  AND     INITCAP(ename) = 'King';


     EMPNO ENAME          DEPTNO LOC
 --------- ---------- --------- --------------
      7839 KING              10 NEW YORK
```

Using Table Aliases

Simplify queries by using table aliases.

```
SQL> SELECT emp.empno,    emp.ename, emp.deptno,
  2            dept.deptno, dept.loc
  3  FROM     emp, dept
  4  WHERE    emp.deptno=dept.deptno;
```

```
SQL> SELECT e.empno,  e.ename, e.deptno,
  2            d.deptno, d.loc
  3  FROM     emp e,    dept d
  4  WHERE    e.deptno= d.deptno;
```

ORACLE®

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 EMP table has been given an alias of E, whereas the DEPT table has an alias of D.

Guidelines

- Table aliases can be up to 30 characters in length, but the shorter they are the 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

CUSTOMER

```
NAME              CUSTID
----------        ------
JOCKSPORTS           100
TKB SPORT SHOP       101
VOLLYRITE            102
JUST TENNIS          103
K+T SPORTS           105
SHAPE UP             106
WOMENS SPORTS        107
...                  ...
9 rows selected.
```

ORD

```
CUSTID    ORDID
-------   ------
   101      610
   102      611
   104      612
   106      601
   102      602
   106
   106
...
21 rows
```

ITEM

```
ORDID    ITEMID
------   -------
  610         3
  611         1
  612         1
  601         1
  602         1
...
64 rows selected.
```

ORACLE®

Additional Search Conditions

Sometimes you may need to join more than two tables. For example, to display the name, the orders placed, the item numbers, the total for each item, and the total for each order for customer TKB SPORT SHOP, you will have to join the CUSTOMER, ORD, and ITEM tables.

```
SQL> SELECT   c.name, o.ordid, i.itemid, i.itemtot, o.total
  2  FROM      customer c, ord o, item i
  3  WHERE     c.custid =  o.custid
  4  AND       o.ordid =   i.ordid
  5  AND       c.name = 'TKB SPORT SHOP';
```

```
NAME              ORDID      ITEMID     ITEMTOT      TOTAL
------------      ---------- ---------- ---------- ----------
TKB SPORT SHOP       610          3           58      101.4
TKB SPORT SHOP       610          1           35      101.4
TKB SPORT SHOP       610          2          8.4      101.4
```

Non-Equijoins

EMP

```
EMPNO  ENAME       SAL
------ -------  ------
  7839 KING       5000
  7698 BLAKE      2850
  7782 CLARK      2450
  7566 JONES      2975
  7654 MARTIN     1250
  7499 ALLEN      1600
  7844 TURNER     1500
  7900 JAMES       950
...
14 rows selected.
```

SALGRADE

```
GRADE LOSAL  HISAL
----- -----  ------
1       700   1200
2      1201   1400
3      1401   2000
4      2001   3000
5      3001   9999
```

◄— **"salary in the EMP table is between low salary and high salary in the SALGRADE table"**

ORACLE®

Non-Equijoins

The relationship between the EMP table and the SALGRADE table is a non-equijoin, meaning that no column in the EMP table corresponds directly to a column in the SALGRADE table. The relationship between the two tables is that the SAL column in the EMP table is between the LOSAL and HISAL column of the SALGRADE table. The relationship is obtained using an operator other than equal (=).

Retrieving Records with Non-Equijoins

```
SQL>   SELECT    e.ename, e.sal, s.grade
   2   FROM      emp e,    salgrade s
   3   WHERE     e.sal
   4   BETWEEN   s.losal AND s.hisal;
```

```
ENAME              SAL    GRADE
----------   ----------   ---------
JAMES              950          1
SMITH              800          1
ADAMS             1100          1
...
14 rows selected.
```

Non-Equijoins (continued)
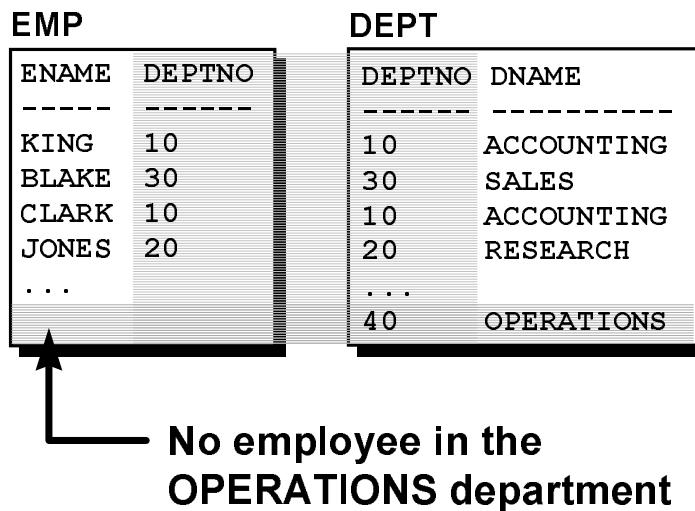
The slide example creates a non-equijoin to evaluate an employee's salary grade. The salary must be *between* any pair of the low and high salary ranges.

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 salary grade table contain grades that overlap. That is, the salary value for an employee can only lie 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 salary grade table. That is, no employee earns less than the lowest value contained in the LOSAL column or more than the highest value contained in the HISAL column.

Note: Other operators such as <= and >= could 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 for performance reasons, not because of possible ambiguity.

Outer Joins

EMP

ENAME	DEPTNO
KING	10
BLAKE	30
CLARK	10
JONES	20
...	

DEPT

DEPTNO	DNAME
10	ACCOUNTING
30	SALES
10	ACCOUNTING
20	RESEARCH
...	
40	OPERATIONS

No employee in the OPERATIONS department

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 EMP and DEPT tables, department OPERATIONS does not appear because no one works in that department.

```
SQL> SELECT e.ename, e.deptno, d.dname
  2  FROM    emp e,   dept d
  3  WHERE   e.deptno = d.deptno;

ENAME           DEPTNO DNAME
---------- --------- -------------
KING                10 ACCOUNTING
BLAKE               30 SALES
CLARK               10 ACCOUNTING
JONES               20 RESEARCH
...
ALLEN               30 SALES
TURNER              30 SALES
JAMES               30 SALES
...
14 rows selected.
```

Outer Joins

- **You use an outer join to also see rows that do not usually meet the join condition.**

- **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®

Returning Records with No Direct Match with Outer Joins

The missing row(s) 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:

table1.column =	is the condition that joins (or relates) the tables together.
table2.column (+)	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.)

Using Outer Joins

```
SQL> SELECT    e.ename, d.deptno, d.dname
  2  FROM      emp e,   dept d
  3  WHERE     e.deptno(+) = d.deptno
  4  ORDER BY e.deptno;
```

```
ENAME           DEPTNO DNAME
---------- --------- -------------
KING                10 ACCOUNTING
CLARK               10 ACCOUNTING
...
                    40 OPERATIONS
15 rows selected.
```

Returning Records with No Direct Match with Outer Joins (continued)

The slide example displays numbers and names for all the departments. The OPERATIONS department, which does not have any employees, is also displayed.

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.

Self Joins

EMP (WORKER)

EMPNO	ENAME	MGR
7839	KING	
7698	BLAKE	7839
7782	CLARK	7839
7566	JONES	7839
7654	MARTIN	7698
7499	ALLEN	7698

EMP (MANAGER)

EMPNO	ENAME
7839	KING
7839	KING
7839	KING
7698	BLAKE
7698	BLAKE

"MGR in the WORKER table is equal to EMPNO in the MANAGER table"

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 EMP table to itself, or perform a self join. For example, to find the name of Blake's manager, you need to:

- Find Blake in the EMP table by looking at the ENAME column.
- Find the manager number for Blake by looking at the MGR column. Blake's manager number is 7839.
- Find the name of the manager with EMPNO 7839 by looking at the ENAME column. King's employee number is 7839, so King is Blake's manager.

In this process, you look in the table twice. The first time you look in the table to find Blake in the ENAME column and MGR value of 7839. The second time you look in the EMPNO column to find 7839 and the ENAME column to find King.

Joining a Table to Itself

```
SQL> SELECT worker.ename||' works for '||manager.ename
  2  FROM    emp worker, emp manager
  3  WHERE   worker.mgr = manager.empno;
```

```
WORKER.ENAME||'WORKSFOR'||MANAG
-------------------------------
BLAKE works for KING
CLARK works for KING
JONES works for KING
MARTIN works for BLAKE
...
13 rows selected.
```

     ORACLE®

Joining a Table to Itself (continued)

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

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

Summary

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

Equijoin   Non-equijoin   Outer join   Self join

ORACLE®

Summary

There are multiple ways to join tables. The common thread, though, is that you want to link them through a condition in the WHERE clause. The method you choose will be based on the required result and the data structures that you are using.

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

Practice Overview

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

ORACLE®

Practice Overview

This practice is intended to give you practical experience in extracting data from more than one table. You will be required to join and restrict rows in the WHERE clause.

Practice 4

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

```
ENAME   DEPTNO DNAME
------  ------ ---------

KING        10 ACCOUNTING

BLAKE       30 SALES

CLARK       10 ACCOUNTING

JONES       20 RESEARCH

MARTIN      30 SALES

ALLEN       30 SALES

TURNER      30 SALES

JAMES       30 SALES

WARD        30 SALES

FORD        20 RESEARCH

SMITH       20 RESEARCH

SCOTT       20 RESEARCH

ADAMS       20 RESEARCH
MILLER      10 ACCOUNTING

14 rows selected.
```

2. Create a unique listing of all jobs that are in department 30. Include the location of department 30 in the output.

```
JOB         LOC
---------   -------------
CLERK       CHICAGO
MANAGER     CHICAGO
SALESMAN    CHICAGO
```

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

```
ENAME   DNAME  LOC
------  ------ -------------
ALLEN   SALES  CHICAGO
WARD    SALES  CHICAGO
MARTIN  SALES  CHICAGO
TURNER  SALES  CHICAGO
```

Practice 4 (continued)

4. Display the employee name and department name for all employees who have an *A* in their name. Save your SQL statement in a file called `p4q4.sql`.

```
ENAME     DNAME
-------   ------------
BLAKE     SALES
CLARK     ACCOUNTING
MARTIN    SALES
ALLEN     SALES
JAMES     SALES
WARD      SALES
ADAMS     RESEARCH
7 rows selected.
```

5. Write a query to display the name, job, department number, and department name for all employees who work in DALLAS.

```
ENAME     JOB      DEPTNO DNAME
-------   -------  ------- ----------
SMITH     CLERK        20 RESEARCH
ADAMS     CLERK        20 RESEARCH
FORD      ANALYST      20 RESEARCH
SCOTT     ANALYST      20 RESEARCH
JONES     MANAGER      20 RESEARCH
```

6. Display the employee name and employee number along with their manager's name and manager number. Label the columns Employee, Emp#, Manager, and Mgr#, respectively. Save your SQL statement in a file called `p4q6.sql`.

```
Employee    Emp# Manager  Mgr#
--------   ------ -------  ----
SCOTT       7788 JONES     7566
FORD        7902 JONES     7566
ALLEN       7499 BLAKE     7698
WARD        7521 BLAKE     7698
JAMES       7900 BLAKE     7698
TURNER      7844 BLAKE     7698
MARTIN      7654 BLAKE     7698
MILLER      7934 CLARK     7782
ADAMS       7876 SCOTT     7788
JONES       7566 KING      7839
CLARK       7782 KING      7839
BLAKE       7698 KING      7839
SMITH       7369 FORD      7902
13 rows selected.
```

Introduction to Oracle: SQL and PL/SQL 4-24

Practice 4 (continued)

7. Modify `p4q6.sql` to display all employees including King, who has no manager. Resave as `p4q7.sql`. Run `p4q7.sql`.

```
Employee    Emp# Manager  Mgr#
--------  ------ -------  -----
SCOTT       7788 JONES     7566
FORD        7902 JONES     7566
ALLEN       7499 BLAKE     7698
WARD        7521 BLAKE     7698
JAMES       7900 BLAKE     7698
TURNER      7844 BLAKE     7698
MARTIN      7654 BLAKE     7698
MILLER      7934 CLARK     7782
ADAMS       7876 SCOTT     7788
JONES       7566 KING      7839
CLARK       7782 KING      7839
BLAKE       7698 KING      7839
SMITH       7369 FORD      7902
KING        7839
14 rows selected.
```

If you have time, complete the following exercises:

8. Create a query that will display the employee name, department number, and all the employees that work in the same department as a given employee. Give each column an appropriate label.

```
DEPARTMENT  EMPLOYEE  COLLEAGUE
----------  --------  ---------
        10  CLARK     KING
        10  CLARK     MILLER
        10  KING      CLARK
        10  KING      MILLER
        10  MILLER    CLARK
        10  MILLER    KING
        20  ADAMS     FORD
        20  ADAMS     JONES
        20  ADAMS     SCOTT
        20  ADAMS     SMITH
        20  FORD      ADAMS
        20  FORD      JONES
        20  FORD      SCOTT
...
56 rows selected.
```

Practice 4 (continued)

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

```
Name           Null?     Type
-----------    --------  -------
GRADE                    NUMBER
LOSAL                    NUMBER
HISAL                    NUMBER
```

```
ENAME    JOB         DNAME        SAL   GRADE
------   ----------  ----------   ----- -----
MILLER   CLERK       ACCOUNTING   1300    2
CLARK    MANAGER     ACCOUNTING   2450    4
KING     PRESIDENT   ACCOUNTING   5000    5
SMITH    CLERK       RESEARCH      800    1
SCOTT    ANALYST     RESEARCH     3000    4
FORD     ANALYST     RESEARCH     3000    4
ADAMS    CLERK       RESEARCH     1100    1
JONES    MANAGER     RESEARCH     2975    4
JAMES    CLERK       SALES         950    1
BLAKE    MANAGER     SALES        2850    4
TURNER   SALESMAN    SALES        1500    3
ALLEN    SALESMAN    SALES        1600    3
WARD     SALESMAN    SALES        1250    2
MARTIN   SALESMAN    SALES        1250    2
14 rows selected.
```

If you want extra challenge, complete the following exercises:

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

```
ENAME    HIREDATE
-------  ---------
KING     17-NOV-81

CLARK    09-JUN-81

MARTIN   28-SEP-81

TURNER   08-SEP-81

JAMES    03-DEC-81

FORD     03-DEC-81

SCOTT    09-DEC-82

ADAMS    12-JAN-83

MILLER   23-JAN-82
9 rows selected.
```

Practice 4 (continued)

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

```
Employee  Emp Hiredate   Manager  Mgr Hiredate
--------  -------------  -------  ------------
ALLEN     20-FEB-81      BLAKE    01-MAY-81
WARD      22-FEB-81      BLAKE    01-MAY-81
JONES     02-APR-81      KING     17-NOV-81
CLARK     09-JUN-81      KING     17-NOV-81
BLAKE     01-MAY-81      KING     17-NOV-81
SMITH     17-DEC-80      FORD     03-DEC-81
6 rows selected.
```

5

Aggregating Data
Using Group Functions

ORACLE®

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**

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.

EMP

DEPTNO	SAL
10	2450
10	5000
10	1300
20	800
20	1100
20	3000
20	3000
20	2975
30	1600
30	2850
30	1250
30	950
30	1500
30	1250

"maximum salary in the EMP table"

MAX(SAL)
5000

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®

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|ALL]*n*)	Average value of *n*, ignoring null values
COUNT({*|[DISTINCT|ALL]*expr*})	Number of rows, where *expr* evaluates to something other than null (Count all selected rows using *, including duplicates and rows with nulls.)
MAX([DISTINCT|ALL]*expr*)	Maximum value of *expr*, ignoring null values
MIN([DISTINCT|ALL]*expr*)	Minimum value of *expr*, ignoring null values
STDDEV([DISTINCT|ALL]*x*)	Standard deviation of *n*, ignoring null values
SUM([DISTINCT|ALL]*n*)	Sum values of *n*, ignoring null values
VARIANCE([DISTINCT|ALL]*x*)	Variance of *n*, ignoring null values

Using Group Functions

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

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 datatypes for the arguments may be CHAR, VARCHAR2, NUMBER, or DATE where *expr* is listed.

• All group functions except COUNT(*) ignore null values. To substitute a value for null values, use the NVL function.

• 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.

Using AVG and SUM Functions

You can use AVG and SUM for numeric data.

```
SQL> SELECT    AVG(sal), MAX(sal),
  2            MIN(sal), SUM(sal)
  3  FROM      emp
  4  WHERE     job LIKE 'SALES%';
```

AVG(SAL)	MAX(SAL)	MIN(SAL)	SUM(SAL)
1400	1600	1250	5600

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 salespeople.

Using MIN and MAX Functions

You can use MIN and MAX for any datatype.

```
SQL> SELECT    MIN(hiredate), MAX(hiredate)
  2  FROM      emp;
```

```
MIN(HIRED MAX(HIRED
--------- ---------
17-DEC-80 12-JAN-83
```

ORACLE®

Group Functions (continued)

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

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

```
SQL> SELECT    MIN(ename), MAX(ename)
  2  FROM      emp;


MIN(ENAME) MAX(ENAME)
---------- ----------
ADAMS      WARD
```

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

Using the COUNT Function

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

```
SQL> SELECT    COUNT(*)
  2  FROM      emp
  3  WHERE     deptno = 30;
```

```
COUNT(*)
---------
        6
```

ORACLE®

The COUNT Function

The COUNT function has two formats:

- COUNT(*)
- COUNT(*expr*)

COUNT(*) returns the number of rows in a table, 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 nonnull rows in the column identified by *expr*.

The slide example displays the number of employees in department 30.

Using the COUNT Function

COUNT(*expr*) returns the number of nonnull rows.

```
SQL> SELECT    COUNT(comm)
  2  FROM      emp
  3  WHERE     deptno = 30;
```

```
COUNT(COMM)
-----------
          4
```

ORACLE®

The COUNT Function (continued)

The slide example displays the number of employees in department 30 who can earn a commission. Notice that the result gives the total number of rows to be four because two employees in department 30 cannot earn a commission and contain a null value in the COMM column.

Example

Display the number of departments in the EMP table.

```
SQL> SELECT    COUNT(deptno)
  2  FROM      emp;
```
```
COUNT(DEPTNO)
-------------
           14
```

Display the number of distinct departments in the EMP table.

```
SQL> SELECT    COUNT(DISTINCT (deptno))
  2  FROM      emp;
```
```
COUNT(DISTINCT(DEPTNO))
-----------------------
                      3
```

Group Functions and Null Values

Group functions ignore null values in the column.

```
SQL>  SELECT  AVG(comm)
  2   FROM    emp;
```

```
AVG(COMM)
---------
      550
```

ORACLE®

Group Functions and Null Values

All group functions except COUNT (*) 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 COMM column. The average is calculated as total commission being paid to all employees divided by the number of employees receiving commission (4).

Using the NVL Function
with Group Functions

The NVL function forces group functions to include null values.

```
SQL>  SELECT  AVG(NVL(comm,0))
   2  FROM    emp;
```

```
AVG(NVL(COMM,0))
----------------
        157.14286
```

ORACLE®

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 COMM column. The average is calculated as total commission being paid to all employees divided by the total number of employees in the company (14).

Creating Groups of Data

EMP

DEPTNO	SAL
10	2450
10	5000
10	1300
20	800
20	1100
20	3000
20	3000
20	2975
30	1600
30	2850
30	1250
30	950
30	1500
30	1250

2916.6667

2175

1566.6667

"average salary in EMP table for each department"

DEPTNO	AVG(SAL)
10	2916.6667
20	2175
30	1566.6667

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: GROUP BY Clause

```
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®

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:

group_by_expression   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 will receive an error message if you fail to include the column list.

- Using a WHERE clause, you can preexclude rows before dividing them into groups.

- You must include the *columns* in the GROUP BY clause.

- You cannot use the 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.

```
SQL> SELECT    deptno, AVG(sal)
  2  FROM      emp
  3  GROUP BY deptno;
```

```
  DEPTNO  AVG(SAL)
--------- ---------
       10 2916.6667
       20      2175
       30 1566.6667
```

ORACLE®

The GROUP BY Clause (continued)

When using the GROUP BY clause, make sure that all columns in the SELECT list that are not in the 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 EMP 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 EMP table.
- The WHERE clause specifies the rows to be retrieved. Since there is no WHERE clause, by default all rows are retrieved.
- 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.*

Using the GROUP BY Clause

The GROUP BY column does not have to be in the SELECT list.

```
SQL>  SELECT    AVG(sal)
  2   FROM      emp
  3   GROUP BY deptno;
```

```
 AVG(SAL)
---------
2916.6667
     2175
1566.6667
```

     ORACLE®

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.

```
SQL>  SELECT    deptno, AVG(sal)
  2   FROM      emp
  3   GROUP BY  deptno
  4   ORDER BY  AVG(sal);

    DEPTNO     AVG(SAL)
---------- ------------
        30    1566.6667
        20         2175
        10    2916.6667
```

Grouping by More Than One Column

EMP

DEPTNO	JOB	SAL
10	MANAGER	2450
10	PRESIDENT	5000
10	CLERK	1300
20	CLERK	800
20	CLERK	1100
20	ANALYST	3000
20	ANALYST	3000
20	MANAGER	2975
30	SALESMAN	1600
30	MANAGER	2850
30	SALESMAN	1250
30	CLERK	950
30	SALESMAN	1500
30	SALESMAN	1250

"sum salaries in the EMP table for each job, grouped by department"

DEPTNO	JOB	SUM(SAL)
10	CLERK	1300
10	MANAGER	2450
10	PRESIDENT	5000
20	ANALYST	6000
20	CLERK	1900
20	MANAGER	2975
30	CLERK	950
30	MANAGER	2850
30	SALESMAN	5600

Groups Within Groups

Sometimes there is a 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 EMP table is grouped first by department number, and within that grouping, it is grouped by job title. For example, the two clerks in department 20 are grouped together and a single result (total salary) is produced for all salespeople within the group.

