SRINIVAS UNIVERSITY

COLLEGE OF COMPUTER SCIENCE & INFORMATION SCIENCE

CITY CAMPUS, PANDESHWAR, MANGALORE – 575 001

BACKGROUND STUDY MATERIAL

DATABASE SYSTEMS

M.C.A - I SEMESTER

SRINIVAS GROUP

SAMAGRA GNANA

ESTD: 1988

Compiled by

Prof. Vaikunth Pai

Faculty, CCIS

DATABASE SYSTEMS

Sub. Code: 20MCADS11 **Total Hours: 40**
Hours/ Week: 04 **Exam Hours: 02**
IA Marks: 50 **Exam Marks: 50**

Course Objective
The objective of the course is to enable students to understand and use a relational database system. Introduction to Databases, Conceptual design using ERD, Functional dependencies and Normalization, Relational Algebra are covered in detail. Students learn how to design and create a good database and use various SQL operations

UNIT – I **- 8 Hrs**
Introduction to database and DBMS, Characteristics of Database, Actors on the scene, workers behind the scene, Advantages and Disadvantages of a DBMS, Data Models, Schemas, Instances, Three schema Architecture and Data Independence, database languages, Data Models for of Database Design, Entity types, entity sets, attributes and keys, relationship types, relationship sets, roles and structural constraints, weak entity types, ER diagram for company database.

UNIT – II **- 8 Hrs**
Relational Model Concepts, Relational Model Constraints and Relational Database schemas, Update Operations, Transactions, and Dealing with Constraint Violations, Relational Database Design Using ER-to-Relational Mapping, Informal Design Guidelines for Relational Schemas, Functional Dependencies, First Normal Form, Second Normal Form, Third Normal Form, Boyce-Codd Normal Form

UNIT – III **- 8 Hrs**
Introduction to Oracle9i, Oracle9i Application Server, Oracle9i Database, Communicating with a RDBMS Using SQL, Relational Database Management System, SQL statements, Writing Basic SQL SELECT Statements, Capabilities of SQL SELECT Statements, Basic SELECT Statement, Selecting All Columns and Specific Columns, Column Heading Defaults, Arithmetic Expressions, Defining a Column Alias, Concatenation Operator, Literal Character Strings, Eliminating Duplicate Rows, Displaying Table Structure, Restricting and Sorting Data, Single-Row Functions – character, number, date, conversion, general

UNIT – IV **- 8 Hrs**
Displaying Data from Multiple Tables, Cartesian Products, different types of Joins, Aggregating Data Using Group Functions, Types of Group Functions, Subqueries, different types of subqueries, Manipulating Data - insert, update, delete, merge, Creating and Managing Tables, Including Constraints, Creating Views

UNIT – V **- 8 Hrs**
Other Database Objects – Sequence, index, synonyms, Controlling User Access – Privileges, Role, Enhancements to the GROUP BY Clause - GROUP BY with ROLLUP and CUBE operators, Hierarchical Retrieval, Oracle9i Extensions to DML and DDL Statements

Reference Books:

1. Fundamentals of Database Systems, Ramez Elmasri, Shamkant B. Navathe, Seventh Edition.
2. Introduction to Oracle 9i: SQL, Oracle University

Chapter 1: Introduction

Oracle9*i*

Oracle9*i* Application Server

Oracle9*i* Database

Relational and Object Relational Database Management System

Oracle Internet Platform

System Development Life Cycle

Data Storage on Different Media

Relational Database Concept

Definition of a Relational Database

Data Models

Entity Relationship Model

Entity Relationship Modeling Conventions

Relating Multiple Tables

Relational Database Terminology

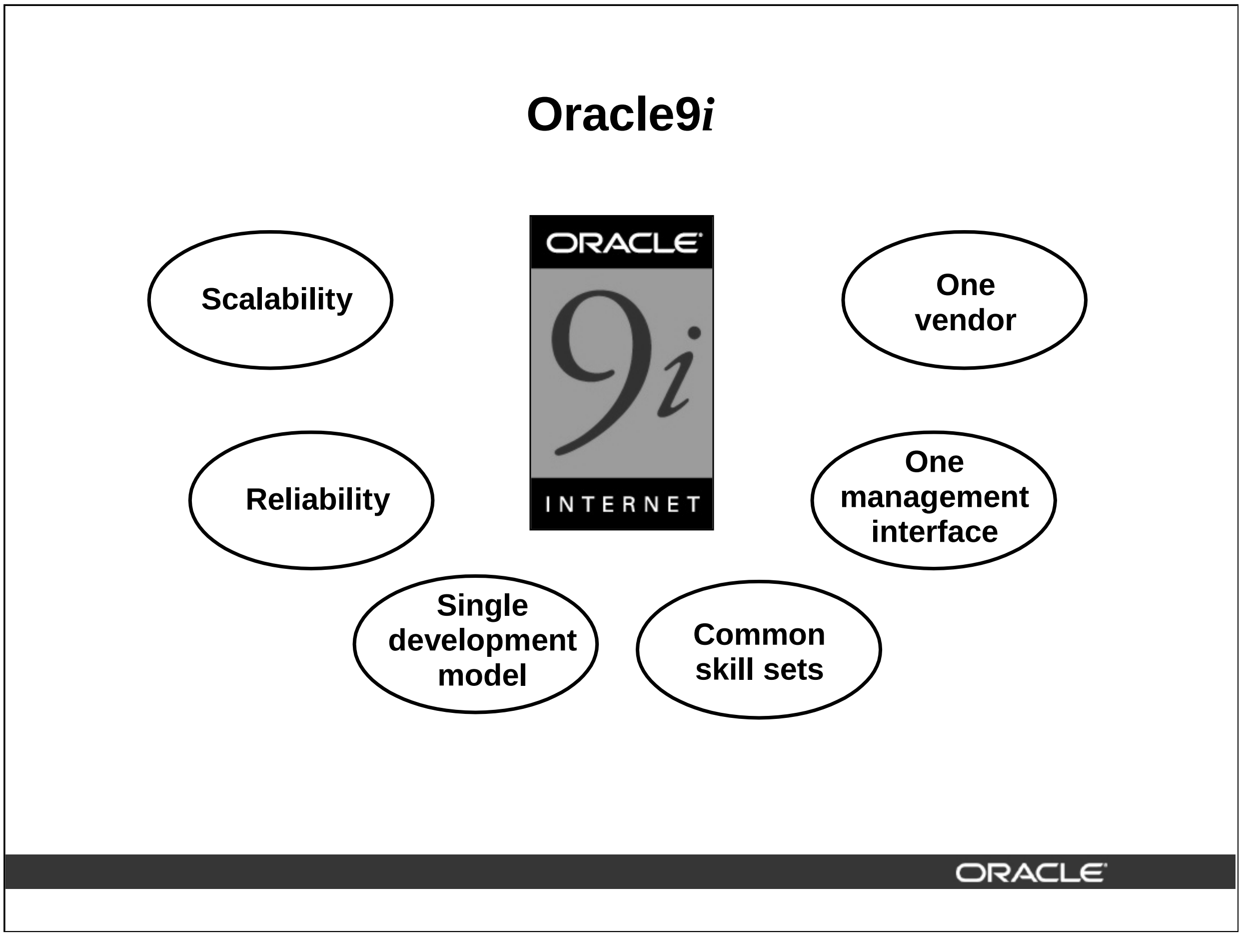
Relational Database Properties

Communicating with a RDBMS Using SQL

Relational Database Management System

SQL Statements

Summary

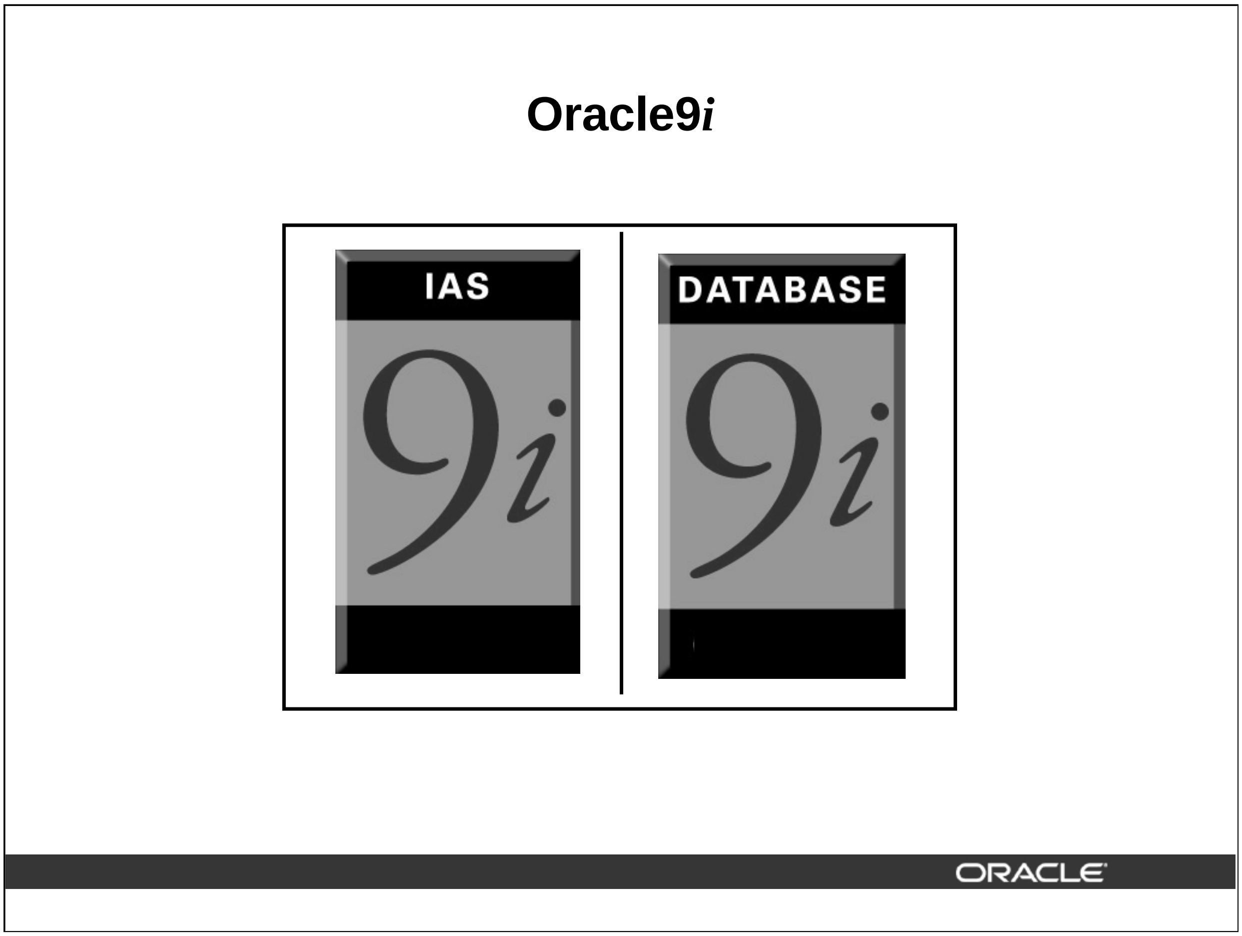


Oracle9*i* Features

Oracle offers a comprehensive high-performance infrastructure for e-business. It is called Oracle9*i*. Oracle9*i* includes everything needed to develop, deploy, and manage Internet applications.

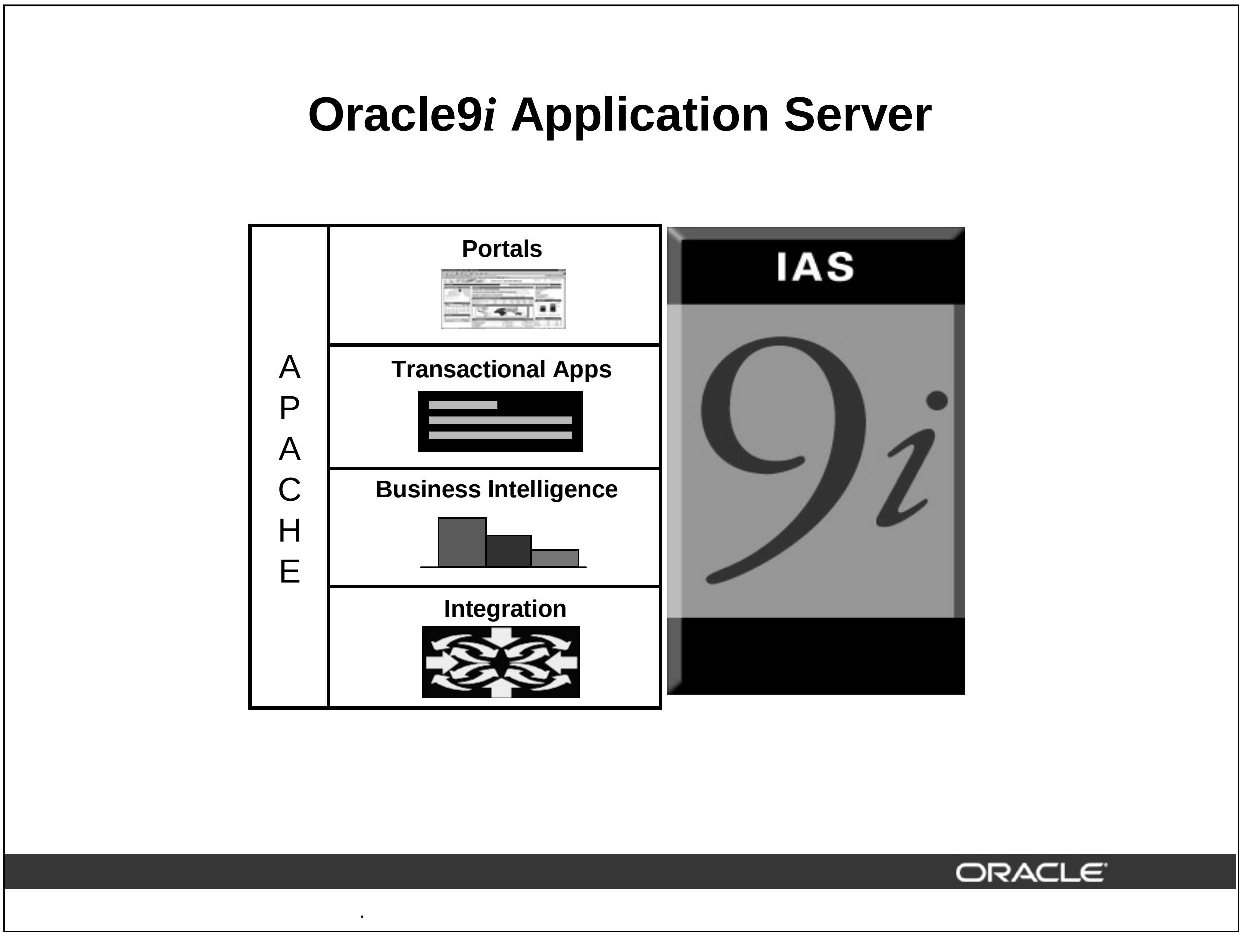
Benefits include:

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



Oracle9i

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



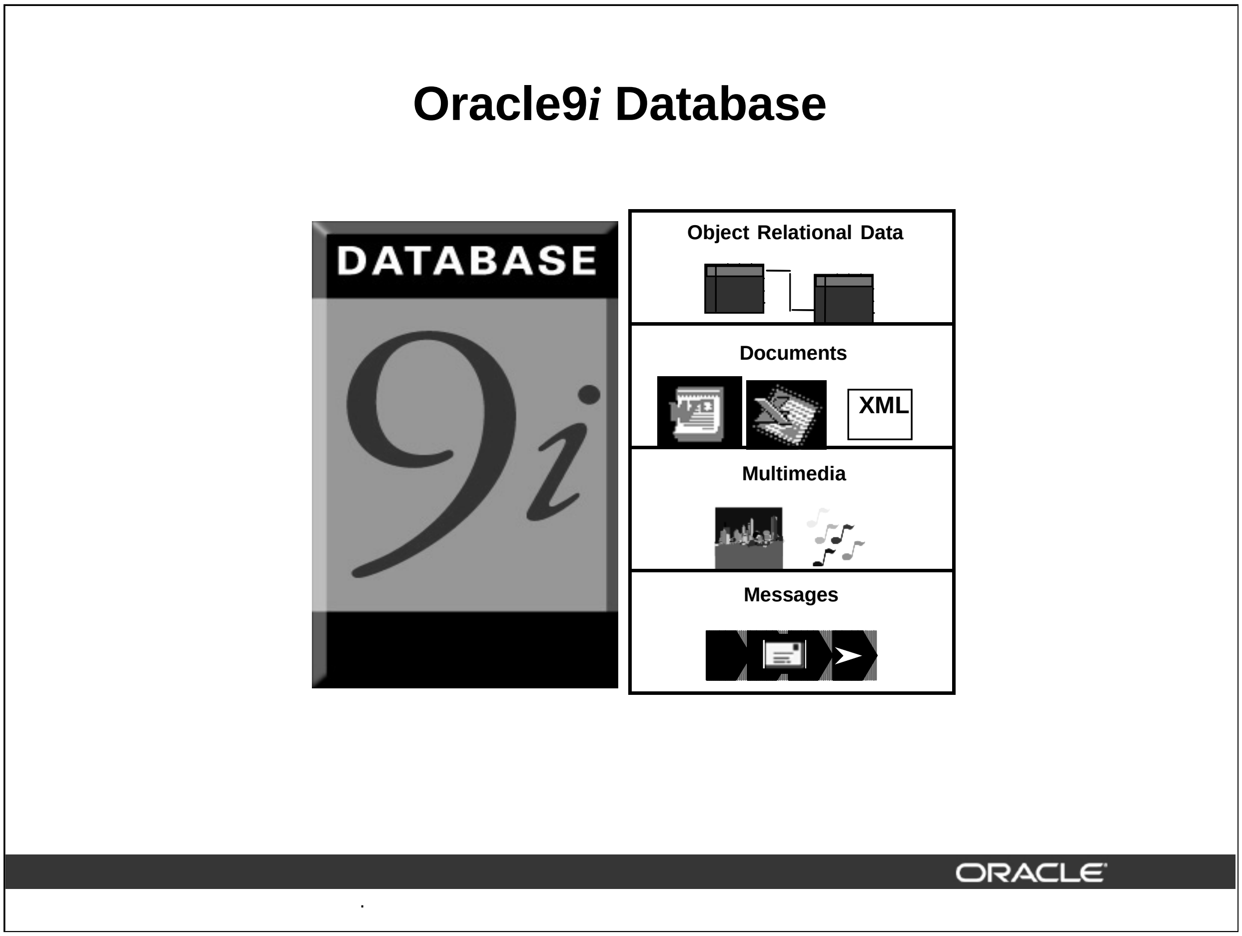
Oracle9*i* Application Server

The Oracle9*i* Application Server (Oracle9*i*AS) runs all your applications. The Oracle9*i* Database stores all your data.

Oracle9*i* Application Server is the only application server to include services for all the different server applications you will want to run. Oracle9*i*AS can run your:

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

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



Oracle9*i* Database

The roles of the two products are very straightforward. Oracle9*i* Database manages all your data. This is not just the object relational data that you expect an enterprise database to manage. It can also be unstructured data like:

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

The data does not even have to be in the database. Oracle9*i* Database has services through which you can store metadata about information stored in file systems. You can use the database server to manage and serve information wherever it is located.

Relational and Object Relational Database Management System

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

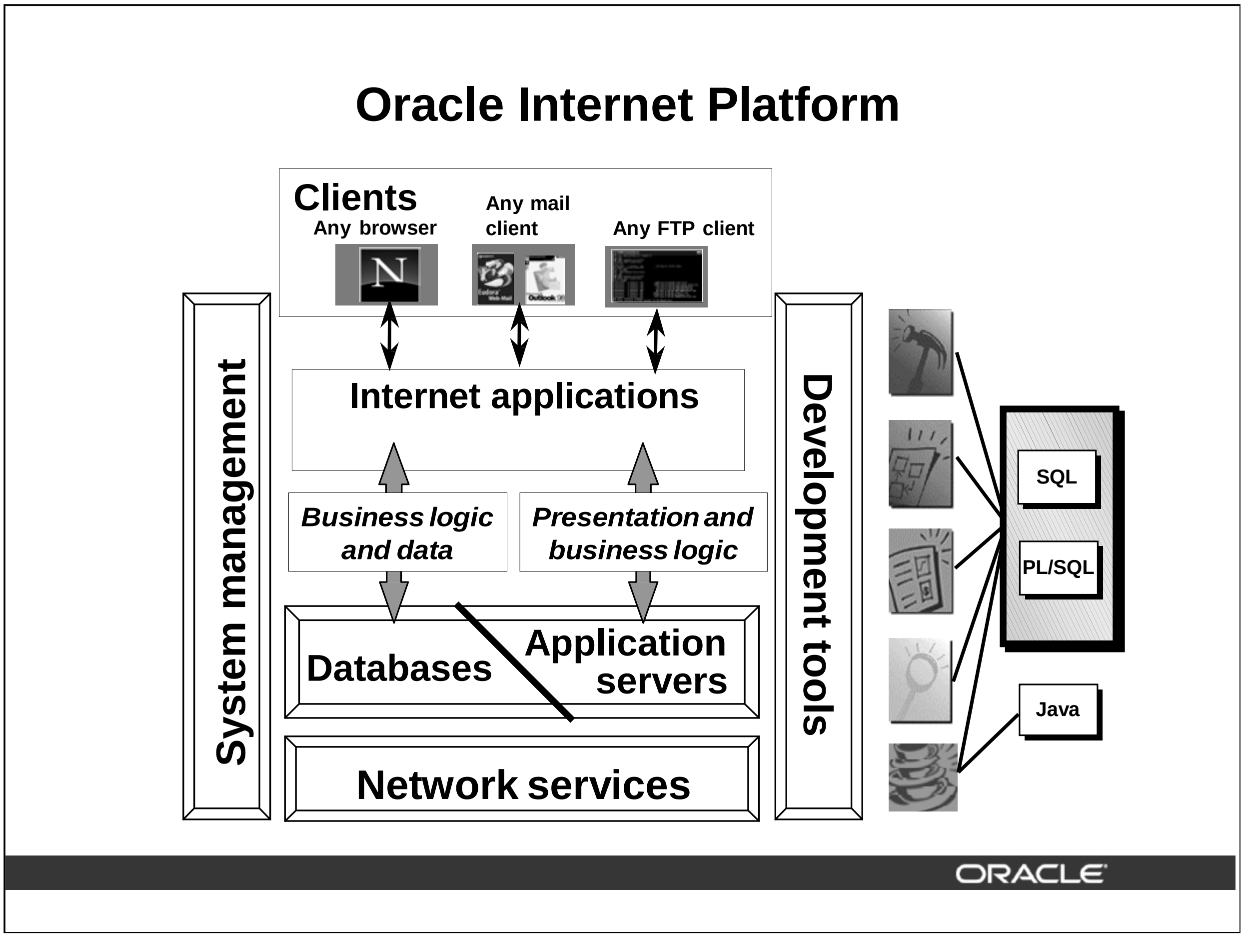
ORACLE

About the Oracle Server

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

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

It includes several features for improved performance and functionality of online transaction processing (OLTP) applications, such as better sharing of run-time data structures, larger buffer caches, and deferrable constraints. Data warehouse applications will benefit from enhancements such as parallel execution of insert, update, and delete operations; partitioning; and parallel-aware query optimization. Operating within the Network Computing Architecture (NCA) framework, Oracle9i supports client-server and Web-based applications that are distributed and multitiered.

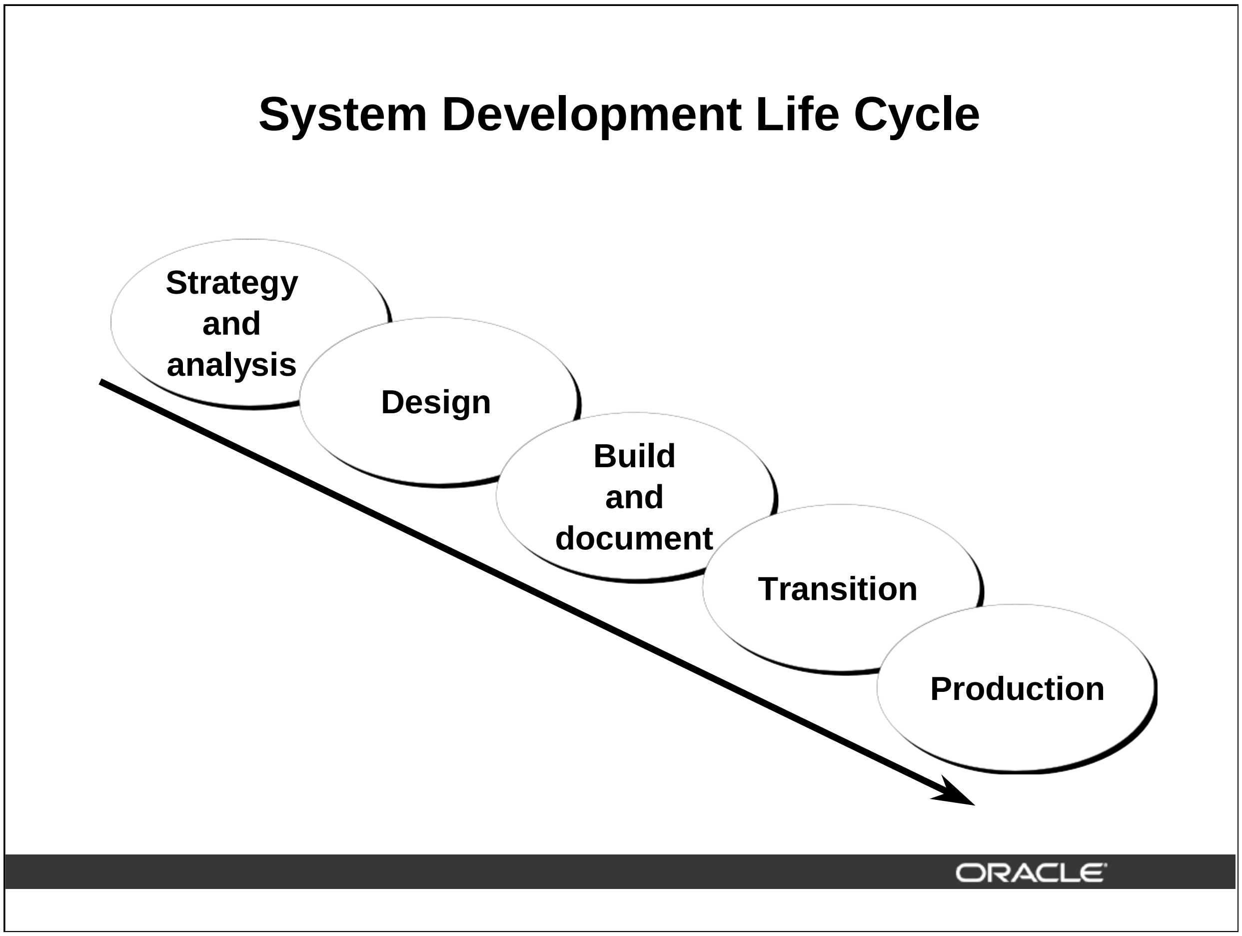


Oracle Internet Platform

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

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

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



System Development Life Cycle

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.

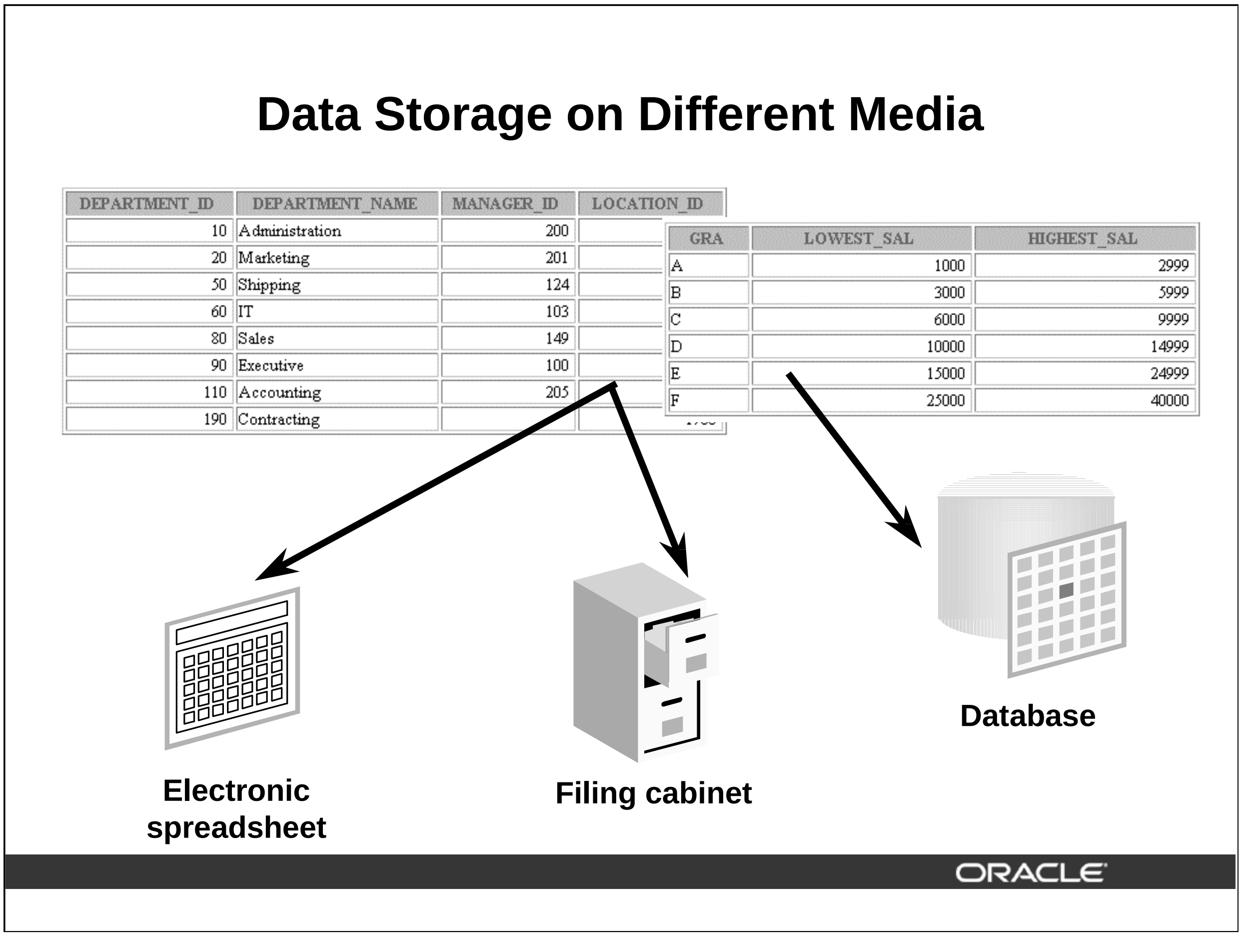
Transition

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

Production

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

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



Storing Information

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

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

A *database* is an organized collection of information.