Using the GROUP BY Clause on Multiple Columns

```
SQL> SELECT    deptno, job, sum(sal)
  2  FROM      emp
  3  GROUP BY deptno, job;
```

```
    DEPTNO JOB        SUM(SAL)
---------- --------- ---------
        10 CLERK          1300
        10 MANAGER        2450
        10 PRESIDENT      5000
        20 ANALYST        6000
        20 CLERK          1900
...
9 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 EMP table
  - Job title in the EMP 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 EMP 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 title.

So the SUM function is being applied to the salary column for all job titles 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.

```
SQL> SELECT    deptno, COUNT(ename)
  2  FROM      emp;
```

Column missing in the GROUP BY clause

```
SELECT deptno, COUNT(ename)
       *
ERROR at line 1:
ORA-00937: not a single-group group function
```

ORACLE®

Illegal Queries Using Group Functions

Whenever you use a mixture of individual items (DEPTNO) and group functions (COUNT) in the same SELECT statement, you must include a GROUP BY clause that specifies the individual items (in this case, DEPTNO). 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.

```
SQL> SELECT    deptno,COUNT(ename)
  2  FROM      emp
  3  GROUP BY  deptno;

    DEPTNO COUNT(ENAME)
---------- -------------
        10            3
        20            5
        30            6
```

Any column or expression in the SELECT list that is not an aggregate function must be in the GROUP BY clause.

Illegal Queries
Using Group Functions

- **You cannot use the WHERE clause to restrict groups.**
- **You use the HAVING clause to restrict groups.**

```
SQL> SELECT    deptno, AVG(sal)
  2  FROM      emp
  3  WHERE     AVG(sal) > 2000
  4  GROUP BY  deptno;
```

```
WHERE AVG(sal) > 2000
      *
ERROR at line 3
ORA-00934: group function is not allowed here
```

Cannot use the WHERE clause to restrict groups

   ORACLE®

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 $2000.

You can correct the slide error by using the HAVING clause to restrict groups.

```
SQL> SELECT    deptno, AVG(sal)
  2  FROM      emp
  3  GROUP BY  deptno
  4  HAVING    AVG(sal) > 2000;

    DEPTNO        AVG(SAL)
---------- ---------------
        10       2916.6667
        20            2175
```

Excluding Group Results

EMP

DEPTNO	SAL
10	2450
10	5000
10	1300
20	800
20	1100
20	3000
20	3000
20	2975
30	1600
30	2850
30	1250
30	950
30	1500
30	1250

5000

3000

2850

"maximum salary per department greater than $2900"

DEPTNO	MAX(SAL)
10	5000
20	3000

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 $2900, you need to do the following:

- Find the average salary for each department by grouping by department number.
- Restrict the groups to those departments with a maximum salary greater than $2900.

Excluding Group Results: HAVING Clause

Use the HAVING clause to restrict groups

- **Rows are grouped.**
- **The group function is applied.**
- **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];
```

The HAVING Clause

You use the HAVING clause to specify which groups are to be displayed. Therefore, 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:

- Rows are grouped.
- The group function is applied to the group.
- The groups that match the criteria in the HAVING clause are displayed.

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

Using the HAVING Clause

```
SQL> SELECT    deptno, max(sal)
  2  FROM      emp
  3  GROUP BY deptno
  4  HAVING    max(sal)>2900;
```

```
    DEPTNO   MAX(SAL)
  --------- ---------
        10      5000
        20      3000
```

    ORACLE®

The HAVING Clause (continued)

The slide example displays department numbers and maximum salary for those departments whose maximum salary is greater than $2900.

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 salary for those departments whose maximum salary is greater than $2900:

```
SQL> SELECT    deptno, AVG(sal)
  2  FROM      emp
  3  GROUP BY deptno
  4  HAVING    MAX(sal) > 2900;

    DEPTNO   AVG(SAL)
  --------- ---------
        10 2916.6667
        20      2175
```

Using the HAVING Clause

```
SQL>  SELECT     job, SUM(sal) PAYROLL
  2   FROM       emp
  3   WHERE      job NOT LIKE 'SALES%'
  4   GROUP BY   job
  5   HAVING     SUM(sal)>5000
  6   ORDER BY   SUM(sal);
```

```
JOB          PAYROLL
---------  ---------
ANALYST         6000
MANAGER         8275
```

The HAVING Clause (continued)

The slide example displays the job title and total monthly salary for each job title with a total payroll exceeding $5000. The example excludes salespeople and sorts the list by the total monthly salary.

Nesting Group Functions

Display the maximum average salary.

```
SQL> SELECT    max(avg(sal))
  2  FROM      emp
  3  GROUP BY deptno;
```

```
MAX(AVG(SAL))
-------------
   2916.6667
```

ORACLE®

Nesting Group Functions

Group functions can be nested to a depth of two. The slide example displays the maximum average salary.

Summary

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

Order of evaluation of the clauses:

- **WHERE clause**

- **GROUP BY clause**

- **HAVING clause**

ORACLE®

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:

- 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.

Practice Overview

- **Showing different queries that use group functions**
- **Grouping by rows to achieve more than one result**
- **Excluding groups by using the HAVING clause**

ORACLE®

Practice 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.

Practice 5

Determine the validity of the following 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. Save your SQL statement in a file called `p5q4.sql`.

   ```
   Maximum  Minimum    Sum Average
   -------  -------  ----- -------
      5000      800  29025    2073
   ```

5. Modify `p5q4.sql` to display the minimum, maximum, sum, and average salary for each job type. Resave to a file called `p5q5.sql`. Rerun your query.

   ```
   JOB          Maximum  Minimum    Sum  Average
   ----------  --------  --------  ----- --------
   ANALYST         3000      3000   6000     3000
   CLERK           1300       800   4150     1038
   MANAGER         2975      2450   8275     2758
   PRESIDENT       5000      5000   5000     5000
   SALESMAN        1600      1250   5600     1400
   ```

6. Write a query to display the number of people with the same job.

   ```
   JOB          COUNT(*)
   ----------  --------
   ANALYST            2
   CLERK              4
   MANAGER            3
   PRESIDENT          1
   SALESMAN           4
   ```

Practice 5 (continued)

7. Determine the number of managers without listing them. Label the column Number of Managers.

```
Number of Managers
------------------
                 6
```

8. Write a query that will display the difference between the highest and lowest salaries. Label the column DIFFERENCE.

```
DIFFERENCE
----------
      4200
```

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 less than $1000. Sort the output in descending order of salary.

```
     MGR       MIN(SAL)
--------       --------
    7566           3000
    7839           2450
    7782           1300
    7788           1100
```

10. Write a query to display the department name, location name, number of employees, and the average salary for all employees in that department. Label the columns dname, loc, Number of People, and Salary, respectively. Round the average salary to two decimal places.

```
DNAME        LOC        Number of People        Salary
-----------  ---------  ----------------        --------
ACCOUNTING   NEW YORK                  3         2916.67
RESEARCH     DALLAS                    5         2175
SALES        CHICAGO                   6         1566.67
```

Practice 5 (continued)

If you want extra challenge, complete the following exercises:

11. Create a query that will display the total number of employees and of that total the number who were hired in 1980, 1981, 1982, and 1983. Give appropriate column headings.

```
TOTAL   1980    1981    1982  1983
-----  -----   -----   ----- -----
   14      1      10       2     1
```

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 all departments, giving each column an appropriate heading.

```
Job         Dept 10   Dept 20   Dept 30   Total
---------   -------   --------  --------  -------
ANALYST                  6000                6000
CLERK          1300      1900       950     4150
MANAGER        2450      2975      2850     8275
PRESIDENT      5000                         5000
SALESMAN                           5600     5600
```

6

Subqueries

ORACLE®

Objectives

After completing this lesson, you should be able to do the following:
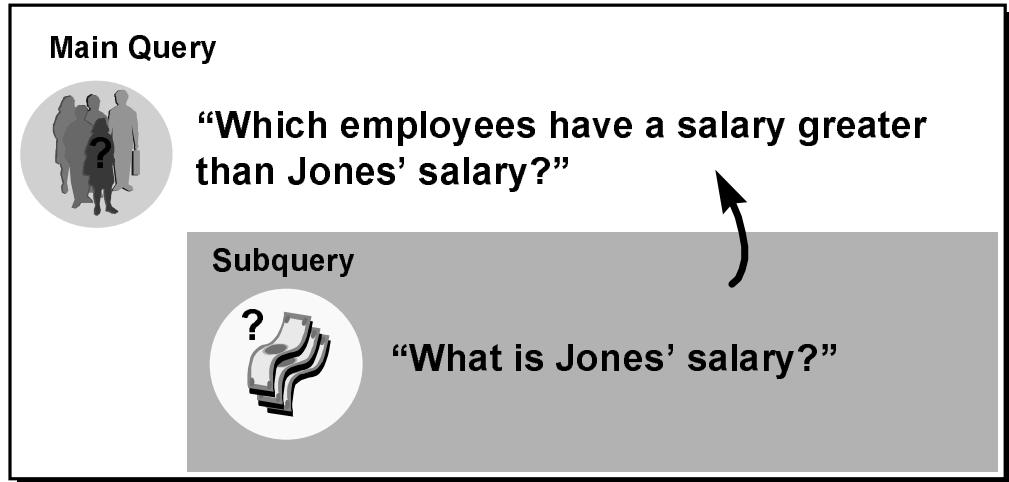
- **Describe the types of problems that subqueries can solve**

- **Define subqueries**

- **List the types of subqueries**

- **Write single-row and multiple-row subqueries**

ORACLE®

Lesson Aim

In this lesson, you will 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 Jones'?"

Main Query

"Which employees have a salary greater than Jones' salary?"

Subquery

?

"What is Jones' salary?"

ORACLE®

Using a Subquery to Solve a Problem

Suppose you want to write a query to find out who earns a salary greater than Jones' salary.

To solve this problem, you need *two* queries: one query to find what Jones 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.

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®

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:

- WHERE clause
- HAVING clause
- FROM clause

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 subquery generally executes first, and its output is used to complete the query condition for the main or outer query.

Using a Subquery

```
SQL> SELECT ename
  2  FROM    emp                    2975
  3  WHERE   sal >
  4              (SELECT  sal
  5               FROM    emp
  6               WHERE   empno=7566);
```

```
ENAME
----------
KING
FORD
SCOTT
```

ORACLE®

Using a Subquery

In the slide, the inner query determines the salary of employee 7566. 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.

Guidelines for Using Subqueries

- **Enclose subqueries in parentheses.**
- **Place subqueries on the right side of the comparison operator.**
- **Do not add an ORDER BY clause to a subquery.**
- **Use single-row operators with single-row subqueries.**
- **Use multiple-row operators with multiple-row subqueries.**
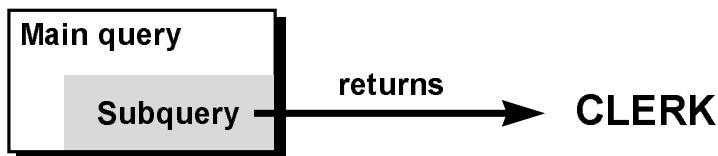
ORACLE®

Guidelines for Using Subqueries

- A subquery must be enclosed in parentheses.
- A subquery must appear on the right side of the comparison operator.
- Subqueries cannot contain an ORDER BY clause. You can have only one ORDER BY clause for a SELECT statement, and if specified it must be the last clause in the main SELECT statement.
- Two classes of comparison operators are used in subqueries: single-row operators and multiple-row operators.

Types of Subqueries
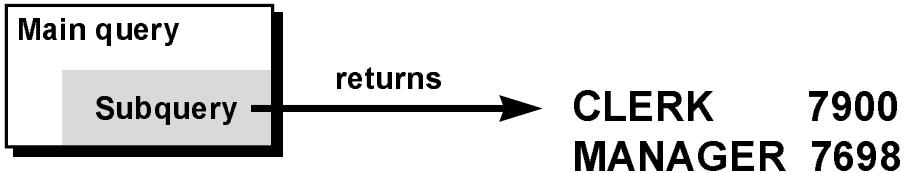
- **Single-row subquery**

Main query	
Subquery	

returns → **CLERK**

- **Multiple-row subquery**

Main query	
Subquery	

returns → **CLERK**
MANAGER

- **Multiple-column subquery**

Main query	
Subquery	

returns → **CLERK   7900**
MANAGER 7698

ORACLE®

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
- Multiple-column subqueries: Queries that return more than one column from the inner SELECT statement

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®

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
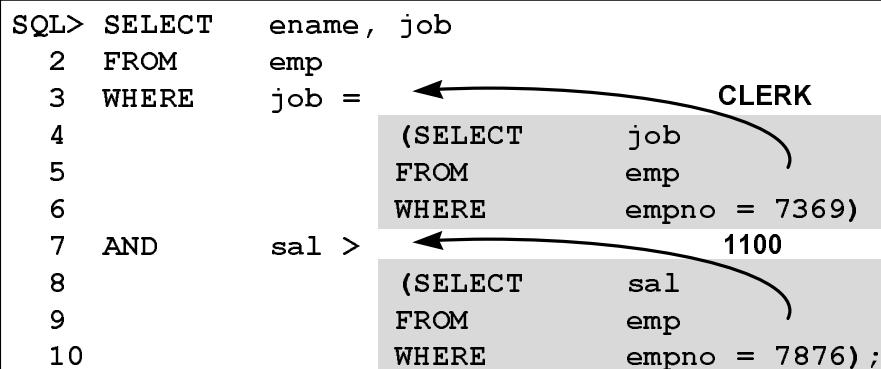
Display the employees whose job title is the same as that of employee 7369.

```
SQL> SELECT    ename, job
  2  FROM      emp
  3  WHERE     job =
  4                  (SELECT   job
  5                   FROM     emp
  6                   WHERE    empno = 7369);


ENAME      JOB
---------- ---------
JAMES      CLERK
SMITH      CLERK
ADAMS      CLERK
MILLER     CLERK
```

Executing Single-Row Subqueries

```
SQL>  SELECT    ename, job
   2  FROM      emp
   3  WHERE     job =                        CLERK
   4                   (SELECT    job
   5                    FROM      emp
   6                    WHERE     empno = 7369)
   7  AND       sal >                        1100
   8                   (SELECT    sal
   9                    FROM      emp
  10                    WHERE     empno = 7876);
```

```
ENAME        JOB
----------   ----------
MILLER       CLERK
```

ORACLE®

Executing Single-Row Subqueries

A SELECT statement can be considered as a query block. The example on the slide displays employees whose job title is the same as that of employee 7369 and whose salary is greater than that of employee 7876.

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: CLERK and 1100, 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 (CLERK and 1100, 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

```
SQL> SELECT    ename, job, sal            800
  2  FROM      emp
  3  WHERE     sal =
  4                    (SELECT        MIN(sal)
  5                     FROM          emp);
```

```
ENAME        JOB           SAL
----------   ----------   ----------
SMITH        CLERK          800
```

ORACLE®

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 operator.

The example on the slide displays the employee name, job title, and salary of all employees whose salary is equal to the minimum salary. The MIN group function returns a single value (800) to the outer query.

HAVING Clause with Subqueries

- **The Oracle Server executes subqueries first.**
- **The Oracle Server returns results into the HAVING clause of the main query.**

```
SQL>  SELECT        deptno, MIN(sal)
   2  FROM          emp
   3  GROUP BY      deptno                    800
   4  HAVING        MIN(sal) >  ◄──────────
   5                            (SELECT    MIN(sal)
   6                            FROM       emp
   7                            WHERE      deptno = 20);
```

ORACLE

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 20.

```
     DEPTNO    MIN(SAL)
  ---------- ----------
          10       1300
          30        950
```

Example

Find the job with the lowest average salary.

```
SQL>  SELECT       job, AVG(sal)
   2  FROM         emp
   3  GROUP BY     job
   4  HAVING       AVG(sal) = (SELECT        MIN(AVG(sal))
   5                           FROM          EMP
   6                           GROUP BY      job);
```

Introduction to Oracle: SQL and PL/SQL 6-11

What Is Wrong
with This Statement?

```
SQL>  SELECT  empno, ename
  2   FROM    emp
  3   WHERE   sal =
  4                    (SELECT   MIN(sal)
  5                     FROM     emp
  6                     GROUP BY deptno);
```

Single-row operator with
multiple-row subquery

```
ERROR:
ORA-01427: single-row subquery returns more than
one row

no rows selected
```

ORACLE®

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 (deptno) 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 800, 1300, and 950.

The outer query takes the results of the subquery (800, 950, 1300) 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 hence generates the error.

To correct this error, change the = operator to IN.

Will This Statement Work?

```
SQL> SELECT ename, job
  2  FROM    emp
  3  WHERE   job =
  4                  (SELECT job
  5                   FROM    emp
  6                   WHERE   ename='SMYTHE');
```

Subquery returns no values

```
no rows selected
```

ORACLE®

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 (ename='SMYTHE') clause. Presumably, the intention is to find the employee whose name is Smythe. The statement seems to be correct but selects no rows when executed.

The problem is that Smythe is misspelled. There is no employee named Smythe. 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 title equal to null and so returns no rows.

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®

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.

```
SQL> SELECT    ename, sal, deptno
  2  FROM      emp
  3  WHERE     sal IN (SELECT    MIN(sal)
  4                    FROM      emp
  5                    GROUP BY deptno);
```

Example

Find the employees who earn the same salary as the minimum salary for departments.

The inner query is executed first, producing a query result containing three rows: 800, 950, 1300. 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 look like the following to the Oracle Server:

```
SQL> SELECT    ename, sal, deptno
  2  FROM      emp
  3  WHERE     sal IN (800, 950, 1300);
```

Using ANY Operator
in Multiple-Row Subqueries

```
SQL> SELECT    empno, ename, job 1300
  2  FROM      emp                   1100
  3  WHERE     sal < ANY             800
  4                       (SELECT    950
                                     sal )
  5                       FROM       emp
  6                       WHERE      job = 'CLERK')
  7  AND       job <> 'CLERK';
```

```
    EMPNO ENAME       JOB
--------- ----------- ---------
     7654 MARTIN      SALESMAN
     7521 WARD        SALESMAN
```

ORACLE®

Multiple-Row Subqueries (continued)

The ANY operator (and its synonym SOME operator) compares a value to *each* value returned by a subquery. The slide example displays employees whose salary is less than any clerk and who are not clerks. The maximum salary that a clerk earns is $1300. The SQL statement displays all the employees who are not clerks but earn less than $1300.

<ANY means less than the maximum. >ANY means more than the minimum. =ANY is equivalent to IN.

Using ALL Operator
in Multiple-Row Subqueries

```
SQL> SELECT    empno, ename, job 1566.6667
  2  FROM      emp               2175
  3  WHERE     sal > ALL         2916.6667
  4                (SELECT       avg(sal)
  5                FROM          emp
  6                GROUP BY      deptno);
```

```
    EMPNO ENAME       JOB
--------- ----------- ---------
     7839 KING        PRESIDENT
     7566 JONES       MANAGER
     7902 FORD        ANALYST
     7788 SCOTT       ANALYST
```

ORACLE®

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 greater than the average salaries of all the departments. The highest average salary of a department is $2916.66, so the query returns those employees whose salary is greater than $2916.66.

>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.

Summary

Subqueries are useful when a query is based on unknown values.

```
SELECT    select_list
FROM      table
WHERE     expr operator
                  (SELECT select_list
                   FROM    table);
```

ORACLE®

Summary

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 Overview

- **Creating subqueries to query values based on unknown criteria**
- **Using subqueries to find out what values exist in one set of data and not in another**

ORACLE®

Practice Overview

In this practice, you will write complex queries using nested SELECT statements.

Paper-Based Questions

You may want to consider creating 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 employee name and hire date for all employees in the same department as Blake. Exclude Blake.

    ```
    ENAME       HIREDATE
    ---------- ---------
    ALLEN      20-FEB-81
    WARD       22-FEB-81
    MARTIN     28-SEP-81
    TURNER     08-SEP-81
    JAMES      03-DEC-81
       5 rows selected.
    ```

2. Create a query to display the employee number and name for all employees who earn more than the average salary. Sort the results in descending order of salary.

    ```
    EMPNO ENAME
    ----- -----------
     7839 KING
     7902 FORD
     7788 SCOTT
     7566 JONES
     7698 BLAKE
     7782 CLARK
    6 rows selected.
    ```

3. Write a query that will display the employee number and name for all employees who work in a department with any employee whose name contains a *T*. Save your SQL statement in a file called `p6q3.sql`.

    ```
     EMPNO ENAME
    ------ --------
      7566 JONES
      7788 SCOTT
      7876 ADAMS
      7369 SMITH
      7902 FORD
      7698 BLAKE
      7654 MARTIN
      7499 ALLEN
      7844 TURNER
      7900 JAMES
      7521 WARD
    11 rows selected.
    ```

Practice 6 (continued)

4. Display the employee name, department number, and job title for all employees whose department location is Dallas.

```
ENAME   DEPTNO  JOB
------  ------  ---------
JONES       20  MANAGER
FORD        20  ANALYST
SMITH       20  CLERK
SCOTT       20  ANALYST
ADAMS       20  CLERK
```

5. Display the employee name and salary of all employees who report to King.

```
ENAME       SAL
------      ----
BLAKE       2850
CLARK       2450
JONES       2975
```

6. Display the department number, name, and job for all employees in the Sales department.

```
DEPTNO  ENAME      JOB
------  --------   ---------
    30  BLAKE      MANAGER
    30  MARTIN     SALESMAN
    30  ALLEN      SALESMAN
    30  TURNER     SALESMAN
    30  JAMES      CLERK
    30  WARD       SALESMAN
6 rows selected.
```

If you have time, complete the following exercises:

7. Modify p6q3.sql to display the employee number, name, and salary for all employees who earn more than the average salary and who work in a department with any employee with a *T* in their name. Resave as p6q7.sql. Rerun your query.

```
EMPNO  ENAME      SAL
-----  ------     ----
 7566  JONES      2975
 7788  SCOTT      3000
 7902  FORD       3000
 7698  BLAKE      2850
```

7

Multiple-Column Subqueries

ORACLE®

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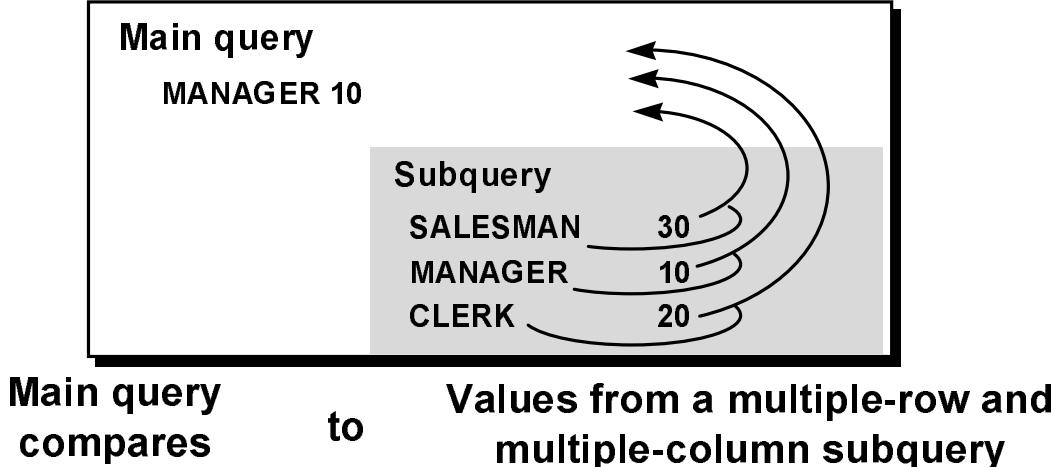
After completing this lesson, you should be able to do the following:

- **Write a multiple-column subquery**

- **Describe and explain the behavior of subqueries when null values are retrieved**

- **Write a subquery in a FROM clause**

ORACLE®

Lesson Aim

In this lesson, you will learn how to write multiple-column subqueries and subqueries in the FROM clause of a SELECT statement.

Multiple-Column Subqueries

Main query

MANAGER 10

Subquery

SALESMAN	30
MANAGER	10
CLERK	20

Main query compares	to	Values from a multiple-row and multiple-column subquery

MANAGER 10	**SALESMAN 30**
	MANAGER 10
	CLERK 20

Multiple-Column Subqueries

So far you have written single-row subqueries and multiple-row subqueries where only one column was compared in the WHERE clause or HAVING clause of the SELECT statement. If you want to compare two or more columns, you must write a compound WHERE clause using logical operators. Multiple-column subqueries enable you to combine duplicate WHERE conditions into a single WHERE clause.

Syntax

```
SELECT    column, column, ...
FROM      table
WHERE     (column, column, ...) IN
                        (SELECT column, column, ...
                         FROM   table
                         WHERE  condition);
```

Using Multiple-Column Subqueries

Display the order id, product id, and quantity of items in the item table that match both the product id and quantity of an item in order 605.

```
SQL> SELECT    ordid, prodid, qty
  2  FROM      item
  3  WHERE     (prodid, qty) IN
  4                            (SELECT prodid, qty
  5                             FROM   item
  6                             WHERE  ordid = 605)
  7  AND       ordid <> 605;
```

   ORACLE®

Using Multiple-Column Subqueries

The example on the slide is that of a multiple-column subquery because the subquery returns more than one column. It compares the values in the PRODID column and the QTY column of each candidate row in the ITEM table to the values in the PRODID column and QTY column for items in order 605.

First, execute the subquery to see the PRODID and QTY values for each item in order 605.

```
SQL> SELECT prodid, qty
  2  FROM   item
  3  WHERE  ordid = 605;

    PRODID        QTY
---------- ----------
    100861        100
    100870        500
    100890          5
    101860         50
    101863        100
    102130         10

6 rows selected.
```

Using Multiple-Column Subqueries

Display the order number, product number, and quantity of any item in which the product number and quantity match both the product number and quantity of an item in order 605.

```
SQL>  SELECT    ordid, prodid, qty
   2  FROM      item
   3  WHERE     (prodid, qty) IN
   4                           (SELECT prodid, qty
   5                            FROM    item
   6                            WHERE   ordid = 605)
   7  AND       ordid <> 605;
```

ORACLE®

Using Multiple-Column Subqueries (continued)

When the SQL statement on the slide is executed, the Oracle server compares the values in both the PRODID and QTY columns and returns those orders where the product number and quantity for *that* product match *both* the product number and quantity for an item in order 605.

The output of the SQL statement is:

```
    ORDID       PRODID       QTY
--------- ---------- ---------
      617     100861       100
      617     100870       500
      616     102130        10
```

The output shows that there are three items in other orders that contain the same product number and quantity as an item in order 605. For example, order 617 has ordered a quantity 500 of product 100870. Order 605 has also ordered a quantity 500 of product 100870. Therefore, these candidate rows are part of the output.

Column Comparisons

Pairwise	
PRODID	QTY
101863	100
100861	100
102130	10
100890	5
100870	500
101860	50

Nonpairwise	
PRODID	QTY
101863	100
100861	100
102130	10
100890	5
100870	500
101860	50

ORACLE®

Pairwise Versus Nonpairwise Comparisons

Column comparisons in a multiple-column subquery can be pairwise comparisons or nonpairwise comparisons.

The slide shows the product numbers and quantities of the items in order 605.

In the example on the previous slide, a pairwise comparison was executed in the WHERE clause. Each candidate row in the SELECT statement must have *both* the same product number and same quantity as an item in order 605. This is illustrated on the left side of the slide above. The arrows indicate that both the product number and quantity in a candidate row match a product number and quantity of an item in order 605.

A multiple-column subquery can also be a nonpairwise comparison. If you want a nonpairwise comparison (a cross product), you must use a WHERE clause with multiple conditions. A candidate row must match the multiple conditions in the WHERE clause but the values are compared individually. A candidate row must match some product number in order 605 as well as some quantity in order 605, but these values do not need to be in the same row. This is illustrated on the right side of the slide. For example, product 102130 appears in other orders, one order matching the quantity in order 605 (10), and another order having a quantity of 500. The arrows show a sampling of the various quantities ordered for a particular product.

Nonpairwise Comparison Subquery

Display the order number, product number, and quantity of any item in which the product number and quantity match any product number and any quantity of an item in order 605.

```
SQL> SELECT    ordid, prodid, qty
  2  FROM      item
  3  WHERE     prodid IN  (SELECT       prodid
  4                          FROM        item
  5                          WHERE       ordid = 605)
  6  AND       qty    IN  (SELECT       qty
  7                          FROM        item
  8                          WHERE       ordid = 605)
  9  AND       ordid <> 605;
```

 ORACLE®

Nonpairwise Comparison Subquery

The slide example does a nonpairwise comparison of the columns. It displays the order number, product number, and quantity of any item in which the product number and quantity match any product number and quantity of an item in order 605. Order 605 is not included in the output.

Nonpairwise Subquery

```
     ORDID      PRODID       QTY
---------- ---------- ----------
       609     100870          5
       616     100861         10
       616     102130         10
       621     100861         10
       618     100870         10
       618     100861         50
       616     100870         50
       617     100861        100
       619     102130        100
       615     100870        100
       617     101860        100
       621     100870        100
       617     102130        100
         . . .
16 rows selected.
```

ORACLE®

Nonpairwise Subquery

The results of the nonpairwise subquery are shown in the slide. Sixteen candidate rows in the ITEM table match the multiple conditions in the WHERE clause.

For example, an item from order 621 is returned from the SQL statement. A product in order 621 (product number 100861) matches a product in an item in order 605. The quantity for product 100861 in order 621 (10) matches the quantity in another item in order 605 (the quantity for product 102130).

Null Values in a Subquery

```
SQL> SELECT    employee.ename
  2  FROM      emp employee
  3  WHERE     employee.empno NOT IN
  4                            (SELECT manager.mgr
  5                             FROM   emp manager);
no rows selected.
```

       ORACLE®

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 eight 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 resultant 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 resultant set of a subquery will not be a problem if you are using
the IN operator. The IN operator is equivalent to =ANY. For example, to display the employees who
have subordinates, use the following SQL statement:

```
SQL> SELECT    employee.ename
  2  FROM      emp employee
  3  WHERE     employee.empno IN (SELECT manager.mgr
  4                               FROM   emp manager);

ENAME
----------
KING
...
6 rows selected.
```

Introduction to Oracle: SQL and PL/SQL 7-9

Using a Subquery
in the FROM Clause

```
SQL> SELECT   a.ename, a.sal, a.deptno, b.salavg
  2  FROM     emp a, (SELECT    deptno, avg(sal) salavg
  3                   FROM       emp
  4                   GROUP BY deptno) b
  5  WHERE    a.deptno = b.deptno
  6  AND      a.sal > b.salavg;
```

```
ENAME             SAL    DEPTNO      SALAVG
---------- ---------- ---------- ----------
KING             5000         10  2916.6667
JONES            2975         20       2175
SCOTT            3000         20       2175
...
6 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 defines a data source for that particular SELECT statement, and only that SELECT statement. The slide example displays employee names, salaries, department numbers, and average salaries for all the employees who make more than the average salary in their department.

Summary

- **A multiple-column subquery returns more than one column.**

- **Column comparisons in multiple-column comparisons can be pairwise or nonpairwise.**

- **A multiple-column subquery can also be used in the FROM clause of a SELECT statement.**

 ORACLE®

Summary

Multiple-column subqueries enable you to combine duplicate WHERE conditions into a single WHERE clause. Column comparisons in a multiple-column subquery can be pairwise comparisons or nonpairwise comparisons. You can use a subquery to define a table to be operated on by a containing query. You do this by placing the subquery in the FROM clause of the containing query as you would a table name.

Practice Overview

Creating multiple-column subqueries

   ORACLE®

Practice Overview

In this practice, you will write multiple-value subqueries.

Practice 7

1. Write a query to display the name, department number, and salary of any employee whose department number and salary match the department number and salary of any employee who earns a commission.

```
ENAME       DEPTNO    SAL
--------    ------    ------
MARTIN          30     1250
WARD            30     1250
TURNER          30     1500
ALLEN           30     1600
```

2. Display the name, department name, and salary of any employee whose salary and commission match the salary and commission of any employee located in Dallas.

```
ENAME      DNAME          SAL
-------    ---------    ------
SMITH      RESEARCH        800
ADAMS      RESEARCH       1100
JONES      RESEARCH       2975
FORD       RESEARCH       3000
SCOTT      RESEARCH       3000
```

3. Create a query to display the name, hire date, and salary for all employees who have both the same salary and commission as Scott.

   Note: Do not display SCOTT in the result set.

```
ENAME      HIREDATE       SAL
-------    ---------    ------
FORD       03-DEC-81     3000
```

4. Create a query to display the employees that earn a salary that is higher than the salary of all of the clerks. Sort the results on salary from highest to lowest.

```
ENAME        JOB          SAL
----------   ----------   ---------
KING         PRESIDENT     5000
FORD         ANALYST       3000
SCOTT        ANALYST       3000
JONES        MANAGER       2975
BLAKE        MANAGER       2850
CLARK        MANAGER       2450
ALLEN        SALESMAN      1600
TURNER       SALESMAN      1500
8 rows selected.
```

8

Producing Readable Output with SQL*Plus

ORACLE®

Objectives

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

- **Produce queries that require an input variable**

- **Customize the SQL*Plus environment**

- **Produce more readable output**

- **Create and execute script files**

- **Save customizations**
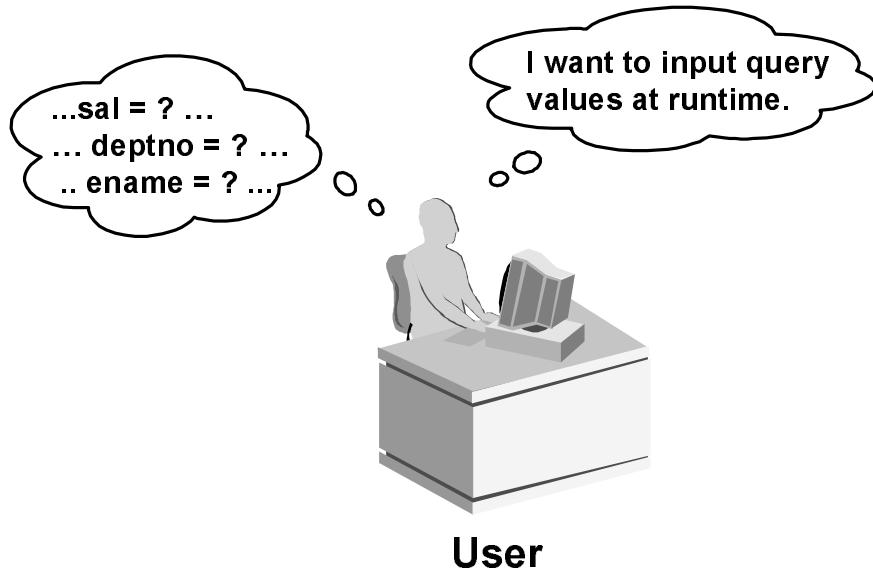
ORACLE®

Lesson Aim

In this lesson, you will learn how to include SQL*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.

Interactive Reports

...sal = ? ...
... deptno = ? ...
.. ename = ? ...

I want to input query values at runtime.

User

ORACLE®

Interactive Reports

The examples so far have not been interactive in any way. 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 SQL*Plus script file.

Using SQL*Plus, you can create reports that prompt the user to supply their own values to restrict the range of data returned. To create interactive reports, 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.

Substitution Variables

- **Use SQL*Plus substitution variables to temporarily store values.**
  - **Single ampersand (&)**
  - **Double ampersand (&&)**
  - **DEFINE and ACCEPT commands**
- **Pass variable values between SQL statements.**
- **Dynamically alter headers and footers.**

ORACLE®

Substitution Variables

In SQL*Plus, you can use single-ampersand (&) substitution variables to temporarily store values.

You can predefine variables in SQL*Plus by using the ACCEPT or DEFINE commands. ACCEPT reads a line of user input and stores it in a variable. DEFINE creates and assigns a value to a variable.

Examples of Restricted Ranges of Data

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

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

SQL*Plus does not support validation checks (except for datatype) on user input. Make sure that the prompts that you write for the user are simple and unambiguous.

Using the & Substitution Variable

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

```
SQL> SELECT    empno, ename, sal, deptno
  2  FROM      emp
  3  WHERE     empno = &employee_num;
```

```
Enter value for employee_num: 7369

    EMPNO ENAME            SAL     DEPTNO
--------- ---------- --------- ---------
     7369 SMITH            800         20
```

ORACLE®

Single-Ampersand Substitution Variable

When running a report, users often want to restrict the data returned dynamically. SQL*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
&user_variable	Indicates a variable in a SQL statement; if the variable does not exist, SQL*Plus prompts the user for a value (SQL*Plus discards a new variable once it is used.)

The example on the slide creates a SQL statement to prompt the user for an employee number at runtime and displays employee number, 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 SET VERIFY Command

Toggling the display of the text of a command before and after SQL*Plus replaces substitution variables with values.

```
SQL> SET VERIFY ON
SQL> SELECT   empno, ename, sal, deptno
  2  FROM     emp
  3  WHERE    empno = &employee_num;
```

```
Enter value for employee_num: 7369
old    3: WHERE empno = &employee_num
new    3: WHERE empno = 7369

...
```

ORACLE®

The SET VERIFY Command

To confirm the changes in the SQL statement, use the SQL*Plus SET VERIFY command. Setting SET VERIFY ON forces SQL*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 column EMPNO.

Character and Date Values with Substitution Variables

Use single quotation marks for date and character values.

```
SQL> SELECT  ename, deptno, sal*12
  2  FROM     emp
  3  WHERE    job='&job_title';
```

```
Enter value for job_title: ANALYST

ENAME          DEPTNO     SAL*12
----------     ----------  ----------
SCOTT              20       36000
FORD               20       36000
```

ORACLE®

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.

To avoid entering the quotation marks at runtime, enclose the variable in single quotation marks within the SQL statement itself.

The slide shows a query to retrieve the employee name, department number, and annual salary of all employees based on the job title entered at the prompt by the user.

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 at Runtime

Use substitution variables to supplement the following:

- **WHERE condition**
- **ORDER BY clause**
- **Column expression**
- **Table name**
- **Entire SELECT statement**

 ORACLE®

Specifying Column Names, Expressions, and Text at Runtime

Not only can you use the substitution variables in the WHERE clause of a SQL statement, but also these variables can be used to substitute column names, expressions, or text.

Example

Display the employee number and any other column and any condition of employees.