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

Relational Database Concept

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

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

Definition of a Relational Database

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

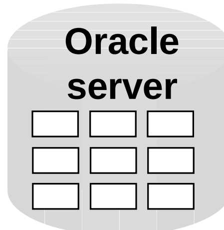


Table Name: **EMPLOYEES**

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	P
100	Steven	King	SKING	51
101	Neena	Kochhar	NKOCHHAR	51
102	Lex	De Haan	LDEHAAN	51

...

Table Name: **DEPARTMENTS**

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

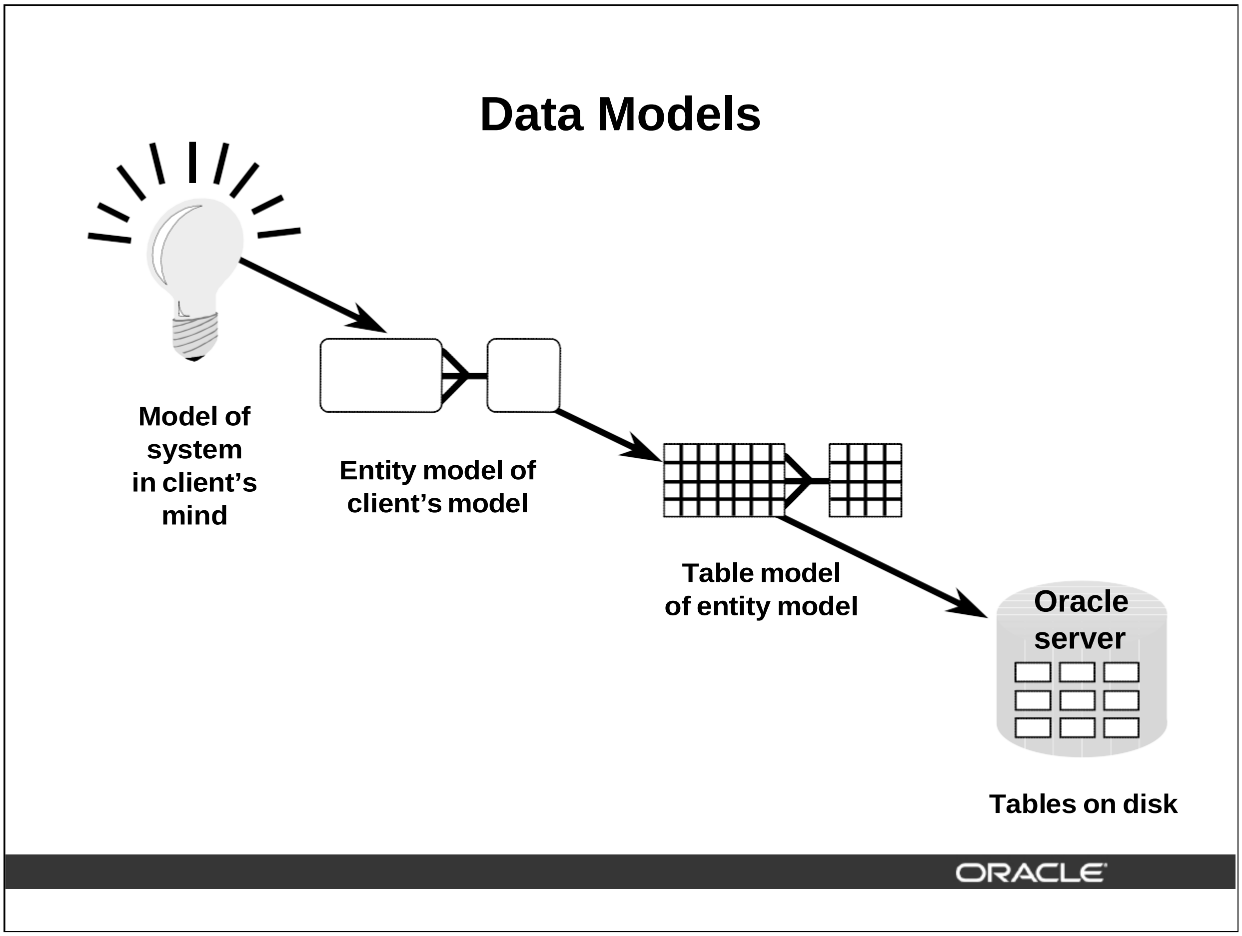
...

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

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.

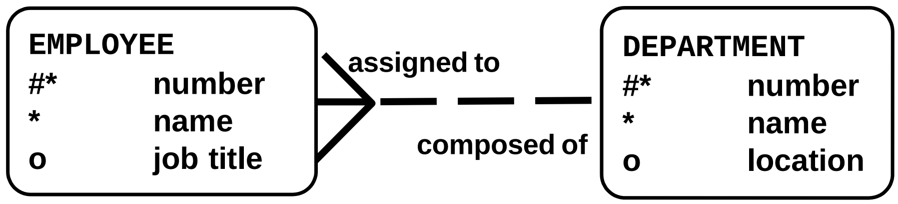
Purpose of Models

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

Entity Relationship Model

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



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

ORACLE

ER Modeling

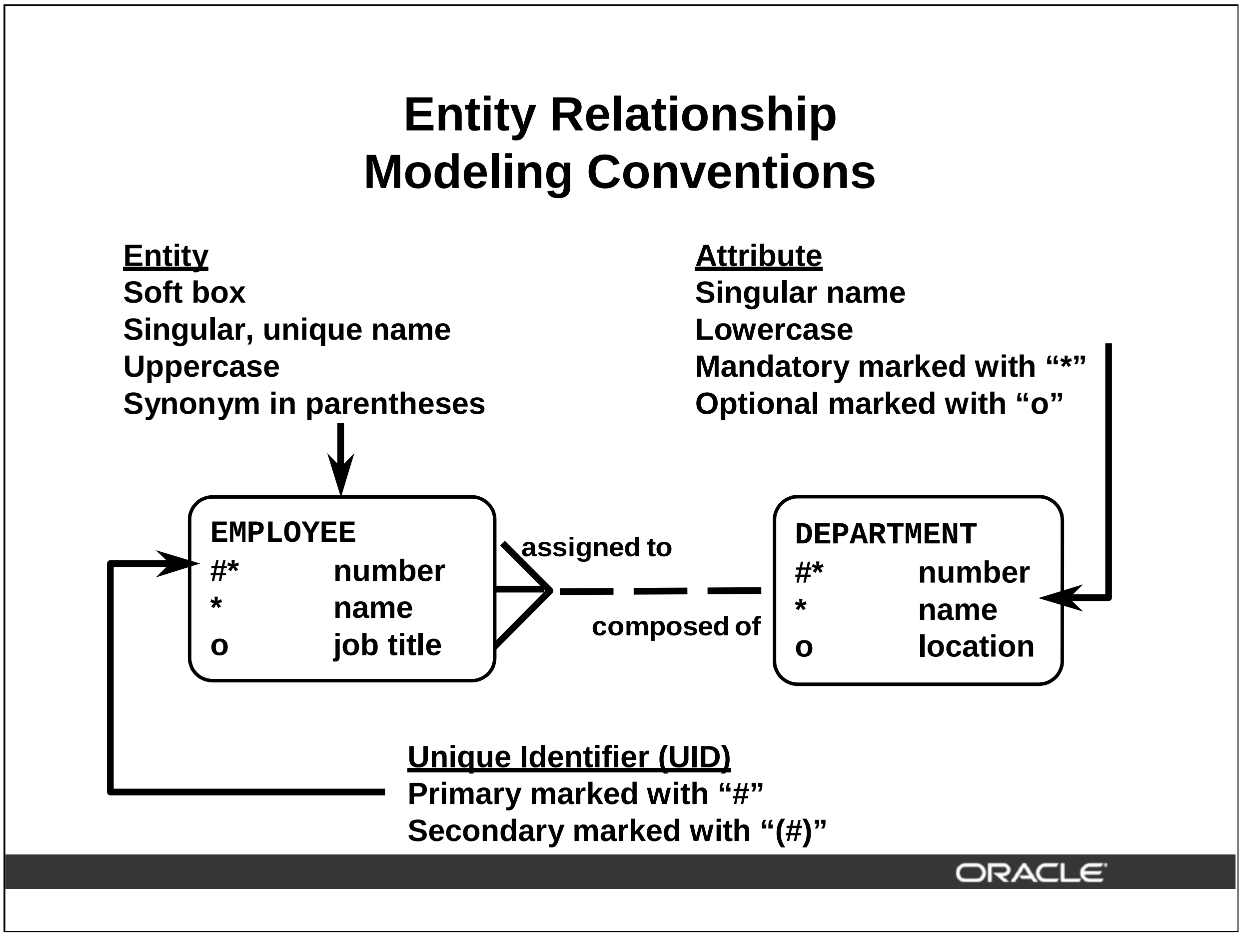
In an effective system, data is divided into discrete categories or entities. An entity relationship (ER) model is an illustration of various entities in a business and the relationships between them. An ER model is derived from business specifications or narratives and built during the analysis phase 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.



ER Modeling (continued)

Entities

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

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

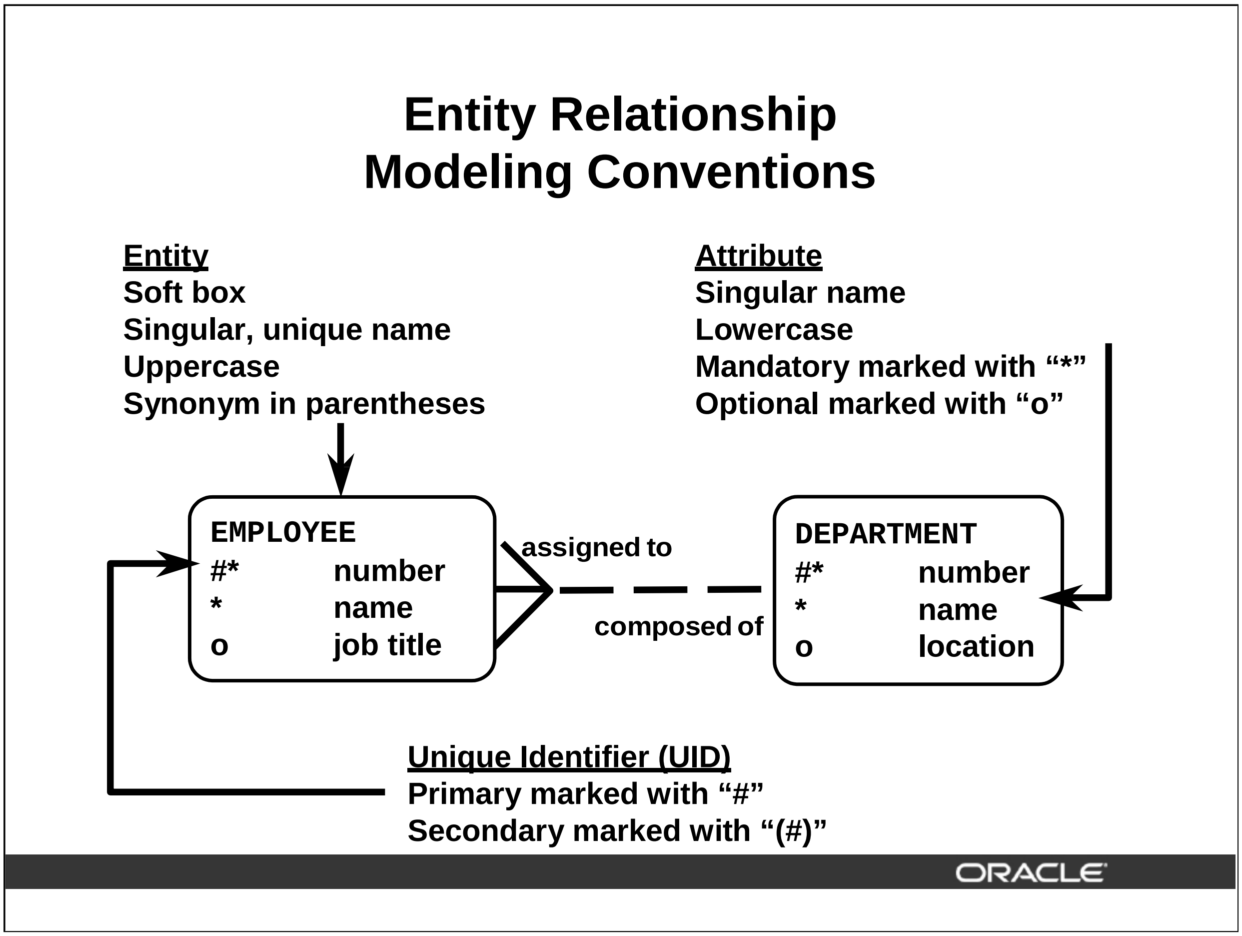
Attributes

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

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

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”



ER Modeling (continued)

Relationships

Each direction of the relationship contains:

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

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

Each source entity {may be | must be} relationship name {one and only one | one or more} destination entity.

Note: The convention is to read clockwise.

Unique Identifiers

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

- Tag each attribute that is part of the UID with a number symbol: #
- Tag secondary UIDs with a number sign in parentheses: (#)

Relating Multiple Tables

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

Table Name: **`EMPLOYEES`**

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

…

Primary key (EMPLOYEE_ID) **Foreign key** (DEPARTMENT_ID)

Table Name: **`DEPARTMENTS`**

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

Primary key (DEPARTMENT_ID)

Relating Multiple Tables

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

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

Guidelines for Primary Keys and Foreign Keys

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

Relational Database Terminology

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

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.