```
SQL> SELECT      empno, &column_name
  2  FROM        emp
  3  WHERE       &condition;


Enter value for column_name: job
Enter value for condition: deptno = 10

    EMPNO JOB
--------- ---------
     7839 PRESIDENT
     7782 MANAGER
     7934 CLERK
```

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

Specifying Column Names, Expressions, and Text at Runtime

```
SQL>  SELECT          empno, ename, job, &column_name
  2   FROM            emp
  3   WHERE           &condition
  4   ORDER BY        &order_column;
```

```
Enter value for column_name: sal
Enter value for condition: sal>=3000
Enter value for order_column: ename

    EMPNO ENAME        JOB             SAL
--------- ---------- --------- ---------
     7902 FORD         ANALYST        3000
     7839 KING         PRESIDENT      5000
     7788 SCOTT        ANALYST        3000
```

ORACLE®

Specifying Column Names, Expressions, and Text at Runtime (continued)

The slide example displays the employee number, name, job title, and any other column specified by the user at runtime, from the EMP table. The user can also specify the condition for retrieval of rows and the column name by which the resultant data has to be ordered.

Using the && Substitution Variable

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

```
SQL>  SELECT        empno, ename, job, &&column_name
  2   FROM          emp
  3   ORDER BY      &column_name;
```

```
Enter value for column_name: deptno
    EMPNO ENAME       JOB          DEPTNO
--------- ---------- --------- ---------
     7839 KING        PRESIDENT        10
     7782 CLARK       MANAGER          10
     7934 MILLER      CLERK            10
...
14 rows selected.
```

ORACLE®

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 (deptno) is used both for display and ordering of data.

SQL*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.

Defining User Variables

- **You can predefine variables using one of two SQL*Plus commands:**
  - **DEFINE: Create a CHAR datatype user variable**
  - **ACCEPT: Read user input and store it in a variable**
- **If you need to predefine a variable that includes spaces, you must enclose the value within single quotation marks when using the DEFINE command.**

 ORACLE®

Defining User Variables

You can predefine user variables before executing a SELECT statement. SQL*Plus provides two commands for defining and setting user variables: DEFINE and ACCEPT.

Command	Description
DEFINE *variable = value*	Creates a CHAR datatype user variable and assigns a value to it
DEFINE *variable*	Displays the variable, its value, and its datatype
DEFINE	Displays all user variables with value and datatype
ACCEPT (*see syntax on next slide*)	Reads a line of user input and stores it in a variable

The ACCEPT Command

- **Creates a customized prompt when accepting user input**

- **Explicitly defines a NUMBER or DATE datatype variable**

- **Hides user input for security reasons**

```
ACCEPT variable [datatype] [FORMAT format]
               [PROMPT text] [HIDE]
```

ORACLE®

The ACCEPT Command

In the syntax:

variable	is the name of the variable that stores the value (If it does not exist, SQL*Plus creates it.)
datatype	is NUMBER, CHAR, or DATE (CHAR has a maximum length limit of 240 bytes. DATE checks against a format model, and the datatype is CHAR.)
FOR[MAT] *format*	specifies the format model—for example, A10 or 9.999
PROMPT *text*	displays the text before the user can enter the value
HIDE	suppresses what the user enters—for example, a password

Note: Do not prefix the SQL*Plus substitution parameter with the ampersand (&) when referencing the substitution parameter in the ACCEPT command.

Using the ACCEPT Command

```
ACCEPT   dept PROMPT 'Provide the department name: '
SELECT   *
FROM     dept
WHERE    dname = UPPER('&dept')
/
```

```
Provide the department name: Sales

    DEPTNO DNAME           LOC
--------- --------------- -------------
       30 SALES           CHICAGO
```

      ORACLE®

Using the ACCEPT Command

The ACCEPT command reads in a variable named DEPT. The prompt that it displays when asking the user for the variable is "Provide the department name:". The SELECT statement then takes the department value that the user enters and uses it to retrieve the appropriate row from the DEPT table.

If the user enters a *valid* value for department name, the SELECT statement executes in the same way as any other SELECT statement, taking the user-entered value and using it in the WHERE clause to compare with DNAME.

Note that the & character does not appear with the DEPT variable in the ACCEPT command. The & appears only in the SELECT statement.

Guidelines

- The ACCEPT and DEFINE commands will create a variable if the variable does not exist; these commands will automatically redefine a variable if it exists.

- When using the DEFINE command, use single quotation marks (' ') to enclose a string that contains an embedded space.

- Use the ACCEPT command to:

    - Give a customized prompt when accepting user input; otherwise, you will see the default "Enter value for variable"

    - Explicitly define a NUMBER or DATE datatype variable

    - Hide user input for security reasons

Introduction to Oracle: SQL and PL/SQL 8-13

DEFINE and UNDEFINE Commands

- **A variable remains defined until you either:**
  - **Use the UNDEFINE command to clear it**
  - **Exit SQL*Plus**
- **You can verify your changes with the DEFINE command.**
- **To define variables for every session, modify your `login.sql` file so that the variables are created at startup.**

 ORACLE®

The DEFINE and UNDEFINE Commands

Variables are defined until you either:

- Issue the UNDEFINE command on a variable
- Exit SQL*Plus

When you undefine variables, you can verify your changes with the DEFINE command. When you exit SQL*Plus, variables defined during that session are lost. To define those variables for every session, modify your `login.sql` file so that those variables are created at startup.

Using the DEFINE Command

- ## Create a variable to hold the department name.

```
SQL> DEFINE deptname = sales
SQL> DEFINE deptname
```

```
DEFINE DEPTNAME          = "sales" (CHAR)
```

- ## Use the variable as you would any other variable.

```
SQL> SELECT  *
  2  FROM    dept
  3  WHERE   dname = UPPER('&deptname');
```

   ORACLE®

Using the DEFINE Command

You can use the DEFINE command to create a variable and then use the variable as you would any other variable. The example on the slide creates a variable DEPTNAME that contains the department name, SALES. The SQL statement then uses this variable to display the number and location of the sales department.

```
    DEPTNO  DNAME          LOC
--------- -----------    -------------
       30  SALES          CHICAGO
```

To erase the variable, you use the UNDEFINE command:

```
SQL> UNDEFINE deptname
SQL> DEFINE deptname
symbol deptname is UNDEFINED
```

Customizing the SQL*Plus Environment

- ## Use SET commands to control current session.

```
SET system_variable value
```

- ## Verify what you have set by using the SHOW command.

```
SQL> SET ECHO ON
```

```
SQL> SHOW ECHO
echo ON
```

ORACLE®

Customizing the SQL*Plus Environment

You can control the environment in which SQL*Plus is currently operating by using the SET commands.

In the syntax:

system_variable   is a variable that controls one aspect of the session environment

value   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 *SQL*Plus User's Guide and Reference*, Release 8, "Command Reference."

SET Command Variables

- **ARRAYSIZE {20 | *n*}**

- **COLSEP {_ | *text*}**

- **FEEDBACK {6 | *n* |OFF | ON}**

- **HEADING {OFF | ON}**

- **LINESIZE {80 | *n*}**

- **LONG {80 | *n*}**

- **PAGESIZE {24 | *n*}**

- **PAUSE {OFF | ON | *text*}**

- **TERMOUT {OFF | ON}**

  ORACLE®

SET Command Variables

SET Variable and Values	Description
ARRAY[SIZE] {20| *n*}	Sets the database data fetch size
COLSEP {_|*text*}	Sets text to be printed between columns (The default is single space.)
FEED[BACK] {6|*n*|OFF|ON}	Displays the number of records returned by a query when the query selects at least *n* records
HEA[DING] {OFF|ON}	Determines whether column headings are displayed in reports
LIN[ESIZE] {80|*n*}	Sets the number of characters per line to *n* for reports
LONG {80|*n*}	Sets the maximum width for displaying LONG values
PAGES[IZE] {24|*n*}	Specifies the number of lines per page of output
PAU[SE] {OFF|ON|*text*}	Allows you to control scrolling of your terminal (You must press [Return] after seeing each pause.)
TERM[OUT] {OFF|ON}	Determines whether output is displayed on screen

Note: The value *n* represents a numeric value. The underlined values indicate default values. If you enter no value with the variable, SQL*Plus assumes the default value.

Saving Customizations in the `login.sql` File

- **The `login.sql` file contains standard SET and other SQL*Plus commands that are implemented at login.**

- **You can modify `login.sql` to contain additional SET commands.**

Default Settings Using the `login.sql` File

The `login.sql` file contains standard SET and other SQL*Plus commands that you may require for every session. The file is read and commands are implemented at login. When you log out of your session, all customized settings are lost.

Changing the Default Settings

The settings implemented by `login.sql` can be changed during the current session. Changes made are current only for that session. As soon as you log out, those settings are lost.

Add permanent changes to settings to the `login.sql` file.

SQL*Plus Format Commands

- COLUMN [*column option*]
- TTITLE [*text* | OFF | ON]
- BTITLE [*text* | OFF | ON]
- BREAK [ON *report_element*]

ORACLE®

Obtaining More Readable Reports

You can control the report features by using the following commands:

Command	Description
COL[UMN] [*column option*]	Controls column formats
TTI[TLE] [*text*|OFF|ON]	Specifies a header to appear at the top of each page
BTI[TLE] [*text*|OFF|ON]	Specifies a footer to appear at the bottom of each page of the report
BRE[AK] [ON *report_element*]	Suppresses duplicate values and sections rows of data with line feeds

Guidelines

- All format commands remain in effect until the end of the SQL*Plus session or until the format setting is overwritten or cleared.
- Remember to reset your SQL*Plus settings to default values after every report.
- There is no command for setting a SQL*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.

Introduction to Oracle: SQL and PL/SQL 8-19

The COLUMN Command

Controls display of a column

```
COL[UMN] [{column|alias} [option]]
```

- **CLE[AR]: Clears any column formats**
- **FOR[MAT]** *format*: **Changes the display of the column using a format model**
- **HEA[DING]** *text*: **Sets the column heading**
- **JUS[TIFY] {*align*}: Aligns the column heading to be left, center, or right**

 ORACLE®

COLUMN Command Options

Option	Description
CLE[AR]	Clears any column formats
FOR[MAT] *format*	Changes the display of the column data
HEA[DING] *text*	Sets the column heading (A vertical line (|) will force a line feed in the heading if you do not use justification.)
JUS[TIFY] {*align*}	Justifies the column heading (not the data) to be left, center, or right
NOPRI[NT]	Hides the column
NUL[L] *text*	Specifies text to be displayed for null values
PRI[NT]	Shows the column
TRU[NCATED]	Truncates the string at the end of the first line of display
WRA[PPED]	Wraps the end of the string to the next line

Using the COLUMN Command

- ## Create column headings.

```
COLUMN ename HEADING 'Employee|Name' FORMAT A15
COLUMN sal JUSTIFY LEFT FORMAT $99,990.00
COLUMN mgr FORMAT 999999999 NULL 'No manager'
```

- ## Display the current setting for the ENAME column.

```
COLUMN ename
```

- ## Clear settings for the ENAME column.

```
COLUMN ename CLEAR
```

ORACLE®

Displaying or Clearing Settings

To show or clear the current COLUMN command settings, use the following commands:

Command	Description
COL[UMN] *column*	Displays the current settings for the specified column
COL[UMN]	Displays the current settings for all columns
COL[UMN] column CLE[AR]	Clears the settings for the specified column
CLE[AR] COL[UMN]	Clears the settings for all columns

If you have a lengthy command, you can continue it on the next line by ending the current line with a hyphen (-).

COLUMN Format Models

Element	Description	Example	Result
A*n*	Sets a display width of *n*	N/A	N/A
9	Single zero-suppression digit	999999	1234
0	Enforces leading zero	099999	01234
$	Floating dollar sign	$9999	$1234
L	Local currency	L9999	L1234
.	Position of decimal point	9999.99	1234.00
,	Thousand separator	9,999	1,234

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

Suppresses duplicates and sections rows

• To suppress duplicates

```
SQL> BREAK ON ename ON job
```

• To section out rows at break values

```
SQL> BREAK ON ename SKIP 4 ON job SKIP 2
```

ORACLE®

The BREAK Command

Use the BREAK command to section out rows 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] [skip n|dup|page] on .. [on report]
```

where:	*page*	throws a new page when the break value changes
	skip n	skips *n* number of lines when the break value changes
		Breaks can be active on:
		− Column
		− Row
		− Page
		− Report
	duplicate	displays duplicate values

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.

```
SQL> TTITLE 'Salary|Report'
```

• Set the report footer.

```
SQL> BTITLE 'Confidential'
```

ORACLE®

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 according to the PAGESIZE value.

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.

In the syntax:

text   represents the title text (Enter single quotes if the text is more than one word.)

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 date and page number on the report.

Note: The slide gives an abridged syntax for TTITLE and BTITLE. Various options for TTITLE and BTITLE are covered in another SQL course.

Creating a Script File
to Run a Report

1. Create the SQL SELECT statement.

2. Save the SELECT statement to 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®

Creating a Script File to Run a Report

You can either enter each of the SQL*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 SQL*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 SQL*Plus commands.

4. Add the required formatting commands before the SELECT statement. Be certain not to place SQL*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. Enter "START *filename*" to run the script.

ORACLE®

How to Create a Script File (continued)

6. Add the format-clearing SQL*Plus commands after the run character. As an alternative, you can call a reset file that contains all the format-clearing commands.

7. Save the script file with your changes.

8. In SQL*Plus, run the script file by entering START *filename* or @ *filename*. This command is required to read and execute the script file.

Guidelines

- You can include blank lines between SQL*Plus commands in a script.
- You can abbreviate SQL*Plus commands.
- Include reset commands at the end of the file to restore the original SQL*Plus environment.

REM represents a remark or comment in SQL*Plus.

Sample Report

```
Fri Oct 24                                    page    1
                        Employee
                         Report

Job
Category         Employee              Salary
-------------------- ---------------------- ----------------
CLERK            ADAMS                $1,100.00
                 JAMES                  $950.00
                 MILLER               $1,300.00
                 SMITH                  $800.00
MANAGER          BLAKE                $2,850.00
                 CLARK                $2,450.00
                 JONES                $2,975.00
SALESMAN         ALLEN                $1,600.00
                 MARTIN               $1,250.00
                 TURNER               $1,500.00
                 WARD                 $1,250.00


                      Confidential
```

       ORACLE®

Example

Create a script file to create a report that displays the job title, name, and salary for every employee whose salary is less than $3000. Add a centered, two-lined 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.

```
SET PAGESIZE 37
SET LINESIZE 60
SET FEEDBACK OFF
TTITLE 'Employee|Report'
BTITLE 'Confidential'
BREAK ON job
COLUMN job    HEADING 'Job|Category' FORMAT A15
COLUMN ename  HEADING 'Employee' FORMAT A15
COLUMN sal    HEADING 'Salary' FORMAT $99,999.99
REM ** Insert SELECT statement
SELECT          job, ename, sal
FROM            emp
WHERE           sal < 3000
ORDER BY        job, ename
/
SET FEEDBACK ON
REM clear all formatting commands ...
```

Summary

- **Use SQL*Plus substitution variables to temporarily store values.**

- **Use SET commands to control current SQL*Plus environment.**

- **Use the COLUMN command to control the display of a column.**

- **Use the BREAK command to suppress duplicates and section rows.**

- **Use TTITLE and BTITLE to display headers and footers.**

 ORACLE ®

Summary

Substitution variables are useful for running reports. They allow 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
- The ACCEPT command
- 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 Overview

- **Creating a query to display values using substitution variables**
- **Starting a command file containing variables**
- **Using the ACCEPT command**

ORACLE®

Practice Overview

This practice gives you the opportunity to create files that can be run interactively by using substitution variables to create runtime selection criteria.

Practice 8

Determine whether the following statements are true or false:

1. A single ampersand substitution variable prompts at most once.
   True/False

2. The ACCEPT command is a SQL command.
   True/False

3. Write a script file to display the employee 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. Prompt the user for the two ranges using the ACCEPT command. Use the format MM/DD/YYYY. Save the script file as p8q3.sql.

```
Please enter the low date range ('MM/DD/YYYY'): 01/01/1981
Please enter the high date range ('MM/DD/YYYY'): 01/01/1982
EMPLOYEES          HIREDATE

----------------   ---------------

KING, PRESIDENT    17-NOV-81
BLAKE, MANAGER     01-MAY-81
CLARK, MANAGER     09-JUN-81
JONES, MANAGER     02-APR-81
MARTIN, SALESMAN   28-SEP-81
ALLEN, SALESMAN    20-FEB-81
TURNER, SALESMAN   08-SEP-81
JAMES, CLERK       03-DEC-81
WARD, SALESMAN     22-FEB-81
FORD, ANALYST      03-DEC-81
10 rows selected.
```

4. Write a script to display the employee name, job, and department name for a given location. The search condition should allow for case-insensitive searches of the department location. Save the script file as p8q4.sql.

```
Please enter the location name: Dallas
EMPLOYEE NAME  JOB           DEPARTMENT NAME

-------------  ------------  ---------------

JONES          MANAGER       RESEARCH
FORD           ANALYST       RESEARCH
SMITH          CLERK         RESEARCH
SCOTT          ANALYST       RESEARCH
ADAMS          CLERK         RESEARCH
```

Practice 8 (continued)

5. Modify `p8q4.sql` to create a report containing the department name, employee name, hire date, salary, and each employee's annual salary for all employees in a given location. Prompt the user for the location. Label the columns DEPARTMENT NAME, EMPLOYEE NAME, START DATE, SALARY, and ANNUAL SALARY, placing the labels on multiple lines. Resave the script as `p8q5.sql`.

```
Please enter the location name: Chicago
DEPARTMENT EMPLOYEE    START                      ANNUAL
NAME       NAME        DATE         SALARY        SALARY
---------- ----------- ---------  ----------  ----------
SALES      BLAKE       01-MAY-81  $2,850.00   $34,200.00
           MARTIN      28-SEP-81  $1,250.00   $15,000.00
           ALLEN       20-FEB-81  $1,600.00   $19,200.00
           TURNER      08-SEP-81  $1,500.00   $18,000.00
           JAMES       03-DEC-81    $950.00   $11,400.00
           WARD        22-FEB-81  $1,250.00   $15,000.00
```

9

Manipulating Data

ORACLE®

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

- Control transactions

ORACLE®

Lesson Aim

In this lesson, you will learn how to insert rows into a table, update existing rows in a table, and
delete existing rows from a table. You will 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®

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.

Adding a New Row to a Table

	50	DEVELOPMENT	DETROIT

New row

DEPT

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

"...insert a new row into DEPT table..."

DEPT

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON
50	DEVELOPMENT	DETROIT

ORACLE®

Adding a New Row to a Table

The slide graphic adds a new department to the DEPT table.

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.**

     ORACLE®

Adding a New Row to a Table (continued)

You can add new rows to a table by issuing the INSERT statement.

In the syntax:

table	is the name of the table
column	is the name of the column in the table to populate
value	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.**

```
SQL> INSERT INTO    dept (deptno, dname, loc)
  2  VALUES         (50, 'DEVELOPMENT', 'DETROIT');
1 row created.
```

- **Enclose character and date values within single quotation marks.**

     ORACLE®

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.

```
SQL> DESCRIBE   dept

Name                              Null?     Type
--------------------------------- --------  ------------
DEPTNO                            NOT NULL  NUMBER(2)
DNAME                                       VARCHAR2(14)
LOC                                         VARCHAR2(13)
```

For clarity, use the column list in the INSERT clause.
Enclose character and date values within single quotation marks; do not enclose numeric values within single quotation marks.

Inserting Rows with Null Values

- ## Implicit method: Omit the column from the column list.

```
SQL> INSERT INTO    dept (deptno, dname )
  2  VALUES         (60, 'MIS');
1 row created.
```

- ## Explicit method: Specify the NULL keyword.

```
SQL> INSERT INTO    dept
  2  VALUES         (70, 'FINANCE', NULL);
1 row created.
```

ORACLE®

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 only.

Be sure that the targeted column allows null values by verifying the Null? status from the SQL*Plus DESCRIBE command.

The Oracle Server automatically enforces all datatypes, data ranges, and data integrity constraints. Any column that is not listed explicitly obtains a null value in the new row.

Inserting Special Values

The SYSDATE function records the current date and time.

```
SQL> INSERT INTO    emp (empno, ename, job,
  2                 mgr, hiredate, sal, comm,
  3                 deptno)
  4  VALUES         (7196, 'GREEN', 'SALESMAN',
  5                 7782, SYSDATE, 2000, NULL,
  6                 10);
1 row created.
```

         ORACLE®

Inserting Special Values by Using SQL Functions

You can use pseudocolumns to enter special values in your table.

The slide example records information for employee Green in the EMP table. It supplies the current date and time in the HIREDATE 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

```
SQL> SELECT   empno, ename, job, hiredate, comm
  2  FROM     emp
  3  WHERE    empno = 7196;


    EMPNO ENAME      JOB        HIREDATE    COMM
--------- ---------- --------- --------- ---------
     7196 GREEN      SALESMAN  01-DEC-97
```

Inserting Specific Date Values

• Add a new employee.

```
SQL> INSERT INTO emp
  2  VALUES      (2296,'AROMANO','SALESMAN',7782,
  3              TO_DATE('FEB 3, 1997', 'MON DD, YYYY'),
  4              1300, NULL, 10);
1 row created.
```

• Verify your addition.

```
EMPNO ENAME    JOB       MGR    HIREDATE   SAL  COMM DEPTNO
----- -------  --------  ----  ----------  ----  ---- ------
 2296 AROMANO  SALESMAN  7782  03-FEB-97   1300           10
```

ORACLE®

Inserting Specific Date and Time Values

The format DD-MON-YY 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, another century and/or a specific time—you must use the TO_DATE function.

The example on the slide records information for employee Aromano in the EMP table. It sets the HIREDATE column to be February 3, 1997.

If the RR format is set, the century may not be the current one.

Inserting Values by Using Substitution Variables

Create an interactive script by using SQL*Plus substitution parameters.

```
SQL> INSERT INTO    dept (deptno, dname, loc)
  2    VALUES        (&department_id,
  3                  '&department_name', '&location');
```

```
Enter value for department_id: 80
Enter value for department_name: EDUCATION
Enter value for location: ATLANTA

1 row created.
```

ORACLE®

Inserting Values by Using Substitution Variables

You can produce an INSERT statement that allows the user to add values interactively by using SQL*Plus substitution variables.

The slide example records information for a department in the DEPT table. It prompts the user for the department number, department name, and location.

For date and character values, the ampersand and the variable name are enclosed in single quotation marks.

Creating a Script
with Customized Prompts

- **ACCEPT stores the value in a variable.**
- **PROMPT displays your customized text.**

```
ACCEPT          department_id PROMPT 'Please enter the -
                department number:'
ACCEPT          department_name PROMPT 'Please enter -
                the department name:'
ACCEPT          location PROMPT 'Please enter the -
                location:'
INSERT INTO     dept (deptno, dname, loc)
VALUES          (&department_id, '&department_name',
                '&location');
```

     ORACLE®

Creating a Script to Manipulate Data

You can save your command with substitution variables to a file and execute the file. Each time you execute the command, it will prompt you for new values. Customize the prompts by using the SQL*Plus ACCEPT command.