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

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

Relational Database Properties

A relational database:

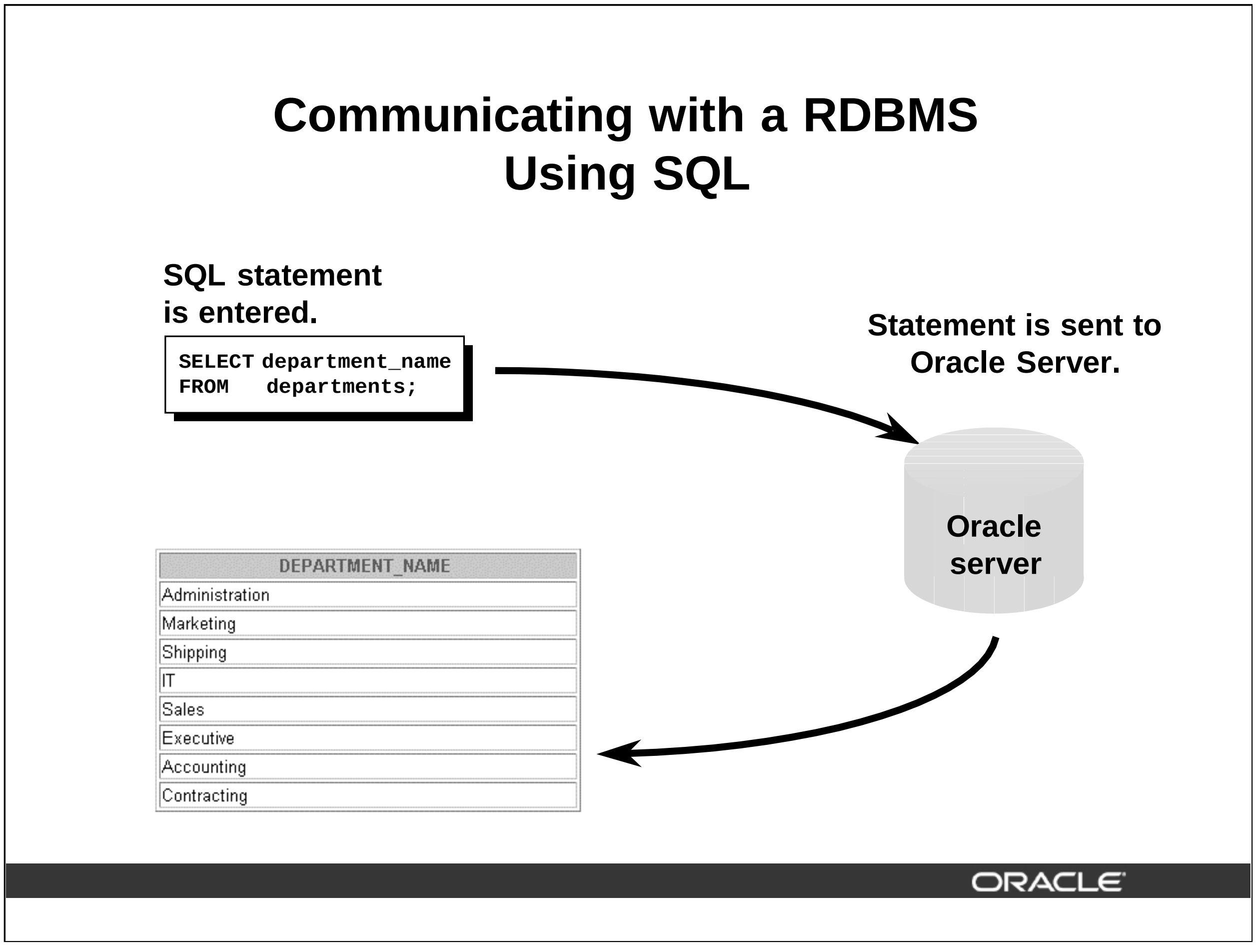
- **Can be accessed and modified by executing structured query language (SQL) statements**
- **Contains a collection of tables with no physical pointers**
- **Uses a set of operators**

ORACLE

Properties of a Relational Database

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

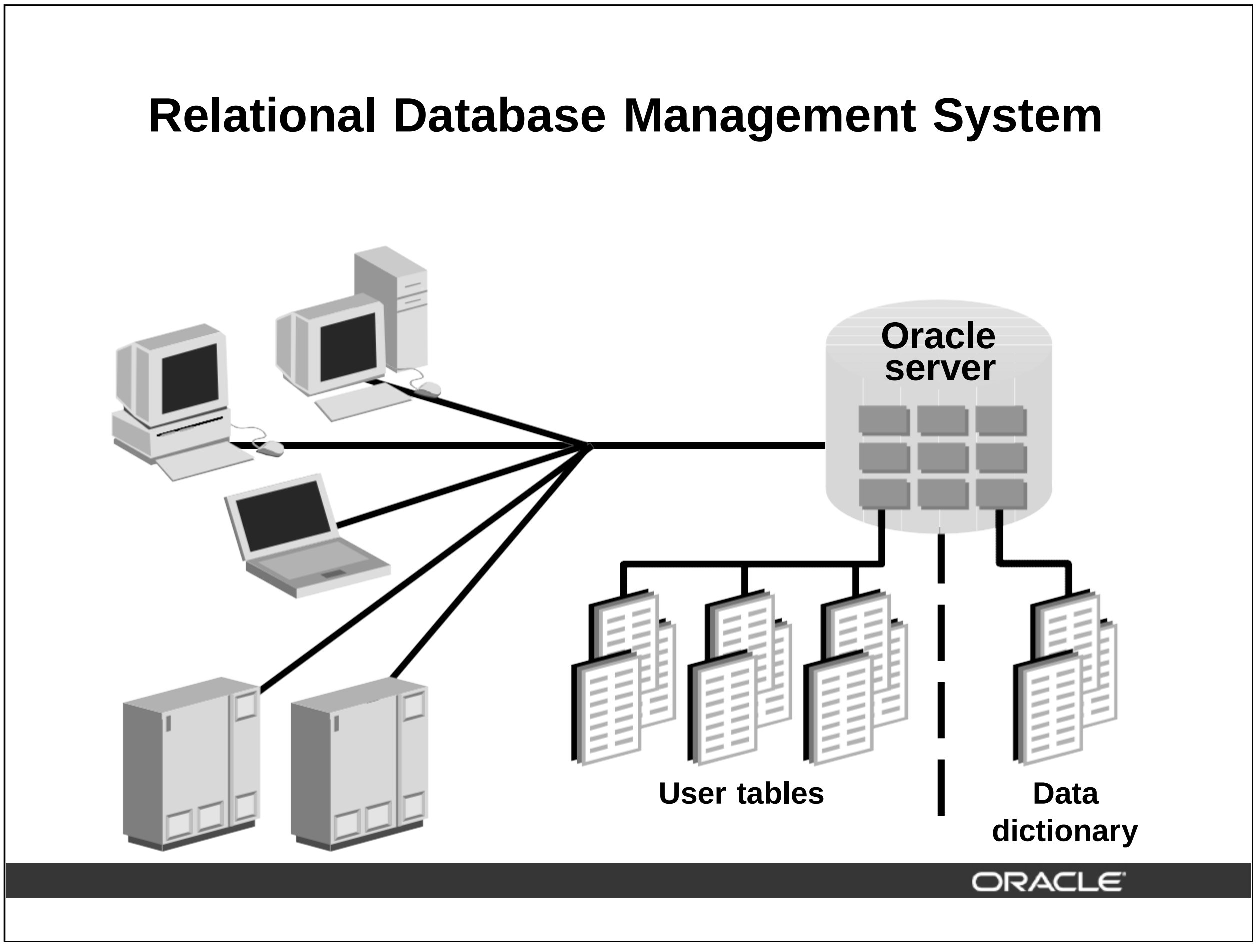
To access the database, you execute a structured query language (SQL) statement, which is the American National Standards Institute (ANSI) standard language for operating relational databases. The language contains a large set of operators for partitioning and combining relations. The database can be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



Structured Query Language

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

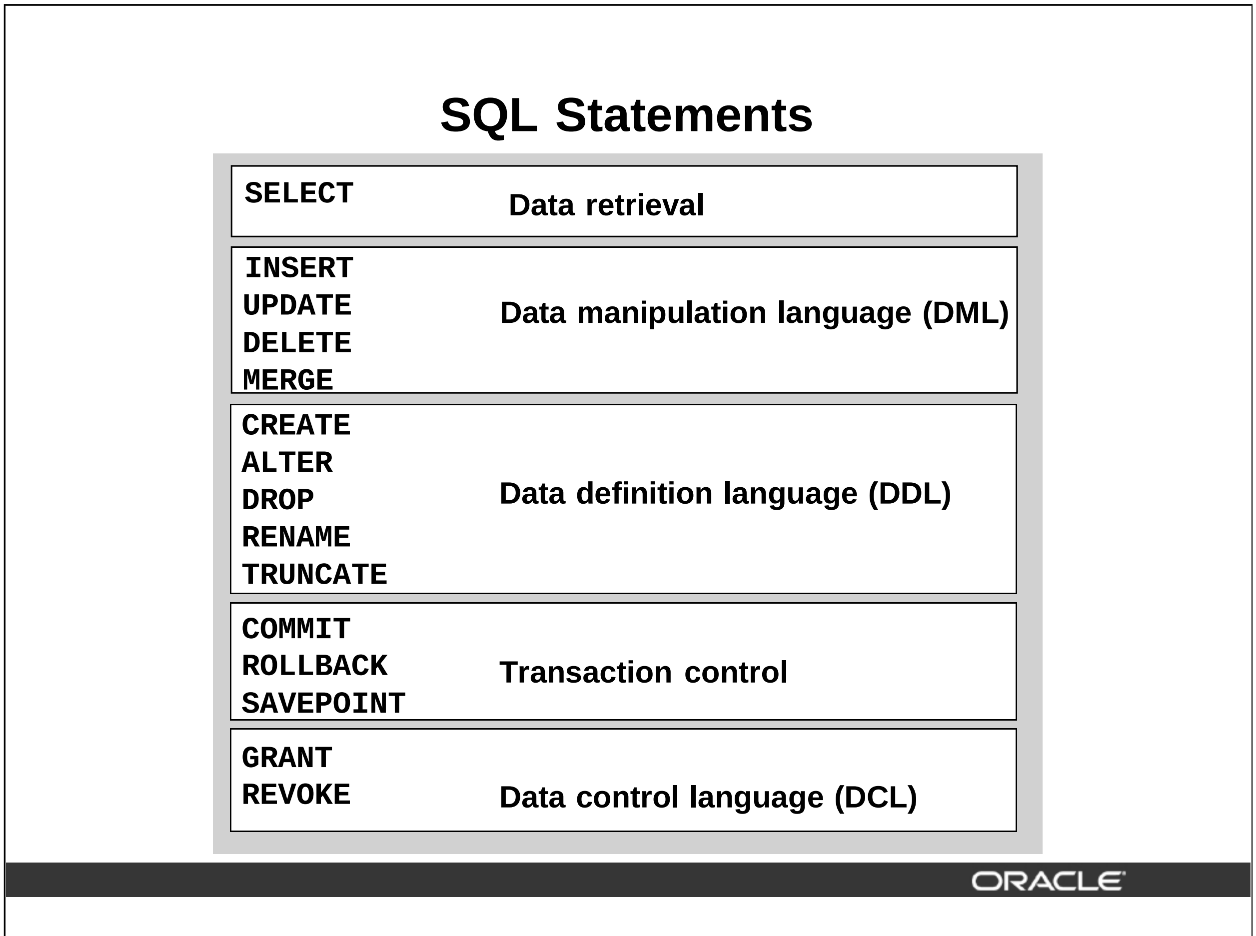
- Efficient
- Easy to learn and use
- Functionally complete (With SQL, you can define, retrieve, and manipulate data in the tables.)



Relational Database Management System

Oracle provides a flexible RDBMS called Oracle9*i*. 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. Oracle9*i* also supports Java and XML. The Oracle server offers the options of retrieving data based on optimization techniques. It includes security features that control how a database is accessed and used. Other features include consistency and protection of data through locking mechanisms.

The Oracle9*i* server provides an open, comprehensive, and integrated approach to information management. An Oracle server consists of an Oracle database and an Oracle server instance. Every time a database is started, a system global area (SGA) is allocated, and Oracle background processes are started. The system global area is an area of memory used for database information shared by the database users. The combination of the background processes and memory buffers is called an Oracle instance.



SQL Statements

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

Statement	Description
SELECT	Retrieves data from the database
INSERT UPDATE DELETE MERGE	Enters new rows, changes existing rows, and removes unwanted rows from tables in the database, respectively. Collectively known as *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).

Summary

- **The Oracle9*i* Server is the database for Internet computing.**
- **Oracle9*i* is based on the object relational database management system.**
- **Relational databases are composed of relations, managed by relational operations, and governed by data integrity constraints.**
- **With the Oracle Server, you can store and manage information by using the SQL language and PL/SQL engine.**

ORACLE

Summary

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

Oracle Corporation produces products and services to meet your relational database management system needs. The main products are the Oracle9*i* Database Server, with which you can store and manage information by using SQL, and the Oracle9*i* Application Server with which you can run all of your applications.