The example on the slide records information for a department in the DEPT table. It prompts the user for the department number, department name, and location by using customized prompt messages.

```
Please enter the department number: 90
Please enter the department name: PAYROLL
Please enter the location: HOUSTON

1 row created.
```

Do not prefix the SQL*Plus substitution parameter with the ampersand (&) when referencing it in the ACCEPT command. Use a dash (-) to continue a SQL*Plus command on the next line.

Copying Rows
from Another Table

- **Write your INSERT statement with a subquery.**

```
SQL> INSERT INTO managers(id, name, salary, hiredate)
  2               SELECT empno, ename, sal, hiredate
  3               FROM   emp
  4               WHERE  job = 'MANAGER';
3 rows created.
```

- **Do not use the VALUES clause.**
- **Match the number of columns in the INSERT clause to those in the subquery.**

     ORACLE®

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;
```

where:	table	is the table name
	column	is the name of the column in the table to populate
	subquery	is the subquery that returns rows into the table

For more information, see *Oracle Server SQL Reference*, Release 8, "SELECT," Subqueries section.

The number of columns and their datatypes in the column list of the INSERT clause must match the number of values and their datatypes in the subquery.

Changing Data in a Table

EMP

EMPNO	ENAME	JOB	. . .	DEPTNO
7839	KING	PRESIDENT		10
7698	BLAKE	MANAGER		30
7782	CLARK	MANAGER		10
7566	JONES	MANAGER		20
. . .				

"...update a row in EMP table..."

EMP

EMPNO	ENAME	JOB	. . .	DEPTNO
7839	KING	PRESIDENT		10
7698	BLAKE	MANAGER		30
7782	CLARK	MANAGER		20
7566	JONES	MANAGER		20
. . .				

ORACLE®

Changing Data in a Table

The slide graphic changes the department number for Clark from 10 to 20.

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.**

       ORACLE®

Updating Rows

You can modify existing rows by using the UPDATE statement.

In the above syntax:

table	is the name of the table
column	is the name of the column in the table to populate
value	is the corresponding value or subquery for the column
condition	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 *Oracle Server SQL Reference*, Release 8, "UPDATE."

Note: In general, use the primary key to identify a single row. Using other columns may unexpectedly cause several rows to be updated. For example, identifying a single row in the EMP 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 when you specify the WHERE clause.**

```
SQL> UPDATE    emp
  2  SET       deptno = 20
  3  WHERE     empno = 7782;
1 row updated.
```

- **All rows in the table are modified if you omit the WHERE clause.**

```
SQL> UPDATE    employee
  2  SET       deptno = 20;
14 rows updated.
```

ORACLE®

Updating Rows (continued)

The UPDATE statement modifies specific rows, if the WHERE clause is specified. The slide example transfers employee 7782 (Clark) to department 20.

If you omit the WHERE clause, all the rows in the table are modified.

```
SQL> SELECT    ename, deptno
  2  FROM      employee;


ENAME           DEPTNO
---------- ----------
KING                20
BLAKE               20
CLARK               20
JONES               20
MARTIN              20
ALLEN               20
TURNER              20
...
14 rows selected.
```

Note: The EMPLOYEE table has the same data as the EMP table.

Updating with
Multiple-Column Subquery

Update employee 7698's job and department to match that of employee 7499.

```
SQL> UPDATE    emp
  2  SET       (job, deptno) =
  3                       (SELECT job, deptno
  4                        FROM    emp
  5                        WHERE   empno = 7499)
  6  WHERE    empno = 7698;
1 row updated.
```

       ORACLE®

Updating Rows with a Multiple-Column Subquery

Multiple-column subqueries can be implemented in the SET clause of an UPDATE statement.

Syntax

```
UPDATE  table
SET     (column, column, ...) =
                 (SELECT  column, column, ...
                  FROM    table
                  WHERE   condition)
WHERE   condition;
```

Updating Rows Based
on Another Table

**Use subqueries in UPDATE statements to
update rows in a table based on values
from another table.**

```
SQL> UPDATE    employee
  2  SET       deptno = (SELECT    deptno
  3                         FROM      emp
  4                         WHERE     empno = 7788)
  5  WHERE     job    = (SELECT    job
  6                         FROM      emp
  7                         WHERE     empno = 7788);
2 rows updated.
```

ORACLE®

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 EMPLOYEE table based on the values from the EMP table. It changes the department number of all employees with employee 7788's job title to employee 7788's current department number.

Updating Rows:
Integrity Constraint Error

```
SQL> UPDATE    emp
  2  SET       deptno = 55
  3  WHERE     deptno = 10;
```

Department number 55 does not exist

```
UPDATE emp
       *
ERROR at line 1:
ORA-02291: integrity constraint (USR.EMP_DEPTNO_FK)
violated - parent key not found
```

 ORACLE®

Integrity Constraint Error

If you attempt to update a record with a value that is tied to an integrity constraint, you will experience an error.

In the example on the slide, department number 55 does not exist in the parent table, DEPT, 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 will cover integrity constraints in greater depth.

Removing a Row from a Table

DEPT

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON
50	DEVELOPMENT	DETROIT
60	MIS	
...		

"...delete a row from DEPT table..."

DEPT

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON
60	MIS	
...		

ORACLE®

Removing a Row from a Table

The slide graphic removes the DEVELOPMENT department from the DEPT table (assuming that there are no constraints defined on the DEPT table).

The DELETE Statement

You can remove existing rows from a table by using the DELETE statement.

```
DELETE [FROM]     table
[WHERE            condition];
```

ORACLE®

Deleting Rows

You can remove existing rows by using the DELETE statement.

In the syntax:

table	is the table name
condition	identifies the rows to be deleted and is composed of column names, expressions, constants, subqueries, and comparison operators

For more information, see *Oracle Server SQL Reference*, Release 8, "DELETE."

Deleting Rows from a Table

- **Specific rows are deleted when you specify the WHERE clause.**

```
SQL> DELETE FROM    department
  2  WHERE          dname = 'DEVELOPMENT';
1 row deleted.
```

- **All rows in the table are deleted if you omit the WHERE clause.**

```
SQL> DELETE FROM    department;
4 rows deleted.
```

ORACLE®

Deleting Rows (continued)

You can delete specific rows by specifying the WHERE clause in the DELETE statement. The slide example deletes the DEVELOPMENT department from the DEPARTMENT table. You can confirm the delete operation by displaying the deleted rows using the SELECT statement.

```
SQL> SELECT  *
  2  FROM    department
  3  WHERE   dname = 'DEVELOPMENT';
no rows selected.
```

Example

Remove all employees who started after January 1, 1997.

```
SQL> DELETE FROM  emp
  2  WHERE        hiredate > TO_DATE('01.01.1997', 'DD.MM.YYYY');
1 row deleted.
```

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 DEPARTMENT table because no WHERE clause has been specified.

Note: The DEPARTMENT table has the same data as the DEPT table.

Deleting Rows Based
on Another Table

Use subqueries in DELETE statements to remove rows from a table based on values from another table.

```
SQL> DELETE FROM     employee
  2  WHERE          deptno =
  3                      (SELECT    deptno
  4                       FROM      dept
  5                       WHERE     dname ='SALES');
6 rows deleted.
```

ORACLE®

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 department 30. The subquery searches the DEPT table to find the department number for the SALES department. The subquery then feeds the department number to the main query, which deletes rows of data from the EMPLOYEE table based on this department number.

Deleting Rows:
Integrity Constraint Error

```
SQL> DELETE FROM    dept
  2  WHERE          deptno = 10;
```

You cannot delete a row that contains a primary key that is used as a foreign key in another table.

```
DELETE FROM dept
            *
ERROR at line 1:
ORA-02292: integrity constraint (USR.EMP_DEPTNO_FK)
violated - child record found
```

ORACLE®

Integrity Constraint Error

If you attempt to delete a record with a value that is tied to an integrity constraint, you will experience an error.

The example on the slide tries to delete department number 10 from the DEPT table, but it results in an error because department number is used as a foreign key in the EMP table. If the parent record that you attempt to delete has child records, then you receive the *child record found* violation ORA-02292.

Database Transactions

Consist of one of the following statements:

- ## DML statements that make up one consistent change to the data
- ## One DDL statement
- ## One DCL statement

ORACLE®

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 executable SQL statement is executed**
- **End with one of the following events:**
  - **COMMIT or ROLLBACK is issued**
  - **DDL or DCL statement executes (automatic commit)**
  - **User exits**
  - **System crashes**

When Does a Transaction Start and End?

A transaction begins when the first executable SQL statement is encountered and terminates when one of the following occurs:

- A COMMIT or ROLLBACK statement is issued
- A DDL statement, such as CREATE, is issued
- A DCL statement is issued
- The user exits SQL*Plus
- A machine fails or the system crashes
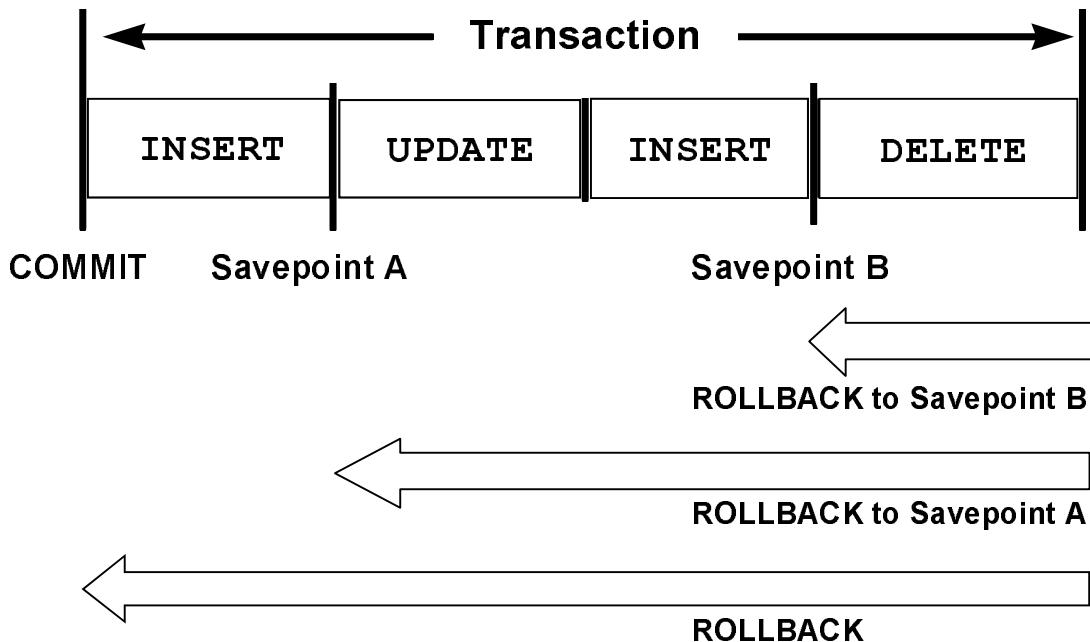
After one transaction ends, the next executable SQL statement automatically starts the next transaction.

A DDL statement or a DCL statement is automatically committed and therefore implicitly ends a transaction.

Advantages of COMMIT and ROLLBACK Statements

- **Ensure data consistency**
- **Preview data changes before making changes permanent**
- **Group logically related operations**

ORACLE®

Controlling Transactions

| INSERT | UPDATE | INSERT | DELETE |

COMMIT   Savepoint A   Savepoint B

ROLLBACK to Savepoint B

ROLLBACK to Savepoint A

ROLLBACK

       ORACLE®

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 *name*	Marks a savepoint within the current transaction
ROLLBACK [TO *SAVEPOINT name*]	ROLLBACK ends the current transaction by discarding all pending data changes; ROLLBACK TO SAVEPOINT rolls back the current transaction to the specified savepoint, thereby discarding the savepoint and any subsequent changes. If you omit this clause, the ROLLBACK statement rolls back the entire transaction.

Note: SAVEPOINT is not ANSI standard SQL.

Implicit Transaction Processing

- **An automatic commit occurs under the following circumstances:**
  - **DDL statement is issued**
  - **DCL statement is issued**
  - **Normal exit from SQL*Plus, without explicitly issuing COMMIT or ROLLBACK**
- **An automatic rollback occurs under an abnormal termination of SQL*Plus or a system failure.**

ORACLE®

Implicit Transaction Processing

Status	Circumstances
Automatic commit	DDL statement or DCL statement is issued Normal exit from SQL*Plus, without explicitly issuing COMMIT or ROLLBACK
Automatic rollback	Abnormal termination of SQL*Plus or system failure

Note: A third command is available in SQL*Plus. The AUTOCOMMIT command can be toggled ON or OFF. If set to ON, each individual DML statement is committed as soon as it is executed. You cannot roll back the changes. If set to OFF, COMMIT can be issued explicitly. Also, COMMIT is issued when a DDL statement is issued or when you exit from SQL*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.

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®

Committing Changes

Every data change made during the transaction is temporary until the transaction is committed.

State of the data before COMMIT or ROLLBACK is 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.

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®

Committing Changes (continued)

Make all pending changes permanent by using the COMMIT statement. Following a COMMIT:

State of the data after a COMMIT is issued:

- 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.**

```
SQL> UPDATE    emp
  2  SET       deptno = 10
  3  WHERE     empno = 7782;
1 row updated.
```

- **Commit the changes.**

```
SQL> COMMIT;
Commit complete.
```

     ORACLE®

Committing Changes (continued)

The slide example updates the EMP table and sets the department number for employee 7782 (Clark) to 10. It then makes the change permanent by issuing the COMMIT statement.

Example

Create a new ADVERTISING department with at least one employee. Make the data change permanent.

```
SQL> INSERT INTO department(deptno, dname, loc)
  2  VALUES        (50, 'ADVERTISING', 'MIAMI');
1 row created.

SQL> UPDATE    employee
  2  SET       deptno = 50
  3  WHERE     empno = 7876;
1 row updated.

SQL> COMMIT;
Commit complete.
```

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.**

```
SQL> DELETE FROM    employee;
14 rows deleted.
SQL> ROLLBACK;
Rollback complete.
```

Copyright © Oracle Corporation, 1999. All rights reserved. **ORACLE**®

Rolling Back Changes

Discard all pending changes by using the ROLLBACK statement. Following a ROLLBACK:

- 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.

```
SQL> DELETE FROM  test;
25,000 rows deleted.
SQL> ROLLBACK;
Rollback complete.
SQL> DELETE FROM  test
  2  WHERE        id = 100;
1 row deleted.
SQL> SELECT   *
  2  FROM      test
  3  WHERE    id = 100;
No rows selected.
SQL> COMMIT;
Commit complete.
```

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.**

```
SQL> UPDATE...
SQL> SAVEPOINT update_done;
Savepoint created.
SQL> INSERT...
SQL> ROLLBACK TO update_done;
Rollback complete.
```

Rolling Back Changes to a Savepoint

You can create a marker in the current transaction by using the SAVEPOINT statement. The transaction therefore can be divided 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.

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®

Statement-Level Rollback

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 will not be 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.

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®

Read Consistency

Database users make two types of access to the database:
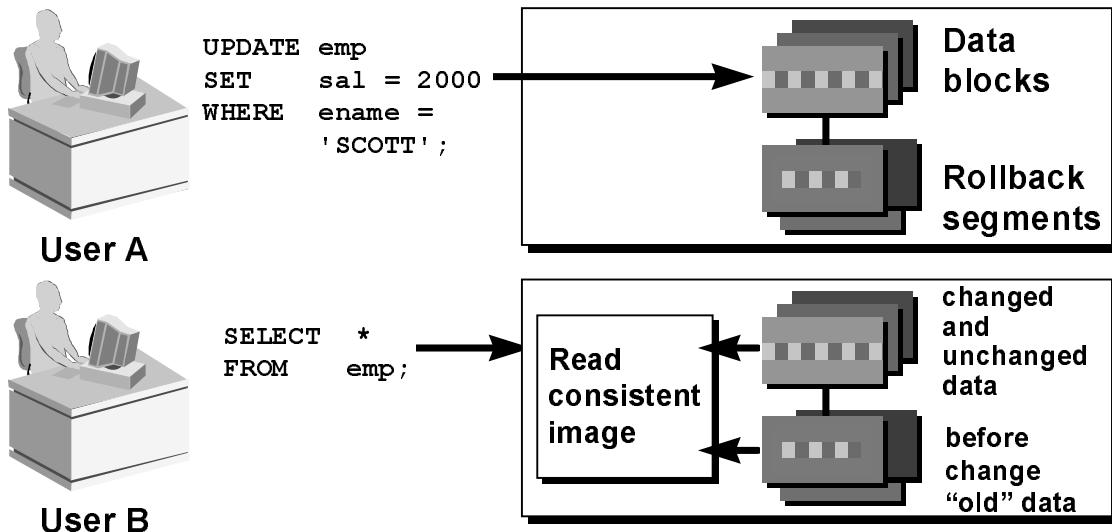
- Read operations (SELECT statement)
- Write operations (INSERT, UPDATE, DELETE statements)

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.

Implementation of Read Consistency

ORACLE®

Implementation of Read Consistency

Read consistency is an automatic implementation. It keeps a partial copy of the database in rollback 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 *rollback 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 segments' "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 rollback 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 rollback 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 rollback segment is written back to the table.
- All users see the database as it existed before the transaction began.

Locking

Oracle 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**
- **Have two basic modes:**
  - **Exclusive**
  - **Share**

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 system objects not visible to users (such as shared data structures and data dictionary rows).

How Oracle Locks Data

Locking in an Oracle database is fully automatic and requires no user action. Implicit locking occurs for all SQL statements except SELECT. The Oracle default locking mechanism automatically uses the lowest applicable level of restrictiveness, thus providing the highest degree of concurrency and maximum data integrity. Oracle also allows the user to lock data manually.

Locking Modes

Oracle uses two modes of locking in a multiuser database:

Lock Mode	Description
exclusive	Prevents a resource from being shared. The first transaction to lock a resource exclusively, is the only transaction that can alter the resource until the exclusive lock is released.
share lock	Allows the resource to be shared. Multiple users reading data can share the data, holding share locks to prevent concurrent access by a writer (who needs an exclusive lock). Several transactions can acquire share locks on the same resource.

Summary

Statement	Description
INSERT	Adds a new row to the table
UPDATE	Modifies existing rows in the table
DELETE	Removes existing rows from the table
COMMIT	Makes all pending changes permanent
SAVEPOINT	Allows a rollback to the savepoint marker
ROLLBACK	Discards all pending data changes

ORACLE®

Summary

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 Overview

- **Inserting rows into the tables**
- **Updating and deleting rows in the table**
- **Controlling transactions**

ORACLE®

Practice Overview

In this practice, you will add rows to the MY_EMPLOYEE table, update and delete data from the table, and control your transactions.

Practice 9

Insert data into the MY_EMPLOYEE table.

1. Run the `lab9_1.sql` script to build the MY_EMPLOYEE table that will 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	795
2	Dancs	Betty	bdancs	860
3	Biri	Ben	bbiri	1100
4	Newman	Chad	cnewman	750
5	Ropeburn	Audry	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     795
  2 Dancs        Betty      bdancs     860
```

Practice 9 (continued)

6. Create a script named `loademp.sql` to load rows into the MY_EMPLOYEE table interactively. Prompt the user for the employee's id, first name, last name, and salary. Concatenate the first letter of the first name and the first seven characters of the last name to produce the userid.

7. Populate the table with the next two rows of sample data by running the script that you created.

8. Confirm your additions to the table.

```
 ID  LAST_NAME   FIRST_NAME USERID   SALARY
 --- ----------- ---------- ------   ------

   1 Patel       Ralph      rpatel      795
   2 Dancs       Betty      bdancs      860
   3 Biri        Ben        bbiri      1100
   4 Newman      Chad       cnewman     750
```

9. Make the data additions permanent.

Update and delete data in the MY_EMPLOYEE table.

10. Change the last name of employee 3 to Drexler.

11. Change the salary to 1000 for all employees with a salary less than 900.

12. Verify your changes to the table.

```
LAST_NAME SALARY
--------- ------
Patel       1000
Dancs       1000
Drexler     1100
Newman      1000
```

13. Delete Betty Dancs from the MY_EMPLOYEE table.

14. 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 9 (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 running the script that you created in step 6.

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   Audry       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   Audry      aropebur    1550
```

23. Make the data addition permanent.

10

Creating and Managing Tables

ORACLE®

Objectives

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

- **Describe the main database objects**

- **Create tables**

- **Describe the datatypes that can be used
  when specifying column definition**

- **Alter table definitions**

- **Drop, rename, and truncate tables**

Lesson Aim

In this lesson, you will learn about main database objects and their relationships to each other. You
will 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	Generates primary key values
Index	Improves the performance of some queries
Synonym	Gives alternative names to objects

 ORACLE®

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: Generates primary key values
- Index: Improves the performance of some queries
- Synonym: Gives alternative names to objects

Oracle8 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.

Naming Conventions

- ## Must begin with a letter
- ## Can 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®

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 can 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.
- Name the same entity consistently in different tables. For example, the department number column is called DEPTNO in both the EMP table and the DEPT table.

Note: Names are case insensitive. For example, EMP is treated as the same name as eMP or eMp.

For more information, see *Oracle Server SQL Reference,* Release 8, "Object Names and Qualifiers."

The CREATE TABLE Statement

- **You must have :**
  - **– CREATE TABLE privilege**
  - **– A storage area**

```
CREATE [GLOBAL TEMPORARY] TABLE [schema.]table
            (column datatype [DEFAULT expr][, ...]);
```

- **You specify:**
  - **– Table name**
  - **– Column name, column datatype, and column size**

ORACLE®

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, which are covered in subsequent lessons. DDL statements are a subset of SQL statements used to create, modify, or remove Oracle8 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:

GLOBAL TEMPORARY	specifies that the table is temporary and that its definition is visible to all sessions. The data in a temporary table is visible only to the session that inserts the data into the table.
schema	is the same as the owner's name
table	is the name of the table
DEFAULT *expr*	specifies a default value if a value is omitted in the INSERT statement
column	is the name of the column
datatype	is the column's datatype and length

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 the table.**

ORACLE®

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.

The DEFAULT Option

- **Specify a default value for a column during an insert.**

```
… hiredate DATE DEFAULT SYSDATE, …
```

- **Legal values are literal value, expression, or SQL function.**
- **Illegal values are another column's name or pseudocolumn.**
- **The default datatype must match the column datatype.**

   ORACLE®

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 datatype of the column.

Creating Tables

• **Create the table.**

```
SQL> CREATE TABLE dept
  2        (deptno  NUMBER(2),
  3         dname   VARCHAR2(14),
  4         loc     VARCHAR2(13));
Table created.
```

• **Confirm table creation.**

```
SQL> DESCRIBE dept
```

```
Name                           Null?    Type
------------------------------ -------- ---------
DEPTNO                                  NUMBER(2)
DNAME                                   VARCHAR2(14)
LOC                                     VARCHAR2(13)
```

Creating Tables

The example on the slide creates the DEPT table, with three columns—namely, 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.

Tables in the Oracle Database

- **User Tables**
  - **Collection of tables created and maintained by the user**
  - **Contain user information**
- **Data Dictionary**
  - **Collection of tables created and maintained by the Oracle server**
  - **Contain database information**

Tables in the Oracle Database

User tables are tables created by the user, such as EMP. 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 include 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 which reflects their 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. These views can be accessed only by people who have been assigned the role DBA.
V$_	These views contain information about dynamic performance views, database server performance and locking.

Querying the Data Dictionary

- **Describe tables owned by the user.**

```
SQL> SELECT   *
  2  FROM     user_tables;
```

- **View distinct object types owned by the user.**

```
SQL> SELECT   DISTINCT object_type
  2  FROM     user_objects;
```

- **View tables, views, synonyms, and sequences owned by the user.**

```
SQL> SELECT   *
  2  FROM     user_catalog;
```

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.

```
SQL> SELECT *
  2  FROM  CAT;
```

Datatypes

Datatype	Description
VARCHAR2(*size*)	Variable-length character data
CHAR(*size*)	Fixed-length character data
NUMBER(*p,s)*	Variable-length numeric data
DATE	Date and time values
LONG	Variable-length character data up to 2 gigabytes
CLOB	Single-byte character data up to 4 gigabytes
RAW and LONG RAW	Raw binary data
BLOB	Binary data up to 4 gigabytes
BFILE	Binary data stored in an external file; up to 4 gigabytes

ORACLE®

Datatypes

Data type	Description
VARCHAR2(*size*)	Variable-length character data (A maximum *size* must be specified. Default and minimum *size* is 1; maximum *size* is 4000.)
CHAR(*size*)	Fixed-length character data of length *size* bytes (Default and minimum *size* is 1; maximum *size* is 2000.)
NUMBER(*p,s*)	Number having precision *p* and scale *s* (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.)
DATE	Date and time values between January 1, 4712 B.C. and December 31, 9999 A.D.
LONG	Variable-length character data up to 2 gigabytes
CLOB	Single-byte character data up to 4 gigabytes

Datatypes

Datatype	Description
VARCHAR2(*size*)	Variable-length character data
CHAR(*size*)	Fixed-length character data
NUMBER(*p,s*)	Variable-length numeric data
DATE	Date and time values
LONG	Variable-length character data up to 2 gigabytes
CLOB	Single-byte character data up to 4 gigabytes
RAW and LONG RAW	Raw binary data
BLOB	Binary data up to 4 gigabytes
BFILE	Binary data stored in an external file; up to 4 gigabytes

ORACLE®

Datatypes (continued)

Data type	Description
RAW(*size*)	Raw binary data of length *size* (A maximum size must be specified. Maximum *size* 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

Creating a Table
by Using a Subquery

- **Create a table and insert rows by combining the CREATE TABLE statement and 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®

Creating a Table from Rows in Another Table

A second method to create a table is to apply the AS *subquery* clause to both create the table and insert rows returned from the subquery.

In the syntax:

table	is the name of the table.
column	is the name of the column, default value, and integrity constraint.
subquery	is the SELECT statement that defines the set of rows to be inserted into the new table.

Guidelines

- The table will be created with the specified column names, and the rows retrieved by the SELECT statement will be 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.

Creating a Table
by Using a Subquery

```
SQL> CREATE TABLE    dept30
  2  AS
  3     SELECT     empno, ename, sal*12 ANNSAL, hiredate
  4     FROM       emp
  5     WHERE      deptno = 30;
Table created.
```

```
SQL> DESCRIBE dept30
```

```
Name                              Null?      Type
------------------------------- --------   -----
EMPNO                           NOT NULL   NUMBER(4)
ENAME                                      VARCHAR2(10)
ANNSAL                                     NUMBER
HIREDATE                                   DATE
```

ORACLE®

Creating a Table from Rows in Another Table (continued)

The slide example creates a table, DEPT30, that contains details of all the employees working in department 30. Notice that the data for the DEPT30 table is coming from the EMP table.

You can verify the existence of a database table and check column definitions by using the SQL*Plus DESCRIBE command.

Give a column alias when selecting an expression.

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**

```
ALTER TABLE table
ADD         (column datatype [DEFAULT expr]
            [, column datatype]...);
```

```
ALTER TABLE table
MODIFY      (column datatype [DEFAULT expr]
            [, column datatype]...);
```

ALTER TABLE Statement

After you create your tables, you may need to change the table structures because you omitted a column or your column definition needs to be changed. You can do this by using the ALTER TABLE statement.

You can add columns to a table by using the ALTER TABLE statement with the ADD clause.

In the syntax:

table	is the name of the table
column	is the name of the new column
datatype	is the datatype and length of the new column
DEFAULT *expr*	specifies the default value for a new column

You can modify existing columns in a table by using the ALTER TABLE statement with the MODIFY clause.

Note: The slide gives the abridged syntax for ALTER TABLE. More about ALTER TABLE is covered in a subsequent lesson.

Adding a Column

DEPT30   **New column**

EMPNO	ENAME	ANNSAL	HIREDATE	JOB
7698	BLAKE	34200	01-MAY-81	
7654	MARTIN	15000	28-SEP-81	
7499	ALLEN	19200	20-FEB-81	
7844	TURNER	18000	08-SEP-81	
. . .				

"...add a new column into DEPT30 table..."

DEPT30

EMPNO	ENAME	ANNSAL	HIREDATE	JOB
7698	BLAKE	34200	01-MAY-81	
7654	MARTIN	15000	28-SEP-81	
7499	ALLEN	19200	20-FEB-81	
7844	TURNER	18000	08-SEP-81	
. . .				

ORACLE®

Adding a Column

The graphic adds the JOB column to DEPT30 table. Notice that the new column becomes the last column in the table.

Adding a Column

- ## You use the ADD clause to add columns.

```
SQL> ALTER TABLE dept30
  2  ADD        (job VARCHAR2(9));
Table altered.
```

- ## The new column becomes the last column.

```
    EMPNO ENAME         ANNSAL HIREDATE  JOB
--------- ---------- --------- --------- ----
     7698 BLAKE          34200 01-MAY-81
     7654 MARTIN         15000 28-SEP-81
     7499 ALLEN          19200 20-FEB-81
     7844 TURNER         18000 08-SEP-81
...
6 rows selected.
```

Guidelines for Adding a Column

- You can add or modify columns, but you cannot drop them from a table.

- 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 to the DEPT30 table. The JOB 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 datatype, size, and default value.**

```
SQL> ALTER TABLE      dept30
  2  MODIFY           (ename VARCHAR2(15));
Table altered.
```

- **A change to the default value affects only subsequent insertions to the table.**

ORACLE®

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 datatype, size, and default value.

Guidelines

- Increase the width or precision of a numeric column.

- Decrease the width of a column if the column contains only null values and if the table has no rows.

- Change the datatype if the column contains null values.

- Convert a CHAR column to the VARCHAR2 datatype or convert a VARCHAR2 column to the CHAR datatype 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

You use the DROP COLUMN clause drop columns you no longer need from the table.

```
SQL> ALTER TABLE    dept30
  2  DROP  COLUMN    job ;
Table altered.
```

ORACLE®

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 from Oracle8*i* onward.

Guidelines

- The column may or may not contain data.
- 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.

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®

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 less. This is a feature available in Oracle8i. 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 it would be 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.

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.

```
SQL> ALTER TABLE  dept30
  2  SET   UNUSED (ename);
Table altered.

SQL> ALTER TABLE  dept30
  2  DROP  UNUSED COLUMNS;
Table altered.
```

Introduction to Oracle: SQL and PL/SQL 10-20

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 this statement.**

```
SQL> DROP TABLE dept30;
Table dropped.
```

ORACLE®

Dropping a Table

The DROP TABLE statement removes the definition of an Oracle8 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;
```

where: *table*   is the name of the table

Guidelines

- All data is deleted from the table.
- Any views and synonyms will 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.

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. All DDL statements issue a commit, therefore making the transaction permanent.

Changing the Name of an Object

- **To change the name of a table, view, sequence, or synonym, you execute the RENAME statement.**

```
SQL> RENAME dept TO department;
Table renamed.
```

- **You must be the owner of the object.**

 ORACLE®

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;
```

where: *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**

```
SQL> TRUNCATE TABLE department;
Table truncated.
```

- **You cannot roll back row removal when using TRUNCATE.**
- **Alternatively, you can remove rows by using the DELETE statement.**

 ORACLE®

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 rollback row removal.

Syntax

```
TRUNCATE  TABLE    table;
```

where: *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.

Adding Comments to a Table

- **You can add comments to a table or column by using the COMMENT statement.**

```
SQL> COMMENT ON TABLE emp
  2  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®

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';
```

where:	*table*	is the name of the table
column	is the name of the column in a table
text	is the text of the comment

You can drop a comment from the database by setting it to empty string (' ').

```
SQL> COMMENT ON TABLE  emp IS ' ';
```

Summary

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®

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 datatypes, 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 Overview

- **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®

Practice 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.

Practice 10

1. Create the DEPARTMENT table based on the following table instance chart. Enter the syntax in a script called `p10q1.sql`, then execute the script to create the table. Confirm that the table is created.

Column Name	Id	Name
Key Type		
Nulls/Unique		
FK Table		
FK Column		
Datatype	Number	Varchar2
Length	7	25

```
Name            Null?     Type
----------  --------  -----------

ID                        NUMBER(7)
NAME                      VARCHAR2(25)
```

2. Populate the DEPARTMENT table with data from the DEPT table. Include only columns that you need.

3. Create the EMPLOYEE table based on the following table instance chart. Enter the syntax in a script called `p10q3.sql`, and then execute 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				
Datatype	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 10 (continued)

4. Modify the EMPLOYEE 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 DEPARTMENT and EMPLOYEE tables are stored in the data dictionary. (*Hint:* USER_TABLES)

```
TABLE_NAME
------------------------
DEPARTMENT
EMPLOYEE
```

6. Create the EMPLOYEE2 table based on the structure of the EMP table. Include only the EMPNO, ENAME, and DEPTNO columns. Name the columns in your new table ID, LAST_NAME, and DEPT_ID, respectively.

7. Drop the EMPLOYEE table.

8. Rename the EMPLOYEE2 table to EMPLOYEE.

9. Add a comment to the DEPARTMENT and EMPLOYEE table definitions describing the tables. Confirm your additions in the data dictionary.

10. Drop the LAST_NAME column from the EMPLOYEE table. Confirm your modification by checking the description of the table.

11. Create the EMPLOYEE2 table based on the structure of the EMP table. Include only the EMPNO, ENAME, and DEPTNO columns. Name the columns in your new table ID, LAST_NAME, and DEPT_ID, respectively. Mark the DEPT_ID column in the EMPLOYEE2 table as UNUSED. Confirm your modification by checking the description of the table.

12. Drop all the UNUSED columns from the EMPLOYEE2 table. Confirm your modification by checking the description of the table.

11

Including Constraints

ORACLE®

Objectives

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

- **Describe constraints**
- **Create and maintain constraints**

ORACLE®

Lesson Aim

In this lesson, you will 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 in Oracle:**
  - **NOT NULL**
  - **UNIQUE**
  - **PRIMARY KEY**
  - **FOREIGN KEY**
  - **CHECK**

Constraints

The Oracle Server uses *constraints* to prevent invalid data entry into tables.

You can use constraints to do the following:

- Enforce rules at the table level 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 this column may not 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 *Oracle Server SQL Reference,* Release 8, "CONSTRAINT Clause."

Constraint Guidelines

- **Name a constraint or the Oracle Server will generate a name by using the SYS_C*n* format.**
- **Create a constraint:**
  - **At the same time as the table is created**
  - **After the table has been created**
- **Define a constraint at the column or table level.**
- **View a constraint in the data dictionary.**

 ORACLE®

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, Oracle generates a name with the format SYS_C*n*, where *n* is an integer to create a unique constraint name.

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 emp(
            empno   NUMBER(4),
            ename   VARCHAR2(10),
            ...
            deptno  NUMBER(2) NOT NULL,
            CONSTRAINT emp_empno_pk
                        PRIMARY KEY (EMPNO));
```

   ORACLE®

Defining Constraints

The slide gives the syntax for defining constraints while creating a table.

In the syntax:

schema	is the same as the owner's name
table	is the name of the table
DEFAULT *expr*	specifies a default value if a value is omitted in the INSERT statement
column	is the name of the column
datatype	is the column's datatype and length
column_constraint	is an integrity constraint as part of the column definition
table_constraint	is an integrity constraint as part of the table definition

For more information, see *Oracle Server SQL Reference,* Release 8, "CREATE TABLE."

Defining Constraints

- ## Column constraint level

```
column [CONSTRAINT constraint_name] constraint_type,
```

- ## Table constraint level

```
column,...
   [CONSTRAINT constraint_name] constraint_type
   (column, ...),
```

     ORACLE®

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:

constraint_name	is the name of the constraint
constraint_type	is the type of the constraint

The NOT NULL Constraint

Ensures that null values are not permitted for the column

EMP

EMPNO	ENAME	JOB	. . .	COMM	DEPTNO
7839	KING	PRESIDENT			10
7698	BLAKE	MANAGER			30
7782	CLARK	MANAGER			10
7566	JONES	MANAGER			20
. . .					

NOT NULL constraint (no row can contain a null value for this column)   **Absence of NOT NULL constraint (any row can contain null for this column)**   **NOT NULL constraint**

ORACLE®

The NOT NULL Constraint

The NOT NULL constraint ensures that null values are not allowed in the column. Columns without the NOT NULL constraint can contain null values by default.

The NOT NULL Constraint

Defined at the column level

```
SQL> CREATE TABLE emp(
  2       empno     NUMBER(4),
  3       ename     VARCHAR2(10) NOT NULL,
  4       job       VARCHAR2(9),
  5       mgr       NUMBER(4),
  6       hiredate  DATE,
  7       sal       NUMBER(7,2),
  8       comm      NUMBER(7,2),
  9       deptno    NUMBER(7,2) NOT NULL);
```

ORACLE®

The NOT NULL Constraint (continued)

The NOT NULL constraint can be specified only at the column level, not at the table level.
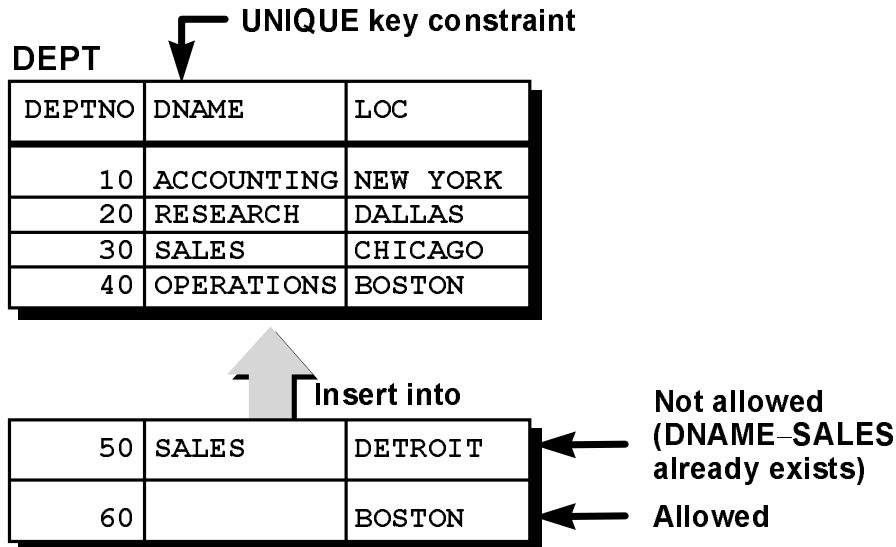
The slide example applies the NOT NULL constraint to the ENAME and DEPTNO columns of the EMP table. Because these constraints are unnamed, the Oracle Server will create names for them.

You can specify the name of the constraint while specifying the constraint.

```
... deptno NUMBER(7,2)

    CONSTRAINT emp_deptno_nn NOT NULL...
```

Note: All 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 KEY Constraint

UNIQUE key constraint

DEPT

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

Insert into

| 50 | SALES | DETROIT | ← Not allowed (DNAME–SALES already exists) |
| 60 | | BOSTON | ← Allowed |

ORACLE®

The UNIQUE Key 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 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 key comprises more than one column, that group of columns is said to be a *composite unique key*.

UNIQUE key 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 key 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.

The UNIQUE KEY Constraint

Defined at either the table level or the column level

```
SQL> CREATE TABLE    dept(
  2        deptno     NUMBER(2),
  3        dname      VARCHAR2(14),
  4        loc        VARCHAR2(13),
  5        CONSTRAINT dept_dname_uk UNIQUE(dname));
```

ORACLE®
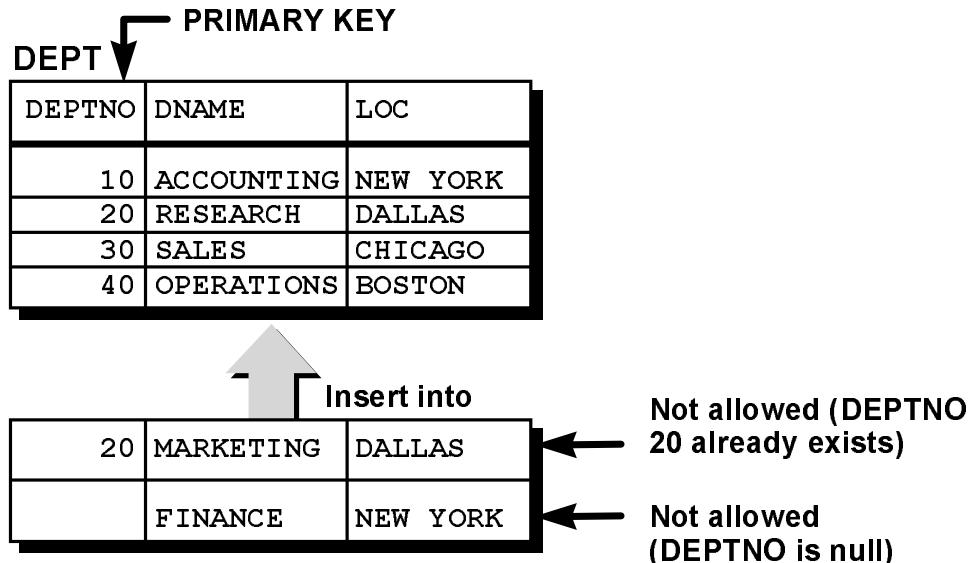
The UNIQUE Key Constraint (continued)

UNIQUE key 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 UNIQUE key constraint to the DNAME column of the DEPT table. The name of the constraint is DEPT_DNAME_UK.

Note: The Oracle Server enforces the UNIQUE key constraint by implicitly creating a unique index on the unique key.