SQL

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

Chapter 2: Writing Basic SQL SELECT Statements

Objectives

Capabilities of SQL SELECT Statements

Basic SELECT Statement

Selecting All Columns

Selecting Specific Columns

Writing SQL Statements

Column Heading Defaults

Arithmetic Expressions

Using Arithmetic Operators

Operator Precedence

Using Parentheses

Defining a Null Value

Null Values in Arithmetic Expressions

Defining a Column Alias

Using Column Aliases

Concatenation Operator

Using the Concatenation Operator

Literal Character Strings

Using Literal Character Strings

Duplicate Rows

Eliminating Duplicate Rows

SQL and *i*SQL*Plus Interaction

SQL Statements Versus iSQL*Plus Commands

Overview of *i*SQL*Plus

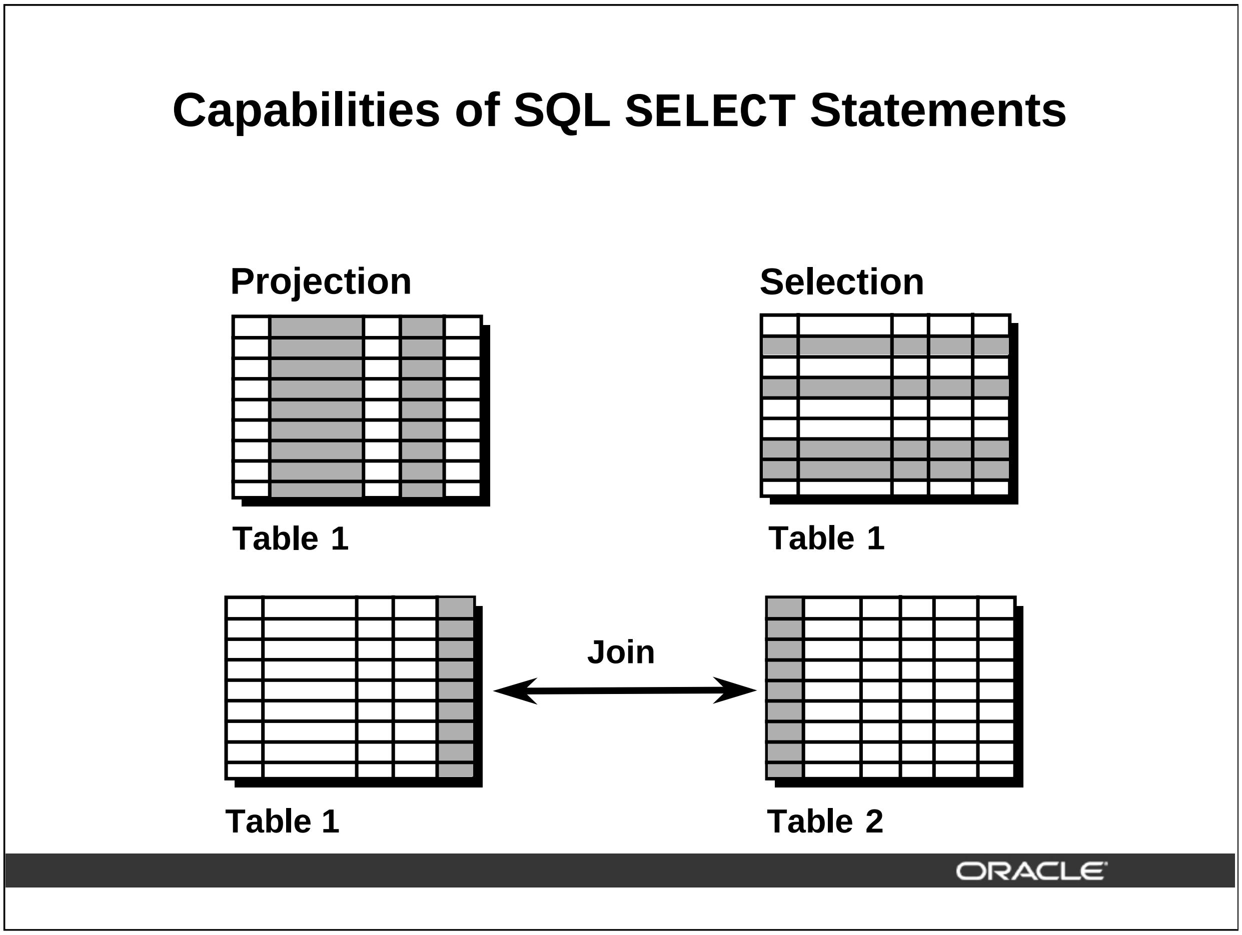
Logging In to *i*SQL*Plus

The *i*SQL*Plus Environment

Displaying Table Structure

Interacting with Script Files

Summary



Capabilities of SQL SELECTStatements

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

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

Basic SELECT Statement

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

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

Basic SELECT Statement

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

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

In the syntax:

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

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

- A *keyword* refers to an individual SQL element.
  For example, SELECT and FROM are keywords.
- A *clause* is a part of a SQL statement.
  For example, SELECT employee_id, last_name, ... is a clause.
  For example, SELECT * FROM employees is a SQL statement.

Selecting All Columns

```
SELECT *
FROM   departments;
```

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

8 rows selected.

ORACLE

Selecting All Columns of All Rows

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

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

SELECT department_id, department_name, manager_id, location_id FROM
departments;

Selecting Specific Columns

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

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

8 rows selected.

Selecting Specific Columns of All Rows

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

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

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

LOCATION_ID	DEPARTMENT_ID
1700	10
1800	20
1500	50

...

8 rows selected.

Writing SQL Statements

- **SQL statements are not case sensitive.**
- **SQL statements can be on one or more lines.**
- **Keywords cannot be abbreviated or split across lines.**
- **Clauses are usually placed on separate lines.**
- **Indents are used to enhance readability.**

Writing SQL Statements

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

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

Executing SQL Statements

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

Column Heading Defaults

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

ORACLE

Column Heading Defaults

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

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

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

20 rows selected.

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

Arithmetic Expressions

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

Operator	Description
+	Add
-	Subtract
*	Multiply
/	Divide

ORACLE

Arithmetic Expressions

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

Arithmetic Operators

The slide lists the arithmetic operators available in SQL. You can use arithmetic operators in any clause of a SQL statement except in the FROMclause.

Using Arithmetic Operators

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

LAST_NAME	SALARY	SALARY+300
King	24000	24300
Kochhar	17000	17300
De Haan	17000	17300
Hunold	9000	9300
Ernst	6000	6300
...		
Hartstein	13000	13300
Fay	6000	6300
Higgins	12000	12300
Gietz	8300	8600

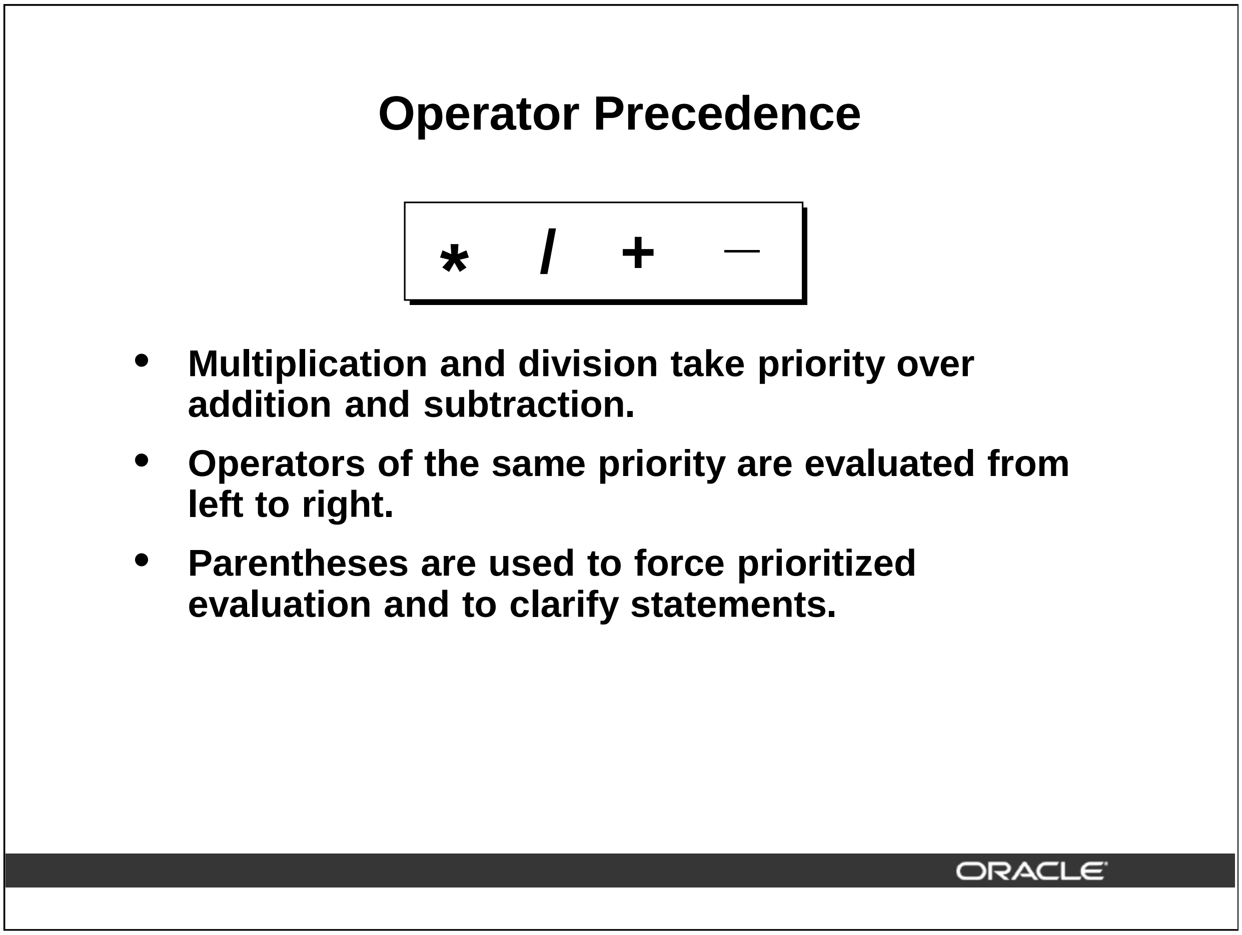
20 rows selected.

Using Arithmetic Operators

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

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

Note: The Oracle9*i* server ignores blank spaces before and after the arithmetic operator.



Operator Precedence

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

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

Operator Precedence

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

LAST_NAME	SALARY	12*SALARY+100
King	24000	288100
Kochhar	17000	204100
De Haan	17000	204100
Hunold	9000	108100
Ernst	6000	72100
...		
Hartstein	13000	156100
Fay	6000	72100
Higgins	12000	144100
Gietz	8300	99700

20 rows selected.

ORACLE

Operator Precedence (continued)

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

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

Using Parentheses

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

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

20 rows selected.

Using Parentheses

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

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

Defining a Null Value

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

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

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

20 rows selected.

ORACLE

Null Values

If a row lacks the data value for a particular column, that value is said to be *null*, or to contain a null. A null is a value that is unavailable, unassigned, unknown, or inapplicable. A null is not the same as zero or a space. Zero is a number, and a space is a character.

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

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

Null Values in Arithmetic Expressions

Arithmetic expressions containing a null value evaluate to null.

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

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

20 rows selected.

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 does not get any commission. Because the COMMISSION_PCTcolumn in the arithmetic expression is null, the result is null.

Defining a Column Alias

A column alias:

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

ORACLE

Column Aliases

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

Specify the alias after the column in the SELECTlist using a space as a separator. By default, alias headings appear in uppercase. If the alias contains spaces or special characters (such as # or $), or is case sensitive, enclose the alias in double quotation marks (" ").

Using Column Aliases

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

NAME	COMM
King	
Kochhar	
De Haan	

...

20 rows selected.

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

Name	Annual Salary
King	288000
Kochhar	204000
De Haan	204000

...

20 rows selected.

ORACLE

Column Aliases (continued)

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

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

Concatenation Operator

A concatenation operator:

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

Concatenation Operator

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

Using the Concatenation Operator

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

Employees
KingAD_PRES
KochharAD_VP
De HaanAD_VP
HunoldIT_PROG
ErnstIT_PROG
LorentzIT_PROG
MourgosST_MAN
RajsST_CLERK

...

20 rows selected.

ORACLE

Concatenation Operator (continued)

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

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

Literal Character Strings

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

ORACLE

Literal Character Strings

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

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

Using Literal Character Strings

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

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

...

20 rows selected.

ORACLE

Literal Character Strings (continued)

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

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

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

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

...

20 rows selected.

Duplicate Rows

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

```
SELECT department_id
FROM   employees;
```

DEPARTMENT_ID
90
90
90
60
60
60
50
50
50

...

20 rows selected.

Duplicate Rows

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

Eliminating Duplicate Rows

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

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

DEPARTMENT_ID
10
20
50
60
80
90
110

8 rows selected.

ORACLE

Duplicate Rows (continued)

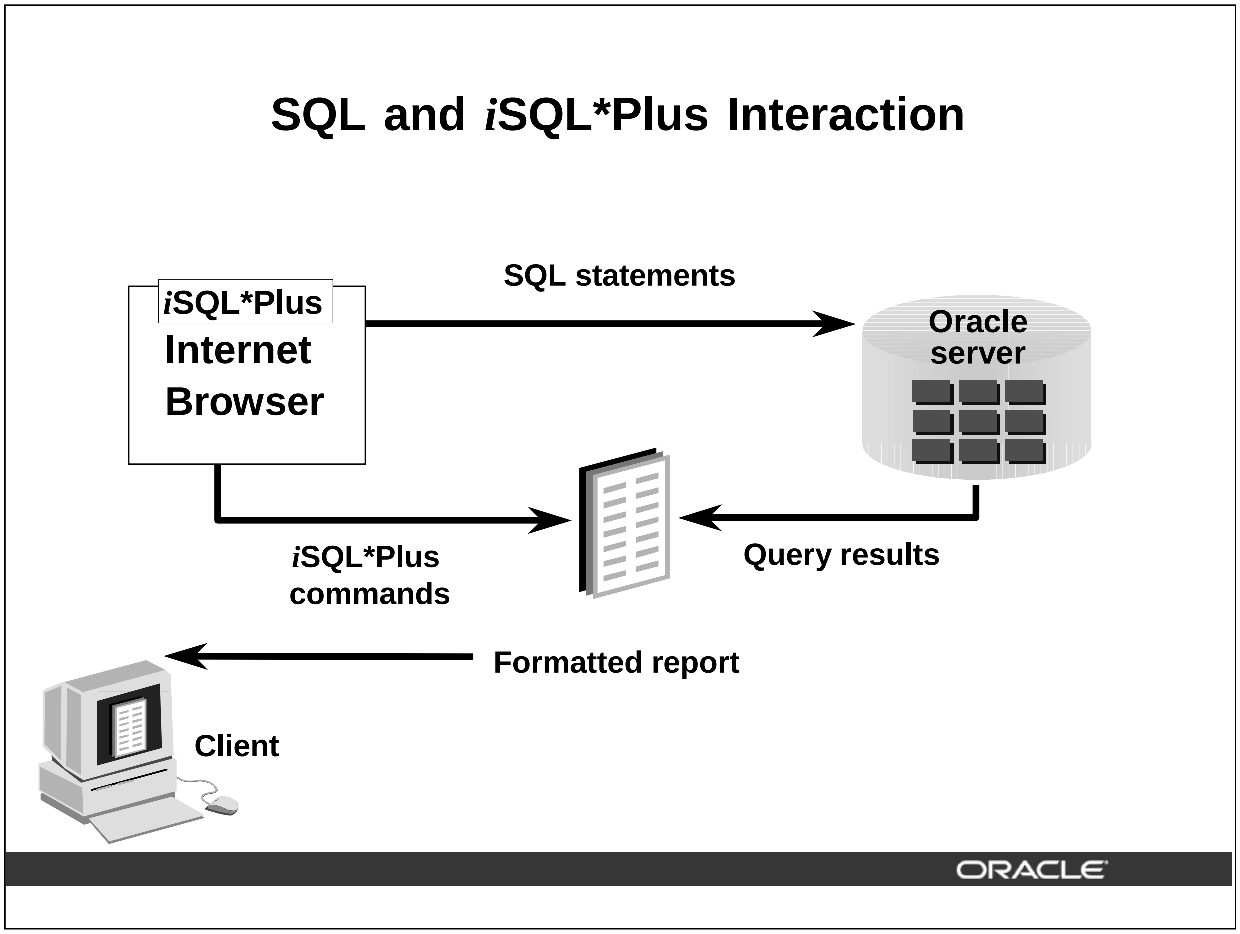
To eliminate duplicate rows in the result, include the `DISTINCT` keyword in the `SELECT` clause immediately after the `SELECT` keyword. In the example on the slide, the `EMPLOYEES` table actually contains 20 rows but there are only seven unique department numbers in the table.

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

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

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

13 rows selected.



SQL and *i*SQL*Plus

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

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

Features of SQL

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

Features of *i*SQL*Plus

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

Displaying Table Structure

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

```
DESC[RIBE] tablename
```

ORACLE

Displaying the Table Structure

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

In the syntax:

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

Displaying Table Structure

```
DESCRIBE employees
```

Name	Null?	Type
EMPLOYEE_ID	NOT NULL	NUMBER(6)
FIRST_NAME		VARCHAR2(20)
LAST_NAME	NOT NULL	VARCHAR2(25)
EMAIL	NOT NULL	VARCHAR2(25)
PHONE_NUMBER		VARCHAR2(20)
HIRE_DATE	NOT NULL	DATE
JOB_ID	NOT NULL	VARCHAR2(10)
SALARY		NUMBER(8,2)
COMMISSION_PCT		NUMBER(2,2)
MANAGER_ID		NUMBER(6)
DEPARTMENT_ID		NUMBER(4)

ORACLE

Displaying the Table Structure (continued)

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

In the result:

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

Type displays the data type for a column

The data types are described in the following table:

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

Summary

In this lesson, you should have learned how to:

- **Write a SELECT statement that:**
  - **Returns all rows and columns from a table**
  - **Returns specified columns from a table**
  - **Uses column aliases to give descriptive column headings**
- **Use the *i*SQL*Plus environment to write, save, and execute SQL statements and *i*SQL*Plus commands.**

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

ORACLE

SELECT Statement

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

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

In the syntax:

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

*i*SQL*Plus

*i*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 1

1. Initiate an *i*SQL*Plus session using the user ID and password provided by the instructor.
2. *i*SQL*Plus commands access the database.
   True/False
3. The following `SELECT` statement executes successfully:

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

   True/False
4. The following `SELECT` statement executes successfully:

   ```
   SELECT *
   FROM   job_grades;
   ```

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

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

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

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

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

8 rows selected.

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

Name	Null?	Type
EMPLOYEE_ID	NOT NULL	NUMBER(6)
FIRST_NAME		VARCHAR2(20)
LAST_NAME	NOT NULL	VARCHAR2(25)
EMAIL	NOT NULL	VARCHAR2(25)
PHONE_NUMBER		VARCHAR2(20)
HIRE_DATE	NOT NULL	DATE
JOB_ID	NOT NULL	VARCHAR2(10)
SALARY		NUMBER(8,2)
COMMISSION_PCT		NUMBER(2,2)
MANAGER_ID		NUMBER(6)
DEPARTMENT_ID		NUMBER(4)

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

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

20 rows selected.

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

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

12 rows selected.

If you have time, complete the following exercises:

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

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

20 rows selected.

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

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

20 rows selected.

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

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

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

20 rows selected.

Chapter 3: Restricting and Sorting Data

Objectives

Limiting Rows Using a Selection

Limiting the Rows Selected

Using the WHERE Clause

Character Strings and Dates

Comparison Conditions

Using Comparison Conditions

Other Comparison Conditions

Using the BETWEEN Condition

Using the IN Condition

Using the LIKE Condition

Using the NULL Conditions

Logical Conditions

Using the AND Operator

Using the OR Operator

Using the NOT Operator

Rules of Precedence

ORDER BY Clause

Sorting in Descending Order

Sorting by Column Alias

Sorting by Multiple Columns

Summary

Limiting Rows Using a Selection

EMPLOYEES

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

...

20 rows selected.

"retrieve all employees in department 90"

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

Limiting Rows Using a Selection

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

Limiting the Rows Selected

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

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

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

ORACLE

Limiting the Rows Selected

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

In the syntax:

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

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

Using the WHERE Clause

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

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

ORACLE

Using the WHERE Clause

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

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

Character Strings and Dates

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

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

ORACLE

Character Strings and Dates

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

All character searches are case sensitive. In the following example, no rows are returned because the

EMPLOYEES table stores all the last names in mixed case:

SELECT last_name, job_id, department_id FROM employees

WHERE last_name = 'WHALEN';

Comparison Conditions

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

ORACLE

Comparison Conditions

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

Syntax

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

For Example

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

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

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

Using Comparison Conditions

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

LAST_NAME	SALARY
Matos	2600
Vargas	2500

Using the Comparison Conditions

In the example, the `SELECT` statement retrieves the last name and salary from the `EMPLOYEES` table, where the employee salary is less than or equal to 3000. Note that there is an explicit value supplied to the `WHERE` clause. The explicit value of 3000 is compared to the salary value in the `SALARY` column of the `EMPLOYEES` table.

Other Comparison Conditions

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

Using the BETWEEN Condition

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

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

Lower limit **Upper limit**

LAST_NAME	SALARY
Rajs	3500
Davies	3100
Matos	2600
Vargas	2500

ORACLE

The BETWEENCondition

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

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

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

Using the IN Condition

Use the IN membership condition to test for values in a list.

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

EMPLOYEE_ID	LAST_NAME	SALARY	MANAGER_ID
202	Fay	6000	201
200	Whalen	4400	101
205	Higgins	12000	101
101	Kochhar	17000	100
102	De Haan	17000	100
124	Mourgos	5800	100
149	Zlotkey	10500	100
201	Hartstein	13000	100

8 rows selected.

The IN Condition

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

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

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

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

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

Using the LIKE Condition

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

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

ORACLE

The LIKE Condition

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

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

The SELECT statement on the slide returns the employee first name from the EMPLOYEES table for any employee whose first name begins with an *S*. Note the uppercase *S*. Names beginning with an *s* are not returned.

The LIKE condition can be used as a shortcut for some BETWEEN comparisons. The following example displays the last names and hire dates of all employees who joined between January 1995 and December 1995:

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

Using the LIKE Condition

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

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

LAST_NAME
Kochhar
Lorentz
Mourgos

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

ORACLE

Combining Wildcard Characters

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

The ESCAPE Option

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

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

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

Using the NULL Conditions

Test for nulls with the IS NULL operator.

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

LAST_NAME	MANAGER_ID
King	

ORACLE

The NULL Conditions

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

The IS NULL condition tests for nulls. A null value means the value is unavailable, unassigned, unknown, or inapplicable. Therefore, you cannot test with = because a null cannot be equal or unequal to any value.

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

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

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

16 rows selected.

Logical Conditions

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

Logical Conditions

A logical condition combines the result of two component conditions to produce a single result based on them or inverts the result of a single condition. A row is returned only if the overall result of the condition is true. Three logical operators are available in SQL:

AND

OR

NOT

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

Using the AND Operator

AND requires both conditions to be true.

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

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

ORACLE

The AND Operator

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

All character searches are case sensitive. No rows are returned if MAN is not in uppercase. Character strings must be enclosed in quotation marks.

AND Truth Table

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

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

Using the OR Operator

OR requires either condition to be true.

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

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

8 rows selected.

The OR Operator

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

The OR Truth Table

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

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

Using the NOT Operator

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

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

10 rows selected.

The NOT Operator

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

The NOT Truth Table

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

NOT	TRUE	FALSE	NULL
	FALSE	TRUE	NULL

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

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

Rules of Precedence

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

Override rules of precedence by using parentheses.

ORACLE

Rules of Precedence

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

Rules of Precedence

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

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

ORACLE

Example of the Precedence of the AND Operator

In the slide example, there are two conditions:

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

The second condition is that the job ID is SA_REP. Therefore, the SELECT statement reads as follows:

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

Rules of Precedence

Use parentheses to force priority.

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

LAST_NAME	JOB_ID	SALARY
King	AD_PRES	24000

Using Parentheses

In the example, there are two conditions:

The first condition is that the job ID is AD_PRES or SA_REP.

The second condition is that salary is greater than $15,000. Therefore, the SELECT statement reads as follows:

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

ORDER BY Clause

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

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

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

...

20 rows selected.

ORACLE

The ORDER BY Clause

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

Syntax

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

In the syntax:

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

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

Sorting in Descending Order

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

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

...

20 rows selected.

ORACLE

Default Ordering of Data

The default sort order is ascending:

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

Reversing the Default Order

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

Sorting by Column Alias

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

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

...

20 rows selected.

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.

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

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

...

20 rows selected.

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

ORACLE

Sorting by Multiple Columns

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

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

Example

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

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

Summary

In this lesson, you should have learned how to:

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

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

Summary

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

Assignments

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

LAST_NAME	SALARY
King	24000
Kochhar	17000
De Haan	17000
Hartstein	13000

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

LAST_NAME	DEPARTMENT_ID
Taylor	80

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

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

10 rows selected.

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

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

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

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

7 rows selected.

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

Employee	Monthly Salary
Mourgos	5800
Fay	6000

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

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

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

LAST_NAME	JOB_ID
King	AD_PRES

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

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

If you have time, complete the following exercises:

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

LAST_NAME
Grant
Whalen

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

LAST_NAME
De Haan
Davies
Whalen
Hartstein

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

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

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

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

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

Chapter 4: Single-Row Functions

Objectives

SQL Functions

Two Types of SQL Functions

Single-Row Functions

Single-Row Functions

Character Functions

Character Functions

Case Manipulation Functions

Using Case Manipulation Functions

Character-Manipulation Functions

Using the Character-Manipulation Functions

Number Functions

Using the ROUND Function

Using the TRUNC Function

Using the MOD Function

Working with Dates

Arithmetic with Dates

Using Arithmetic Operators with Dates

Date Functions

Using Date Functions

Conversion Functions

Implicit Data Type Conversion

Explicit Data Type Conversion

Using the TO_CHAR Function with Dates

Elements of the Date Format Model

Using the TO_CHAR Function with Dates

Using the TO_CHAR Function with Numbers

Using the TO_NUMBER and TO_DATE Functions

RR Date Format

Example of RR Date Format

Nesting Functions

General Functions

NVL Function

Using the NVL Function

Using the NVL2 Function

Using the NULLIF Function

Using the COALESCE Function

Conditional Expressions

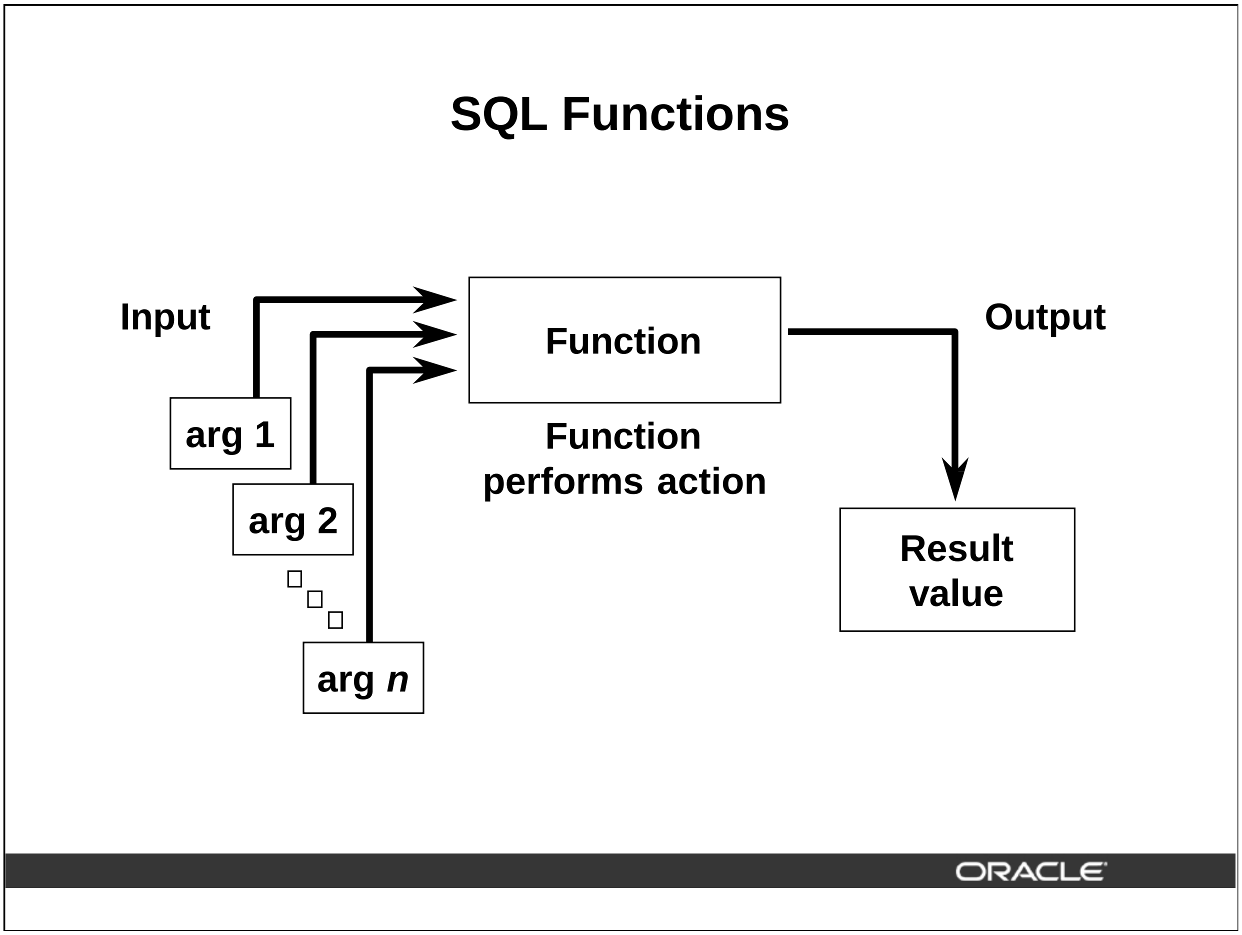
The CASE Expression

Using the CASE Expression

The DECODE Function

Using the DECODE Function

Summary



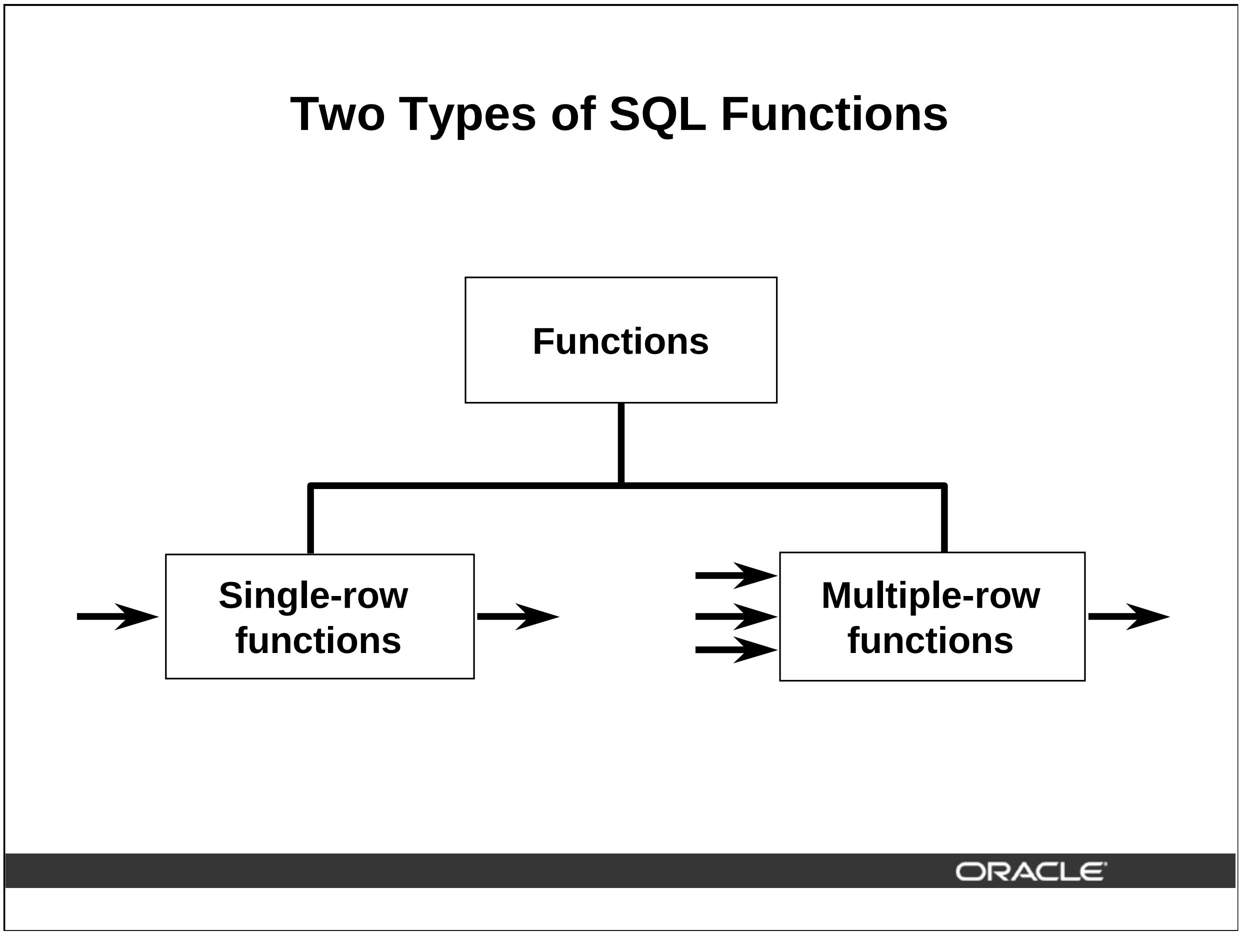
SQL Functions

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

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

SQL functions sometimes take arguments and always return a value.

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



SQL Functions (continued)

There are two distinct types of functions:

- Single-row functions
- Multiple-row functions

Single-Row Functions

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

- Character
- Number
- Date
- Conversion

Multiple-Row Functions

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

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

Single-Row Functions

Single row functions:

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

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

ORACLE

Single-Row Functions

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

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

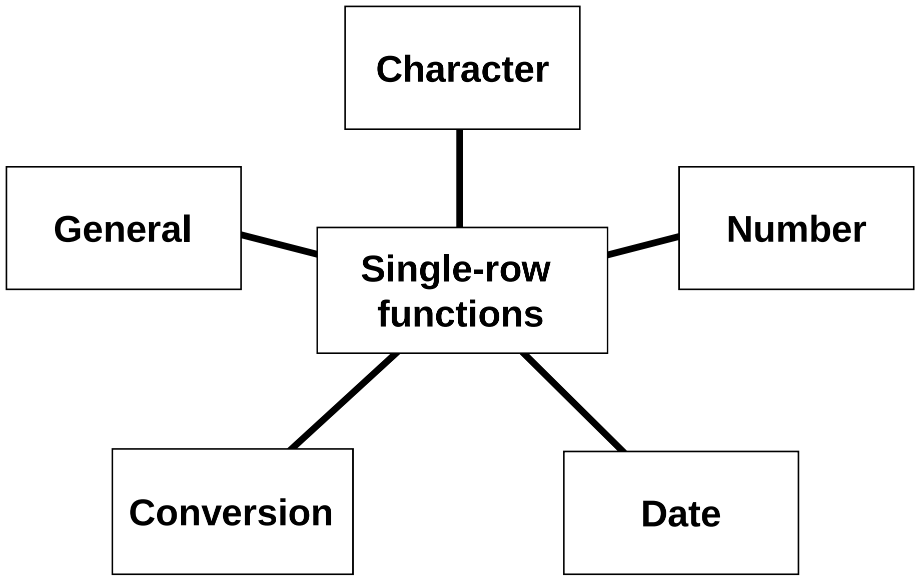
Features of single-row functions include:

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

In the syntax:

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

Single-Row Functions

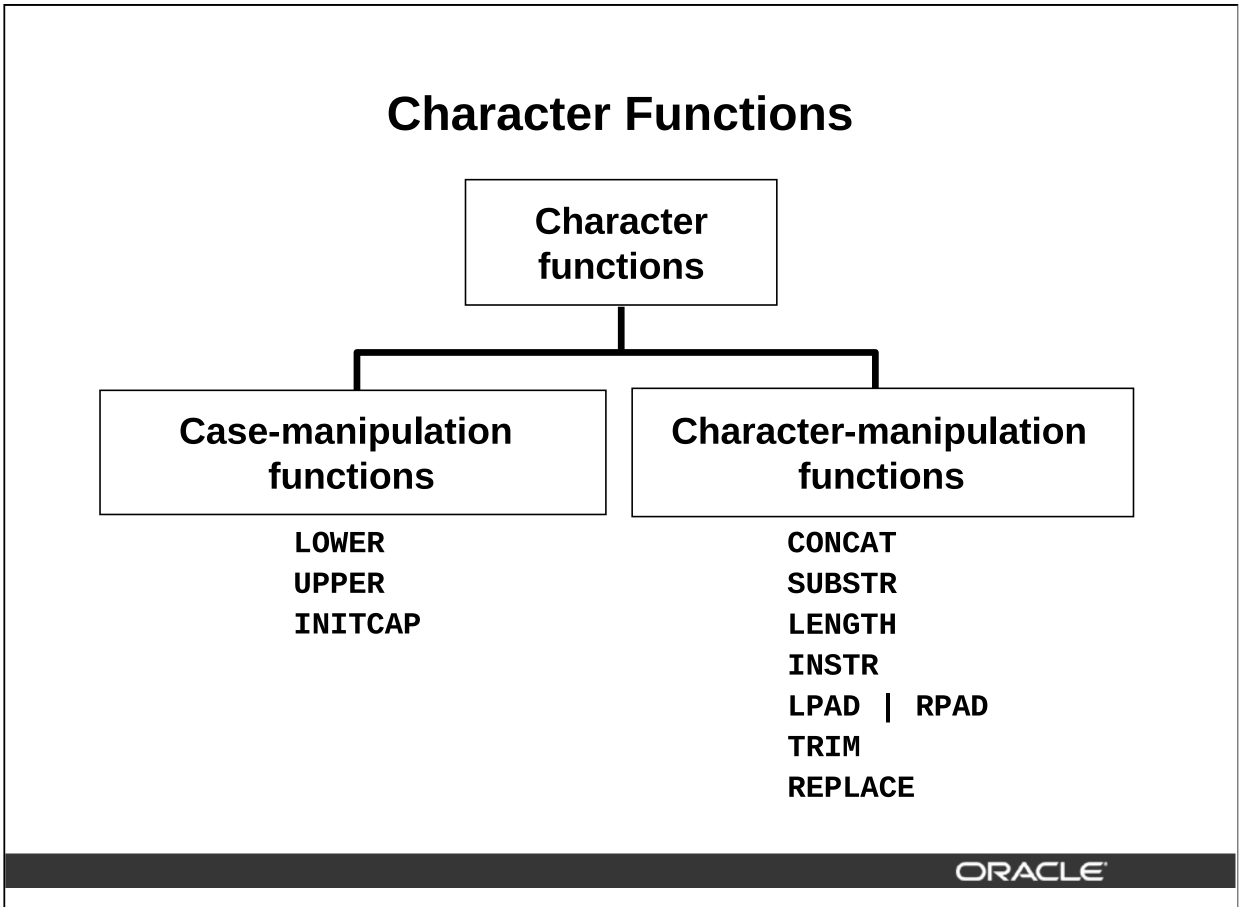


ORACLE

Single-Row Functions (continued)

This lesson covers the following single-row functions:

- Character functions: Accept character input and can return both character and number values
- Number functions: Accept numeric input and return numeric values
- Date functions: Operate on values of the `DATE` data type (All date functions return a value of `DATE` data type except the `MONTHS_BETWEEN` function, which returns a number.)
- Conversion functions: Convert a value from one data type to another
- General functions:
  - `NVL`
  - `NVL2`
  - `NULLIF`
  - `COALSECE`
  - `CASE`
  - `DECODE`



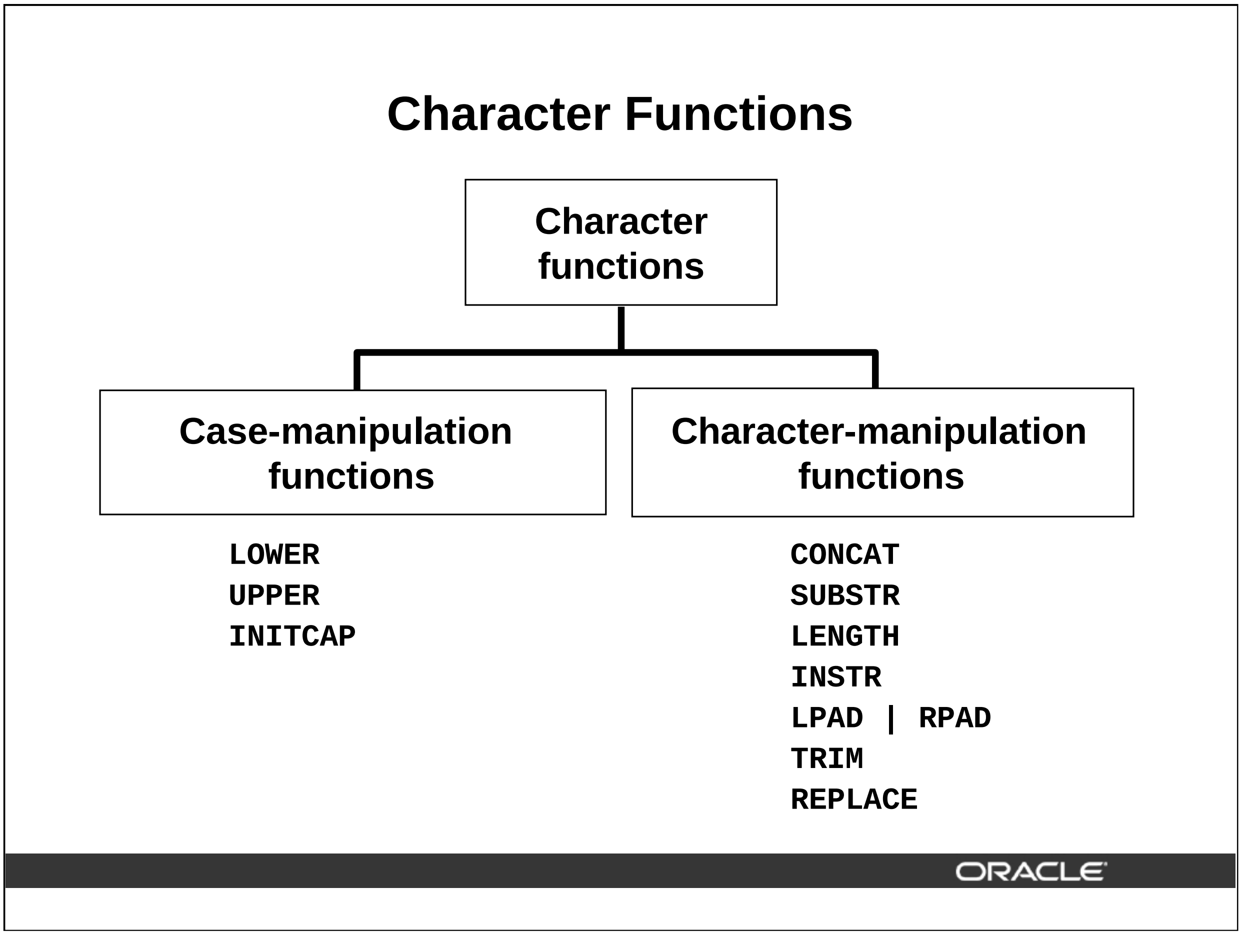
Character Functions

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

- Case-manipulation functions
- Character-manipulation functions

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

Note: The functions discussed in this lesson are only some of the available functions.



Character Functions (continued)

Function	Purpose
`LENGTH(`*`column`*`|`*`expression`*`)`	Returns the number of characters in the expression
`INSTR(`*`column`*`|`*`expression`*`, 'string', [,m], [n] )`	Returns the numeric position of a named string. Optionally, you can provide a position *m* to start searching, and the occurrence *n* of the string. *m* and *n* default to 1, meaning start the search at the beginning of the search and report the first occurrence.
`LPAD(`*`column`*`|`*`expression`*`, n, 'string')` `RPAD(`*`column`*`|`*`expression`*`, n, 'string')`	Pads the character value right-justified to a total width of *n* character positions Pads the character value left-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* and later.
`REPLACE(`*`text`*`, search_string, replacement_string)`	Searches a text expression for a character string and, if found, replaces it with a specified replacement string

Case Manipulation Functions

These functions convert case for character strings.

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

ORACLE

Case Manipulation Functions

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

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

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

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

20 rows selected.

Using Case Manipulation Functions

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

```
SELECT employee_id, last_name, department_id
FROM   employees
WHERE  last_name = 'higgins';
no rows selected
```

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

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
205	Higgins	110

ORACLE

Case Manipulation Functions (continued)

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

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

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

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

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

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

Character-Manipulation Functions

These functions manipulate character strings:

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

Character Manipulation Functions

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

- `CONCAT`: Joins values together (You are limited to using two parameters with `CONCAT.`)
- `SUBSTR`: Extracts a string of determined length
- `LENGTH`: Shows the length of a string as a numeric value
- `INSTR`: Finds numeric position of a named character
- `LPAD`: Pads the character value right-justified
- `RPAD`: Pads the character value left-justified
- `TRIM`: Trims heading or trailing characters (or both) from a character string (If *`trim_character`* or *`trim_source`* is a character literal, you must enclose it in single quotes.)