The PRIMARY KEY Constraint

PRIMARY KEY

DEPT

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

Insert into

20	MARKETING	DALLAS		**Not allowed (DEPTNO 20 already exists)**
	FINANCE	NEW YORK		**Not allowed (DEPTNO is null)**

ORACLE®

The PRIMARY KEY Constraint

A PRIMARY KEY constraint creates a primary key for the table. Only one primary key can be created for a 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

```
SQL> CREATE TABLE    dept(
  2        deptno       NUMBER(2),
  3        dname        VARCHAR2(14),
  4        loc          VARCHAR2(13),
  5        CONSTRAINT dept_dname_uk UNIQUE  (dname),
  6        CONSTRAINT dept_deptno_pk PRIMARY KEY(deptno));
```

     ORACLE®
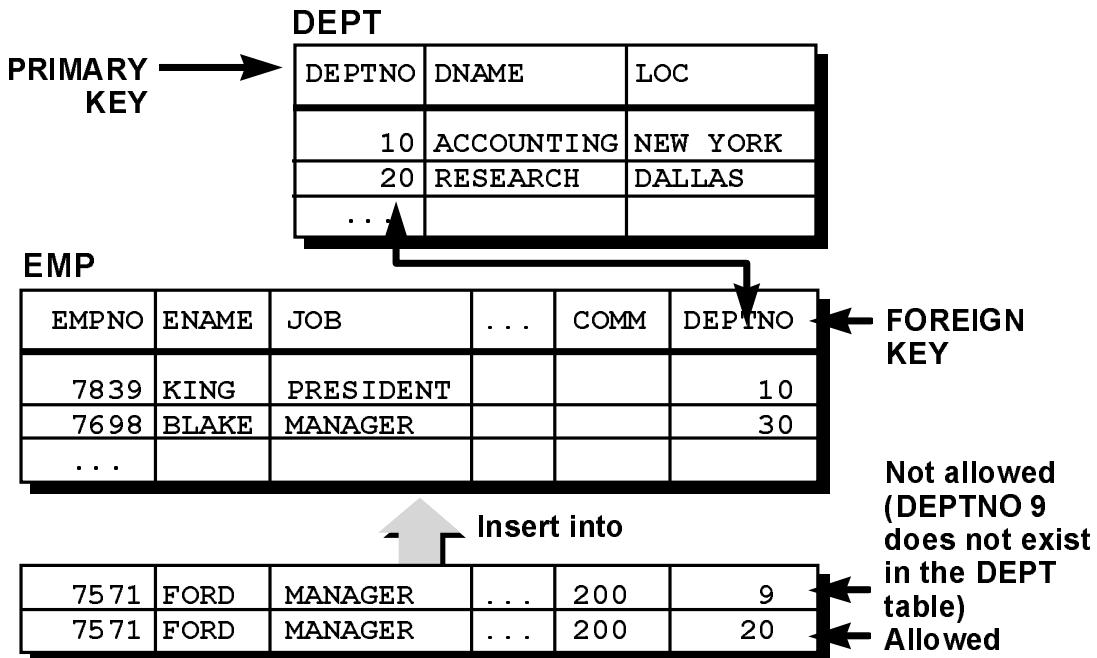
**The PRIMARY KEY Constraint (continued)**

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.

The example on the slide defines a PRIMARY KEY constraint on the DEPTNO column of the DEPT table. The name of the constraint is DEPT_DEPTNO_PK.

Note: A UNIQUE index is automatically created for a PRIMARY KEY column.

The FOREIGN KEY Constraint

DEPT

PRIMARY KEY →

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
..		

EMP

EMPNO	ENAME	JOB	. . .	COMM	DEPTNO
7839	KING	PRESIDENT			10
7698	BLAKE	MANAGER			30
...					

← **FOREIGN KEY**

Insert into

| 7571 | FORD | MANAGER | . . . | 200 | 9 |
| 7571 | FORD | MANAGER | . . . | 200 | 20 |

← **Not allowed (DEPTNO 9 does not exist in the DEPT table)**

← **Allowed**

ORACLE®

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, DEPTNO has been defined as the foreign key in the EMP table (dependent or child table); it references the DEPTNO column of the DEPT table (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.

The FOREIGN KEY Constraint

Defined at either the table level or the column level

```
SQL> CREATE TABLE emp(
  2      empno     NUMBER(4),
  3      ename     VARCHAR2(10) NOT NULL,
  4      job       VARCHAR2(9),
  5      mgr       NUMBER(4),
  6      hiredate  DATE,
  7      sal       NUMBER(7,2),
  8      comm      NUMBER(7,2),
  9      deptno    NUMBER(7,2) NOT NULL,
 10      CONSTRAINT emp_deptno_fk FOREIGN KEY (deptno)
 11                 REFERENCES dept (deptno));
```

ORACLE®

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 DEPTNO column of the EMP table, using table level syntax. The name of the constraint is EMP_DEPTNO_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:

```
SQL> CREATE TABLE emp
     (... .
     deptno NUMBER(2) CONSTRAINT emp_deptno_fk REFERENCES
                                         dept(deptno),
     ... .
     );
```

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: Allows deletion in the parent table and deletion of the dependent rows in the child table**

ORACLE®

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.

Without the ON DELETE CASCADE option, 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**
- **Expressions that 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**

```
..., deptno   NUMBER(2),
        CONSTRAINT emp_deptno_ck
             CHECK (DEPTNO BETWEEN 10 AND 99),...
```

ORACLE®

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 that reference the column in its definition. There is no limit to the number of CHECK constraints that you can define on a column.

CHECK constraints can be defined at the column level or table level.

Adding a Constraint

```
ALTER TABLE  table
ADD [CONSTRAINT  constraint]  type  (column);
```

- **Add or drop, but not modify, a constraint**
- **Enable or disable constraints**
- **Add a NOT NULL constraint by using the MODIFY clause**

ORACLE®

Adding a Constraint

You can add a constraint for existing tables by using the ALTER TABLE statement with the ADD clause.

In the syntax:

table	is the name of the table
constraint	is the name of the constraint
type	is the constraint type
column	is the name of the column affected by the constraint

The constraint name syntax is optional, although recommended. If you do not name your constraints, the system 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 contains no rows, because data cannot be specified for existing rows at the same time that the column is added.

Adding a Constraint

Add a FOREIGN KEY constraint to the EMP table indicating that a manager must already exist as a valid employee in the EMP table.

```
SQL> ALTER TABLE       emp
  2  ADD CONSTRAINT   emp_mgr_fk
  3                   FOREIGN KEY(mgr) REFERENCES emp(empno);
Table altered.
```

ORACLE®

Adding a Constraint (continued)

The example on the slide creates a FOREIGN KEY constraint on the EMP table. The constraint ensures that a manager exists as a valid employee in the EMP table.

Dropping a Constraint

- **Remove the manager constraint from the EMP table.**

```
SQL> ALTER TABLE       emp
  2   DROP CONSTRAINT   emp_mgr_fk;
Table altered.
```

- **Remove the PRIMARY KEY constraint on the DEPT table and drop the associated FOREIGN KEY constraint on the EMP.DEPTNO column.**

```
SQL> ALTER TABLE      dept
  2   DROP PRIMARY KEY CASCADE;
Table altered.
```

ORACLE®

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];
```

where: *table*   is the name of the table

 column   is the name of the column affected by the constraint

 constraint   is the name of the constraint

When you drop an integrity constraint, that constraint is no longer enforced by the Oracle Server and is no longer available in the data dictionary.

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.**

```
SQL> ALTER TABLE          emp
  2  DISABLE CONSTRAINT    emp_empno_pk CASCADE;
Table altered.
```

ORACLE®

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];
```

where: *table*   is the name of the table

           constraint   is the name of the constraint

Guidelines

- You can use the DISABLE clause in both the CREATE TABLE statement and the ALTER TABLE statement.
- The CASCADE clause disables dependent integrity constraints.

Enabling Constraints

- **Activate an integrity constraint currently disabled in the table definition by using the ENABLE clause.**

```
SQL> ALTER TABLE          emp
  2   ENABLE CONSTRAINT    emp_empno_pk;
Table altered.
```

- **A UNIQUE or PRIMARY KEY index is automatically created if you enable a UNIQUE key or PRIMARY KEY constraint.**

ORACLE®

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;
```

where: *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 automatically created.
- You can use the ENABLE clause in both the CREATE TABLE statement and the ALTER TABLE statement.

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.**

 ORACLE®

Cascading CONSTRAINTS

This statement illustrates the usage of the CASCADE CONSTRAINTS clause. Assume table test1 is created as follows:

```
SQL> CREATE TABLE test1 (
  2   pk NUMBER PRIMARY KEY,
  3   fk NUMBER,
  4   col1 NUMBER,
  5   col2 NUMBER,
  6   CONSTRAINT fk_constraint FOREIGN KEY (fk) REFERENCES test1,
  7   CONSTRAINT ck1 CHECK (pk > 0 and col1 > 0),
  8   CONSTRAINT ck2 CHECK (col2 > 0));
```

An error will be returned for the following statements:

```
SQL> ALTER TABLE test1 DROP (pk); -- pk is a parent key
SQL> ALTER TABLE test1 DROP (col1); -- c1 is referenced by multicolumn constraint ck1
```

Cascading Constraints

The CASCADE CONSTRAINTS clause also drops all multicolumn constraints defined on the dropped columns.

   ORACLE®

Cascading CONSTRAINTS (continued)

Submitting the following statement drops column pk, the primary key constraint, the foreign key constraint, fk_constraint, and the check constraint, ck1:

```
SQL> 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, then it is valid to submit the following statement without the CASCADE CONSTRAINTS clause:

```
SQL> ALTER TABLE test1 DROP (pk, fk, col1);
```

Viewing Constraints

Query the USER_CONSTRAINTS table to view all constraint definitions and names.

```
SQL>   SELECT  constraint_name, constraint_type,
  2            search_condition
  3    FROM    user_constraints
  4    WHERE   table_name = 'EMP';
```

```
CONSTRAINT_NAME              C SEARCH_CONDITION
-----------------------      - -------------------------
SYS_C00674                   C EMPNO IS NOT NULL
SYS_C00675                   C DEPTNO IS NOT NULL
EMP_EMPNO_PK                 P
...
```

ORACLE®

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 all the constraints on the EMP 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.

Viewing the Columns
Associated with Constraints

View the columns associated with the constraint names in the USER_CONS_COLUMNS view.

```
SQL> SELECT    constraint_name, column_name
  2  FROM      user_cons_columns
  3  WHERE     table_name = 'EMP';
```

```
CONSTRAINT_NAME                  COLUMN_NAME
------------------------------   ----------------------
EMP_DEPTNO_FK                    DEPTNO
EMP_EMPNO_PK                     EMPNO
EMP_MGR_FK                       MGR
SYS_C00674                       EMPNO
SYS_C00675                       DEPTNO
```

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 the system-assigned name.

Summary

- **Create the following types of constraints:**
  - **NOT NULL**
  - **UNIQUE**
  - **PRIMARY KEY**
  - **FOREIGN KEY**
  - **CHECK**
- **Query the USER_CONSTRAINTS table to view all constraint definitions and names.**

ORACLE®

Summary

The Oracle Server uses constraints to prevent invalid data entry into tables.

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 Overview

- **Adding constraints to existing tables**
- **Adding more columns to a table**
- **Displaying information in data dictionary views**

Copyright © Oracle Corporation, 1999. All rights reserved.   ORACLE®

Practice 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 to name the constraints that you will be defining during the practices.

Practice 11

1. Add a table-level PRIMARY KEY constraint to the EMPLOYEE table using the ID column. The constraint should be named at creation.

Hint: The constraint is enabled as soon as the ALTER TABLE command executes successfully.

2. Create a PRIMARY KEY constraint on the DEPARTMENT table using the ID column. The constraint should be named at creation.

Hint: The constraint is enabled as soon as the ALTER TABLE command executes successfully.

3. Add a foreign key reference on the EMPLOYEE table that will ensure that the employee is not assigned to a nonexistent department.

4. Confirm that the constraints were added by querying USER_CONSTRAINTS. Note the types and names of the constraints. Save your statement text in a file called `p11q4.sql`.

```
CONSTRAINT_NAME          C
------------------------ --
DEPARTMENT_ID_PK         P
EMPLOYEE_ID_PK           P
EMPLOYEE_DEPT_ID_FK      R
```

5. Display the object names and types from the USER_OBJECTS data dictionary view for EMPLOYEE and DEPARTMENT tables. You may want to format the columns for readability. Notice that the new tables and a new index were created.

```
OBJECT_NAME        OBJECT_TYPE
----------------   -----------
DEPARTMENT         TABLE
DEPARTMENT_ID_PK   INDEX
EMPLOYEE           TABLE
EMPLOYEE_ID_PK     INDEX
```

If you have time, complete the following exercise:

6. Modify the EMPLOYEE table. Add a SALARY column of NUMBER datatype, precision 7.

12

Creating Views

ORACLE®

Objectives

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

- **Describe a view**

- **Create a view**

- **Retrieve data through a view**

- **Alter the definition of a view**

- **Insert, update, and delete data through a view**

- **Drop a view**

Lesson Aim

In this lesson, you will learn to create and use views. You will also learn to query the relevant data dictionary object to retrieve information about views.

Objectives

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

- **Describe an inline view**
- **Perform "Top-N" Analysis**

ORACLE®

Lesson Aim

In this lesson, you will also 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?

EMP Table

```
EMPNO ENAME     JOB       MGR HIREDATE    SAL  COMM DEPTNO
----- -------- --------- ---- --------- ------ ----- -------
```

EMPVU10 View

EMPNO	ENAME	JOB
7839	KING	PRESIDENT
7782	CLARK	MANAGER
7934	MILLER	CLERK

```
20
20
20
20
30
                                    1400    30
                             300            30
                               0            30
7900  JAMES   CLERK    7698 03-DEC-81   950       30
7521  WARD    SALESMAN 7698 22-FEB-81  1250  500  30
```

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.

Why Use Views?

- **To restrict data access**
- **To make complex queries easy**
- **To allow data independence**
- **To present different views of the same data**

 ORACLE®

Advantages of Views

- Views restrict access to the data because the view can display selective columns from the table.
- Views allow users to make simple queries to retrieve the results from complicated queries. For example, views allow users to query information from multiple tables without 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 *Oracle Server SQL Reference*, Release 8, "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 through view	Yes	Not always

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 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 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];
```

- **The subquery can contain complex SELECT syntax.**

- **The subquery cannot contain an ORDER BY clause.**

Creating a View

You can create a view by embedding a subquery within the CREATE VIEW statement.

In the syntax:

OR REPLACE	re-creates the view if it already exists
FORCE	creates the view regardless of whether or not the base tables exist
NOFORCE	creates the view only if the base tables exist (This is the default.)
view	is the name of the view
alias	specifies names for the expressions selected by the view's query (The number of aliases must match the number of expressions selected by the view.)
subquery	is a complete SELECT statement (You can use aliases for the columns in the SELECT list.)
WITH CHECK OPTION	specifies that only rows accessible to the view can be inserted or updated
constraint	is the name assigned to the CHECK OPTION constraint
WITH READ ONLY	ensures that no DML operations can be performed on this view

Creating a View

- **Create a view, EMPVU10, that contains details of employees in department 10.**

```
SQL> CREATE VIEW     empvu10
  2  AS SELECT        empno, ename, job
  3  FROM            emp
  4  WHERE           deptno = 10;
View created.
```

- **Describe the structure of the view by using the SQL*Plus DESCRIBE command.**

```
SQL> DESCRIBE empvu10
```

ORACLE®

Creating a View (continued)

The example on the slide creates a view that contains the employee number, name, and job title for all the employees in department 10.

You can display the structure of the view by using the SQL*Plus DESCRIBE command.

```
Name                            Null?    Type
------------------------------- -------- ------------
EMPNO                           NOT NULL NUMBER(4)
ENAME                                    VARCHAR2(10)
JOB                                      VARCHAR2(9)
```

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 CHECK OPTION, the system will assign 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.**

```
SQL> CREATE VIEW      salvu30
  2  AS SELECT       empno EMPLOYEE_NUMBER, ename NAME,
  3                  sal SALARY
  4  FROM            emp
  5  WHERE           deptno = 30;
View created.
```

- **Select the columns from this view by the given alias names.**

 ORACLE®

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 (empno) with the alias EMPLOYEE_NUMBER, name (ename) with the alias NAME, and salary (sal) with the alias SALARY for department 30.

Alternatively, you can control the column names by including column aliases in the CREATE VIEW clause.

Retrieving Data from a View

```
SQL>  SELECT *
  2   FROM     salvu30;

EMPLOYEE_NUMBER NAME            SALARY
--------------- ---------- ----------
           7698 BLAKE            2850
           7654 MARTIN           1250
           7499 ALLEN            1600
           7844 TURNER           1500
           7900 JAMES             950
           7521 WARD             1250

6 rows selected.
```
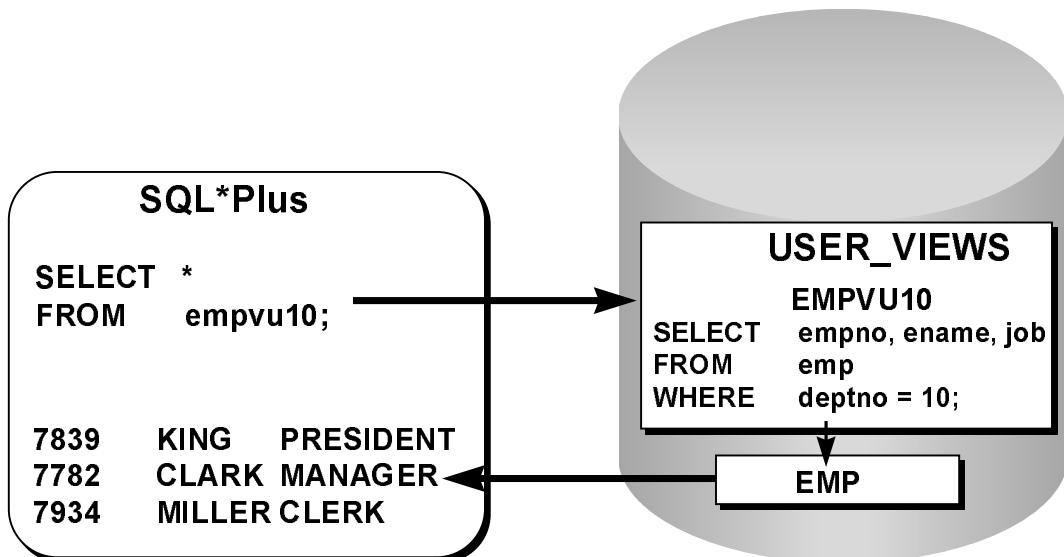
  ORACLE®

Retrieving Data from a View

You can retrieve data from a view as you would from any table. You can either display the contents of the entire view or just view specific rows and columns.

Querying a View

ORACLE®

Views in the Data Dictionary

Once your view has been created, you can query the data dictionary table 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. Retrieves the view definition from the data dictionary table USER_VIEWS.
2. Checks access privileges for the view base table.
3. 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 made to, the base table(s).

Modifying a View

- **Modify the EMPVU10 view by using CREATE OR REPLACE VIEW clause. Add an alias for each column name.**

```
SQL> CREATE OR REPLACE VIEW empvu10
  2          (employee_number, employee_name, job_title)
  3  AS SELECT      empno, ename, job
  4  FROM           emp
  5  WHERE          deptno = 10;
View created.
```

- **Column aliases in the CREATE VIEW clause are listed in the same order as the columns in the subquery.**

ORACLE®

Modifying a View

The OR REPLACE option allows a view to 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.

Creating a Complex View

Create a complex view that contains group functions to display values from two tables.

```
SQL> CREATE VIEW      dept_sum_vu
  2                   (name, minsal, maxsal, avgsal)
  3  AS SELECT        d.dname, MIN(e.sal), MAX(e.sal),
  4                   AVG(e.sal)
  5  FROM             emp e, dept d
  6  WHERE            e.deptno = d.deptno
  7  GROUP BY         d.dname;
View created.
```

ORACLE®

Creating a Complex View

The example on the slide creates a complex view of the department names, minimum salary, maximum salary, and average salary by the 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 SQL*Plus DESCRIBE command. Display the contents of the view by issuing a SELECT statement.

```
SQL> SELECT   *
  2  FROM     dept_sum_vu;
```

NAME	MINSAL	MAXSAL	AVGSAL
ACCOUNTING	1300	5000	2916.6667
RESEARCH	800	3000	2175
SALES	950	2850	1566.6667

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**

Copyright © Oracle Corporation, 1999. All rights reserved.   ORACLE®

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 pseudocolumn ROWNUM keyword

Rules for Performing
DML Operations on a View

- **You cannot modify data in a view if it contains:**
    - **Any of the conditions mentioned in the previous slide**
    - **Columns defined by expressions**
    - **The ROWNUM pseudocolumn**
- **You cannot add data if:**
    - **The view contains any of the conditions mentioned above or in the previous slide**
    - **There are NOT NULL columns in the base tables that are not selected by the view**

Performing DML Operations on a View (continued)

You can modify data in a view unless it contains any of the conditions mentioned in the previous slide and any of the following:

- Columns defined by expressions—for example, SAL * 12
- The ROWNUM pseudocolumn

You can add data through a view unless it contains any of the above and there are NOT NULL columns, without a default value, 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 *0racle8 Server SQL Reference,* Release 8, "CREATE VIEW."

Using the WITH CHECK OPTION Clause

- **You can ensure that DML on the view stays within the domain of the view by using the WITH CHECK OPTION clause.**

```
SQL> CREATE OR REPLACE VIEW empvu20
  2  AS SELECT      *
  3  FROM          emp
  4  WHERE         deptno = 20
  5  WITH CHECK OPTION CONSTRAINT empvu20_ck;
View created.
```

- **Any attempt to change the department number for any row in the view will fail because it violates the WITH CHECK OPTION constraint.**

 ORACLE®

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 are not allowed to create rows that 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.

```
SQL> UPDATE empvu20
  2  SET    deptno = 10
  3  WHERE  empno = 7788;
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 department 20 employees 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.**

```
SQL> CREATE OR REPLACE VIEW empvu10
  2          (employee_number, employee_name, job_title)
  3  AS SELECT     empno, ename, job
  4  FROM          emp
  5  WHERE         deptno = 10
  6  WITH READ ONLY;
View created.
```

- **Any attempt to perform a DML on any row in the view will result in Oracle Server error.**

ORACLE®

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.

Any attempts to remove a row from the view will result in an error.

```
SQL> DELETE FROM  empvu10
  2  WHERE  employee_number = 7782;
DELETE FROM empvu10
            *
ERROR at line 1:
ORA-01752:Cannot delete from view without exactly one key-preserved
table
```

Any attempts to insert a row or modify a row using the view will result in Oracle Server error –
01733: virtual column not allowed here.

Removing a View

Remove a view without losing data because a view is based on underlying tables in the database.

```
DROP VIEW view;
```

```
SQL> DROP VIEW empvu10;
View dropped.
```

ORACLE®

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 (correlation name) that you can use within a SQL statement.**

- **An inline view is similar to using a named subquery in the FROM clause of the main query.**

- **An inline view is not a schema object.**

ORACLE®

Inline Views

An inline view in the FROM clause of a SELECT statement defines a data source for the SELECT statement. In the example below the inline view b returns the details of all department numbers and the maximum salary for each department from the EMP table. The WHERE a.deptno = b.deptno AND a.sal < 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 .

```
SQL> SELECT  a.ename, a.sal, a.deptno, b.maxsal
  2    FROM   emp a, (SELECT   deptno, max(sal) maxsal
  3                     FROM      emp
  4                     GROUP BY deptno) b
  5   WHERE   a.deptno = b.deptno
  6   AND     a.sal < b.maxsal;

    ENAME           SAL    DEPTNO    MAXSAL
    ---------- --------- --------- ---------
    CLARK          2450        10      5000
    MILLER         1300        10      5000
    ...
    TURNER         1500        30      2850
    JAMES           950        30      2850
10 rows selected.
```

Introduction to Oracle: SQL and PL/SQL 12-20

"Top-N" Analysis

- **Top-N queries ask for the n largest or smallest values of a column.**
  - **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®

"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

Performing "Top-N" Analysis

The high-level structure of a Top-N analysis query is:

```
SQL> SELECT [column_list], ROWNUM
  2 FROM   (SELECT [column_list] FROM table
  3          ORDER BY Top-N_column)
  4 WHERE  ROWNUM <=  N
```

ORACLE®

Performing "Top-N" Analysis

Top-N queries use a consistent nested query structure with the elements described below:

- 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.

- Outer query to limit the number of rows in the final result set. The outer query includes the following components:

    - The ROWNUM pseudo-column, which assigns a sequential value starting with 1 to each of the rows returned from the subquery.

    - 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 earners names and salaries from the EMP table.

```
SQL> SELECT ROWNUM as RANK, ename, sal
  2  FROM   (SELECT ename,sal FROM emp
  3             ORDER BY sal DESC)
  4  WHERE ROWNUM <= 3;
```

```
RANK            ENAME           SAL
----------      ----------      ----------
         1      KING            5000
         2      SCOTT           3000
         3      FORD            3000
```

 ORACLE®

Example of "Top-N" Analysis

The example on the slide describes how to display the names and salaries of the top three earners from the EMP table. The subquery returns the details of all employee names and salaries from the EMP 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 is 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.

```
SQL> SELECT ROWNUM as SENIOR,E.ename, E.hiredate
  2  FROM   (SELECT ename,hiredate FROM emp
  3             ORDER BY hiredate)E
  4  WHERE rownum <= 4;


SENIOR ENAME       HIREDATE
--------- ---------- ---------
        1 SMITH       17-DEC-80
        2 ALLEN       20-FEB-81
        3 WARD        22-FEB-81
        4 JONES       02-APR-81
```

Summary

- **A view is derived from data in other tables or other views.**

- **A view provides the following advantages:**

  - **Restricts database access**

  - **Simplifies queries**

  - **Provides data independence**

  - **Allows multiple views of the same data**

  - **Can be dropped without removing the underlying data**

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

- Allow 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 be replaced if one of the same name exists

- Can Contain a check constraint

- Can be read-only

Summary

- **An inline view is a subquery with an alias name.**
- **"Top-N" analysis can be done using:**
  - **Subquery**
  - **Outer query**

ORACLE®

Practice Overview

- **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**

Practice Overview

In this practice, you will create simple and complex views and attempt to perform DML statements on the views.

Practice 12

1. Create a view called EMP_VU based on the employee number, employee name, and department number from the EMP table. Change the heading for the employee name to EMPLOYEE.

2. Display the contents of the EMP_VU view.

```
EMPNO EMPLOYEE DEPTNO
----- -------- ------
 7839 KING         10
 7698 BLAKE        30
 7782 CLARK        10
 7566 JONES        20
 7654 MARTIN       30
 7499 ALLEN        30
 7844 TURNER       30
 7900 JAMES        30
 7521 WARD         30
 7902 FORD         20
 7369 SMITH        20
 7788 SCOTT        20
 7876 ADAMS        20
 7934 MILLER       10
14 rows selected.
```

3. Select the view name and text from the data dictionary USER_VIEWS.

```
VIEW_NAME   TEXT
----------- -----------------------------------------
EMP_VU      SELECT empno, ename employee, deptno
            FROM emp
```

4. Using your view EMP_VU, enter a query to display all employee names and department numbers.

```
EMPLOYEE      DEPTNO
----------  ---------
KING             10
BLAKE            30
CLARK            10
JONES            20
MARTIN           30
...
14 rows selected.
```

Practice 12 (continued)

5. Create a view named DEPT20 that contains the employee number, employee name, and department number for all employees in department 20. Label the view column EMPLOYEE_ID, EMPLOYEE, and DEPARTMENT_ID. Do not allow an employee to be reassigned to another department through the view.

6. Display the structure and contents of the DEPT20 view.

```
Name              Null?     Type
---------------   --------  ------
EMPLOYEE_ID       NOT NULL  NUMBER(4)
EMPLOYEE                    VARCHAR2(10)
DEPARTMENT_ID     NOT NULL  NUMBER(2)


EMPLOYEE_ID EMPLOYEE DEPARTMENT_ID
----------- -------- -------------
       7566 JONES               20
       7902 FORD                20
       7369 SMITH               20
       7788 SCOTT               20
       7876 ADAMS               20
```

7. Attempt to reassign Smith to department 30.

If you have time, complete the following exercise:

8. Create a view called SALARY_VU based on the employee name, department name, salary, and salary grade for all employees. Label the columns Employee, Department, Salary, and Grade, respectively.

13

Other Database Objects

ORACLE®

Objectives

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

- **Describe some database objects and their uses**
- **Create, maintain, and use sequences**
- **Create and maintain indexes**
- **Create private and public synonyms**

 ORACLE®

Lesson Aim

In this lesson, you will 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®

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?

- **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®

What Is a Sequence?

A sequence is a database object created by a user, and can be shared by multiple users to generate unique integers. You can use the sequences to automatically generate primary key values.

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 Oracle8 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

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®

Creating a Sequence

Automatically generate sequential numbers by using the CREATE SEQUENCE statement.

In the syntax:

sequence	is the name of the sequence generator
INCREMENT BY n	specifies the interval between sequence numbers where n is an integer (If this clause is omitted, the sequence will increment by 1.)
START WITH n	specifies the first sequence number to be generated (If this clause is omitted, the sequence will start with 1.)
MAXVALUE n	specifies the maximum value the sequence can generate
NOMAXVALUE	specifies a maximum value of 10^{27} for an ascending sequence and -1 for a descending sequence (This is the default option.)
MINVALUE n	specifies the minimum sequence value
NOMINVALUE	specifies a minimum value of 1 for an ascending sequence and $-(10^{26})$ for a descending sequence (This is the default option.)
CYCLE	NOCYCLE	specifies that the sequence continues to generate values after reaching either its maximum or minimum value or does not generate additional values (NOCYCLE is the default option.)
CACHE n	NOCACHE	specifies how many values the Oracle Server will preallocate and keep in memory (By default, the Oracle Server will cache 20 values.)

Creating a Sequence

- **Create a sequence named DEPT_DEPTNO to be used for the primary key of the DEPT table.**

- **Do not use the CYCLE option.**

```
SQL> CREATE SEQUENCE dept_deptno
  2         INCREMENT BY 1
  3         START WITH 91
  4         MAXVALUE 100
  5         NOCACHE
  6         NOCYCLE;
Sequence created.
```

ORACLE®

Creating a Sequence (continued)

The example on the slide creates a sequence named DEPT_DEPTNO to be used for the DEPTNO column of the DEPT table. The sequence starts at 91, does not allow caching, and does not allow the sequence to 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 *Oracle Server SQL Reference,* Release 8, "CREATE SEQUENCE."

Confirming Sequences

- **Verify your sequence values in the USER_SEQUENCES data dictionary table.**

```
SQL> SELECT   sequence_name, min_value, max_value,
  2           increment_by, last_number
  3  FROM     user_sequences;
```

- **The LAST_NUMBER column displays the next available sequence number.**

ORACLE®

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 data dictionary USER_SEQUENCES table.

SEQUENCE_NAME	MIN_VALUE	MAX_VALUE	INCREMENT_BY	LAST_NUMBER
CUSTID	1	1.000E+27	1	109
DEPT_DEPTNO	1	100	1	91
ORDID	1	1.000E+27	1	622
PRODID	1	1.000E+27	1	200381

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.

Using a Sequence

Once you create your sequence, you can use the sequence to generate 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.

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®

Rules for Using NEXTVAL and CURRVAL

You can use NEXTVAL and CURRVAL in the following:

- 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:

- A SELECT list of a view
- A SELECT statement with the DISTINCT keyword
- A SELECT statement with the GROUP BY, HAVING, or ORDER BY clauses
- A subquery in a SELECT, DELETE, or UPDATE statement
- A DEFAULT expression in a CREATE TABLE or ALTER TABLE statement

For more information, see *Oracle Server SQL Reference,* Release 8, "Pseudocolumns" section and "CREATE SEQUENCE."

Using a Sequence

- **Insert a new department named "MARKETING" in San Diego.**

```
SQL> INSERT INTO    dept(deptno, dname, loc)
  2  VALUES         (dept_deptno.NEXTVAL,
  3                 'MARKETING', 'SAN DIEGO');
1 row created.
```

- **View the current value for the DEPT_DEPTNO sequence.**

```
SQL> SELECT   dept_deptno.CURRVAL
  2  FROM     dual;
```

ORACLE®

Using a Sequence

The example on the slide inserts a new department in the DEPT table. It uses the DEPT_DEPTNO sequence for generating a new department number.

You can view the current value of the sequence:

```
SQL> SELECT   dept_deptno.CURRVAL
  2  FROM     dual;


CURRVAL
-------
     91
```

Suppose now you want to hire employees to staff the new department. The INSERT statement that can be executed repeatedly for all the new employees can include the following code:

```
SQL> INSERT INTO emp ...
  2  VALUES (emp_empno.NEXTVAL, dept_deptno.CURRVAL, ...);
```

Note: The above example assumes that a sequence EMP_EMPNO has already been created for generating a new employee number.

Using a Sequence

- **Caching sequence values in memory allows 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**
- **View the next available sequence, if it was created with NOCACHE, by querying the USER_SEQUENCES table.**

   ORACLE®

Caching Sequence Values

Cache sequences in the memory to allow faster access to those sequence values. The cache is populated at the first reference to the sequence. Each request for the next sequence value is retrieved from the cached sequence. After the last sequence is used, the next request for the sequence pulls another cache of sequences into memory.

Beware of Gaps in Your 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 this occurs, 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.

Modifying a Sequence

Change the increment value, maximum value, minimum value, cycle option, or cache option.

```
SQL> ALTER SEQUENCE dept_deptno
  2         INCREMENT BY 1
  3         MAXVALUE 999999
  4         NOCACHE
  5         NOCYCLE;
Sequence altered.
```

ORACLE®

Altering a Sequence

If you reach the MAXVALUE limit for your sequence, no additional values from the sequence will be 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}];
```

where: *sequence*   is the name of the sequence generator

For more information, see *Oracle Server SQL Reference*, Release 8, "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®

Guidelines

- You must be the owner 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 cannot be imposed that is less than the current sequence number.

```
SQL> ALTER SEQUENCE dept_deptno
  2         INCREMENT BY 1
  3         MAXVALUE 90
  4         NOCACHE
  5         NOCYCLE;
ALTER SEQUENCE dept_deptno
*
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.**

```
SQL> DROP SEQUENCE dept_deptno;
Sequence dropped.
```

ORACLE®

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;
```

where: *sequence* is the name of the sequence generator

For more information, see *Oracle Server SQL Reference*, Release 8, "DROP SEQUENCE."

What Is 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 rapid path access method to locate the data quickly**
- **Is independent of the table it indexes**
- **Is used and maintained automatically by the Oracle Server**

ORACLE®

What Is an Index?

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 will occur.

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 *Oracle Server Concepts Manual,* Release 8, "Schema Objects" section, "Indexes" topic.

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 time to the rows.**

 ORACLE®

How Are Indexes Created?

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 that a user can create is a nonunique index. For example, you can create a FOREIGN KEY column index for a join in a query to improve retrieval speed.

Creating an Index

- **Create an index on one or more columns.**

```
CREATE INDEX index
ON table (column[, column]...);
```

- **Improve the speed of query access on the ENAME column in the EMP table.**

```
SQL> CREATE INDEX    emp_ename_idx
  2  ON              emp(ename);
Index created.
```

ORACLE®

Creating an Index

Create an index on one or more columns by issuing the CREATE INDEX statement.

In the syntax:

index	is the name of the index
table	is the name of the table
column	is the name of the column in the table to be indexed

For more information, see *Oracle Server SQL Reference,* Release 8, "CREATE INDEX."

When to Create an Index

- **The column is used frequently in the WHERE clause or in a join condition.**

- **The column contains a wide range of values.**

- **The column contains a large number of null values.**

- **Two 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 2–4% of the rows.**

 ORACLE®

More Is Not Always Better

More indexes on a table does not mean it will speed up 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.

When to Create an Index

- The column is used frequently in the WHERE clause or in a join condition.
- The column contains a wide range of values.
- The column contains a large number of null values.
- Two 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.

When Not to Create an Index

- **The table is small.**
- **The columns are not often used as a condition in the query.**
- **Most queries are expected to retrieve more than 2–4% of the rows.**
- **The table is updated frequently.**

ORACLE®

When Not to Create an Index

- The table is small.
- The columns are not often used as a condition in the query.
- Most queries are expected to retrieve more than 2–4% of the rows.
- The table is updated frequently. If you have one or more indexes on a table, the DML statements that access the table take more time.

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.**

```
SQL> SELECT   ic.index_name, ic.column_name,
  2           ic.column_position col_pos,ix.uniqueness
  3  FROM      user_indexes ix, user_ind_columns ic
  4  WHERE     ic.index_name = ix.index_name
  5  AND       ic.table_name = 'EMP';
```

   ORACLE®

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, affected column names, and uniqueness on the EMP table.

```
INDEX_NAME          COLUMN_NAME      COL_POS UNIQUENESS
------------------  ---------------- ------- ----------
EMP_EMPNO_PK        EMPNO                  1 UNIQUE
EMP_ENAME_IDX       ENAME                  1 NONUNIQUE
```

Note: The output has been formatted.

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.

```
SQL> CREATE TABLE test (col1 NUMBER);

SQL> CREATE INDEX test_index on test(col1,col1+10);

SQL> SELECT col1+10 FROM test;
```

ORACLE®

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:

```
SQL> CREATE INDEX uppercase_idx ON emp (UPPER(ename));
```

Facilitates processing queries such as:

```
SQL> SELECT * FROM emp WHERE UPPER(ename) = 'KING';
```

To ensure that Oracle will use 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 statement below is guaranteed to use the index, but without the where clause Oracle may perform a full table scan.

```
SQL> SELECT * FROM emp
   2  WHERE UPPER (ename) IS NOT NULL
   3  ORDER BY UPPER (ename);
```

Oracle treats indexes with columns marked DESC as function-based indexes. The columns marked DESC are sorted in descending order.

Removing an Index

- **Remove an index from the data dictionary.**

```
SQL> DROP INDEX index;
```

- **Remove the EMP_ENAME_IDX index from the data dictionary.**

```
SQL> DROP INDEX emp_ename_idx;
Index dropped.
```

- **To drop an index, you must be the owner of the index or have the DROP ANY INDEX privilege.**

ORACLE®

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

Synonyms

Simplify access to objects by creating a synonym (another name for an object).

- **Refer to a table owned by another user.**
- **Shorten lengthy object names.**

```
CREATE [PUBLIC] SYNONYM synonym
FOR     object;
```

ORACLE®

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:

PUBLIC	creates a synonym accessible to all users
synonym	is the name of the synonym to be created
object	identifies the object for which the synonym is created

Guidelines

- The object cannot be contained in a package.
- A private synonym name must be distinct from all other objects owned by the same user.

For more information, see *Oracle Server SQL Reference,* Release 8, "CREATE SYNONYM."

Creating and Removing Synonyms

- **Create a shortened name for the DEPT_SUM_VU view.**

```
SQL> CREATE SYNONYM d_sum
  2  FOR    dept_sum_vu;
Synonym Created.
```

- **Drop a synonym.**

```
SQL> DROP SYNONYM d_sum;
Synonym dropped.
```

 ORACLE®

Creating a Synonym for an Object (continued)

The slide example creates a synonym for the DEPT_SUM_VU view for quicker reference.

The DBA can create a public synonym accessible to all users. The following example creates a public synonym named DEPT for Alice's DEPT table:

```
SQL> CREATE PUBLIC SYNONYM  dept
  2  FOR    alice.dept;
Synonym created.
```

Removing a Synonym

To drop a synonym, use the DROP SYNONYM statement. Only the DBA can drop a public synonym.

```
SQL> DROP SYNONYM  dept;
Synonym dropped.
```

For more information, see *Oracle Server SQL Reference,* Release 8, "DROP SYNONYM."

Summary

- **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.**

Sequences

The sequence generator can be used to automatically generate sequence numbers for rows in tables. This can be time saving 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 the 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

DBAs 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 Overview

- **Creating sequences**

- **Using sequences**

- **Creating nonunique indexes**

- **Displaying data dictionary information about sequences and indexes**

- **Dropping indexes**

 ORACLE®

Practice Overview

In this practice, you will create a sequence to be used when populating your DEPARTMENT table. You will also create implicit and explicit indexes.

Practice 13

1. Create a sequence to be used with the primary key column of the DEPARTMENT table. The sequence should start at 60 and have a maximum value of 200. Have your sequence increment by ten numbers. Name the sequence DEPT_ID_SEQ.

2. Write a script to display the following information about your sequences: sequence name, maximum value, increment size, and last number. Name the script `p13q2.sql`. Execute your script.

```
SEQUENCE_NAME MAX_VALUE INCREMENT_BY  LAST_NUMBER
------------- --------- ------------  -----------
CUSTID        1.000E+27            1          109
DEPT_ID_SEQ         200           10           60
ORDID         1.000E+27            1          622
PRODID        1.000E+27            1       200381
```

3. Write an interactive script to insert a row into the DEPARTMENT table. Name your script `p13q3.sql`. Be sure to use the sequence that you created for the ID column. Create a customized prompt to enter the department name. Execute your script. Add two departments named Education and Administration. Confirm your additions.

4. Create a non-unique index on the foreign key column (dept_id) in the EMPLOYEE table.

5. Display the indexes and uniqueness that exist in the data dictionary for the EMPLOYEE table. Save the statement into a script named `p13q5.sql`.

```
INDEX_NAME            TABLE_NAME    UNIQUENESS
--------------------- ------------- ----------
EMPLOYEE_DEPT_ID_IDX  EMPLOYEE      NONUNIQUE
EMPLOYEE_ID_PK        EMPLOYEE      UNIQUE
```