Using the Character-Manipulation Functions

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

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

ORACLE

Character-Manipulation Functions (continued)

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

Example

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

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

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

Number Functions

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

  `ROUND(45.926, 2)` → `45.93`

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

  `TRUNC(45.926, 2)` → `45.92`

- **`MOD`: Returns remainder of division**

  `MOD(1600, 300)` → `100`

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, then *n* defaults to zero
MOD(*m*,*n*)	Returns the remainder of *m* divided by *n*

Using the ROUND Function

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

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

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

ORACLE

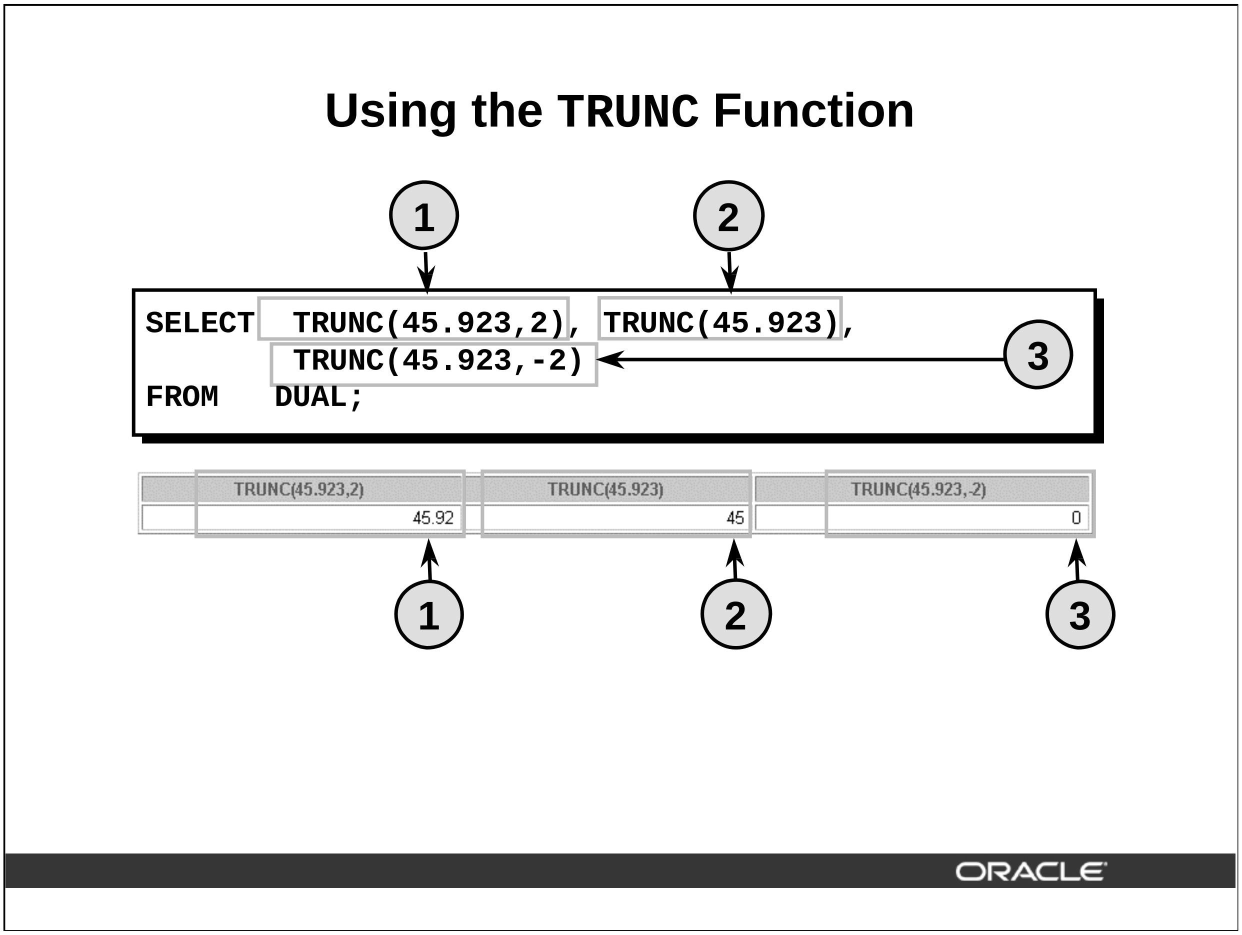
ROUND Function

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 Table

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



TRUNC Function

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

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

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

Using the MOD Function

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

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

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

ORACLE

MOD Function

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

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

Working with Dates

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

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

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

Oracle Date Format

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

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

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

This data is stored internally as follows:

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

Centuries and the Year 2000

The Oracle server is year 2000 compliant. When a record with a date column is inserted into a table, the *century* information is picked up from the SYSDATE function. However, when the date column is displayed on the screen, the century component is not displayed by default.

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

Working with Dates

SYSDATE is a function that returns:

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

The SYSDATE Function

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

Example

Display the current date using the DUAL table.

```
SELECT SYSDATE
FROM   DUAL;
```

SYSDATE
28-SEP-01

Arithmetic with Dates

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

ORACLE

Arithmetic with Dates

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

You can perform the following operations:

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

Using Arithmetic Operators with Dates

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

LAST_NAME	WEEKS
King	744.245395
Kochhar	626.102538
De Haan	453.245395

Arithmetic with Dates (continued)

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

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

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

Date Functions

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

ORACLE

Date Functions

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

- `MONTHS_BETWEEN(`*`date1, date2`*`)`: Finds the number of months between *`date1`* and *`date2`*. The result can be positive or negative. If *`date1`* is later than *`date2`*, the result is positive; if *`date1`* is earlier than *`date2`*, the result is negative. The noninteger part of the result represents a portion of the month.
- `ADD_MONTHS(`*`date, n`*`)`: Adds *`n`* number of calendar months to *`date`*. The value of *`n`* must be an integer and can be negative.
- `NEXT_DAY(`*`date,`*` '`*`char`*`')`: Finds the date of the next specified day of the week `('`*`char`*`')` following *`date`*. The value of *`char`* may be a number representing a day or a character string.
- `LAST_DAY(`*`date`*`)`: Finds the date of the last day of the month that contains *`date`*.
- `ROUND(`*`date`*`[,'`*`fmt`*`'])`: Returns *`date`* rounded to the unit specified by the format model *`fmt`*. If the format model *fmt* is omitted, *date* is rounded to the nearest day.
- `TRUNC(`*`date`*`[, '`*`fmt`*`'])`: Returns *`date`* with the time portion of the day truncated to the unit specified by the format model *`fmt`*. If the format model *`fmt`* is omitted, *`date`* is truncated to the nearest day.

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

Using Date Functions

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

ORACLE

Date Functions (continued)

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

```
SELECT employee_id, hire_date,
       MONTHS_BETWEEN (SYSDATE, hire_date) TENURE,
       ADD_MONTHS (hire_date, 6) REVIEW,
       NEXT_DAY (hire_date, 'FRIDAY'), LAST_DAY(hire_date)
FROM   employees
WHERE  MONTHS_BETWEEN (SYSDATE, hire_date) < 36;
```

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

Using Date Functions

Assume SYSDATE = ’25-JUL-95’:

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

ORACLE

Date Functions (continued)

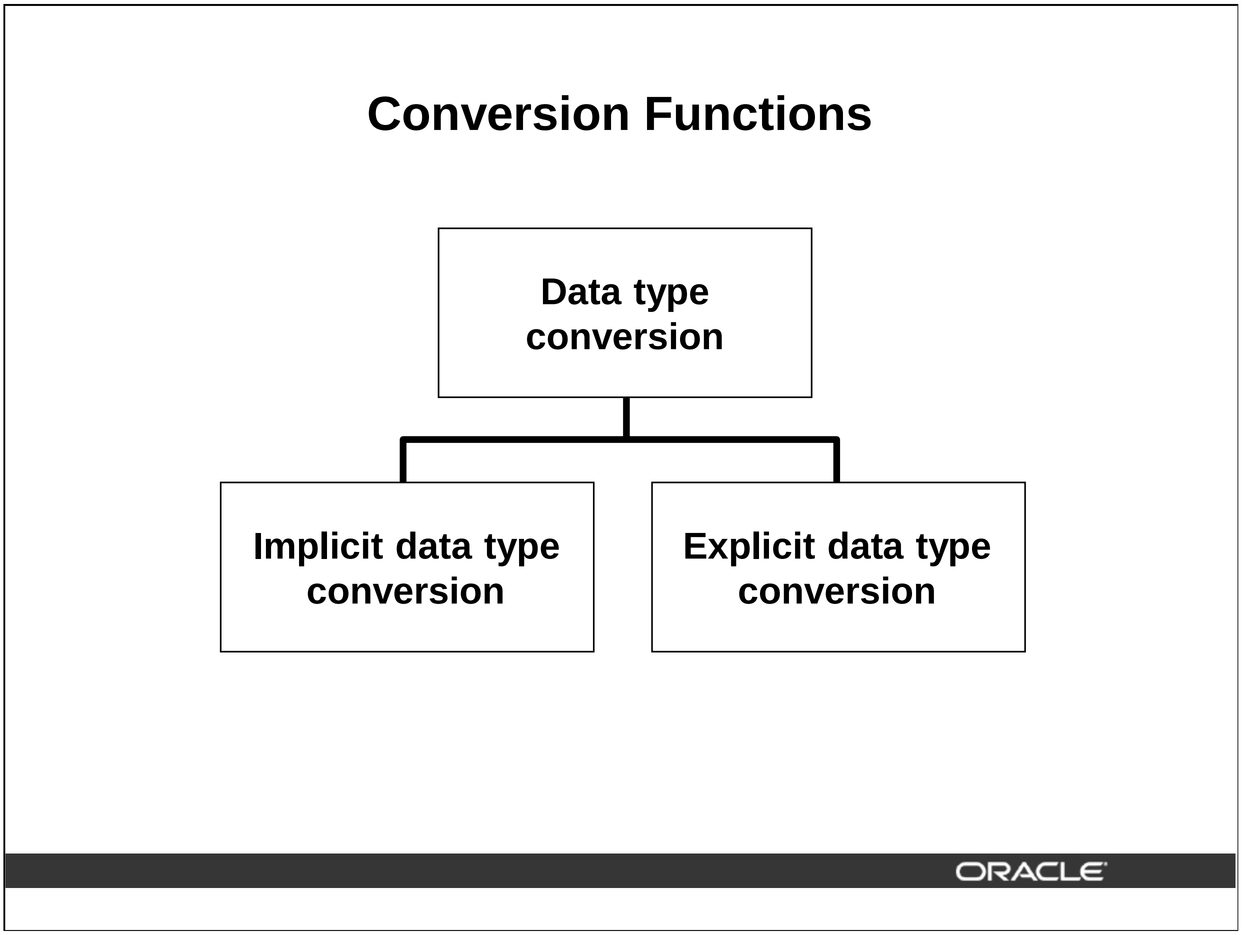
The ROUND and TRUNC functions can be used for number and date values. When used with dates, these functions round or truncate to the specified format model. Therefore, you can round dates to the nearest year or month.

Example

Compare the hire dates for all employees who started in 1997. Display the employee number, hire date, and start month using the ROUND and TRUNC functions.

```
SELECT employee_id, hire_date,
       ROUND(hire_date, ’MONTH’), TRUNC(hire_date, ’MONTH’)
FROM   employees
WHERE  hire_date LIKE ’%97’;
```

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



Conversion Functions

In addition to Oracle data types, columns of tables in an Oracle9*i* database can be defined using ANSI, DB2, and SQL/DS data types. However, the Oracle server internally converts such data types to Oracle data types.

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

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

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

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

Implicit Data Type Conversion

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

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

ORACLE

Implicit Data Type Conversion

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

Implicit Data Type Conversion

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

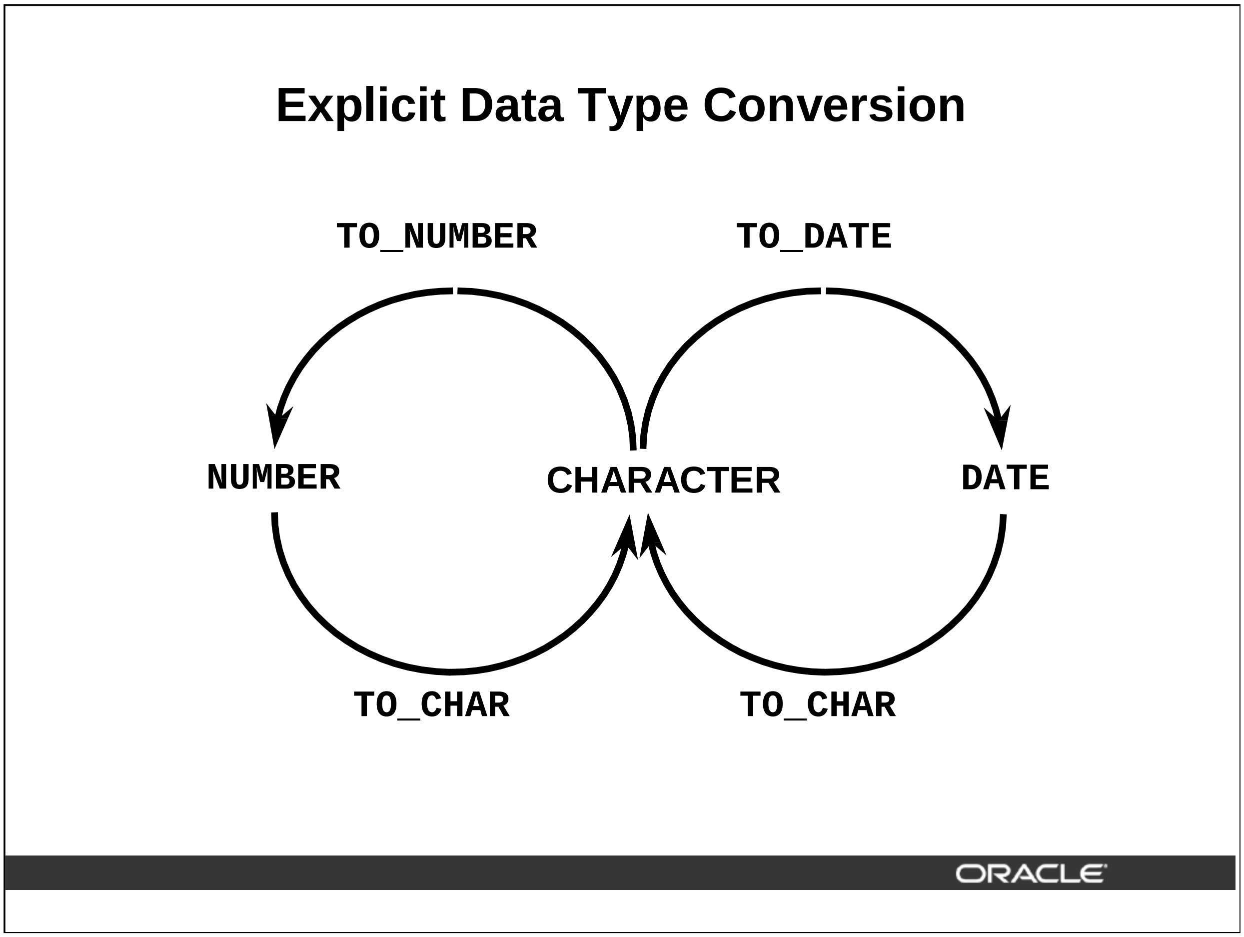
From	To
`VARCHAR2 or CHAR`	`NUMBER`
`VARCHAR2 or CHAR`	`DATE`

ORACLE

Implicit Data Type Conversion (continued)

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

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

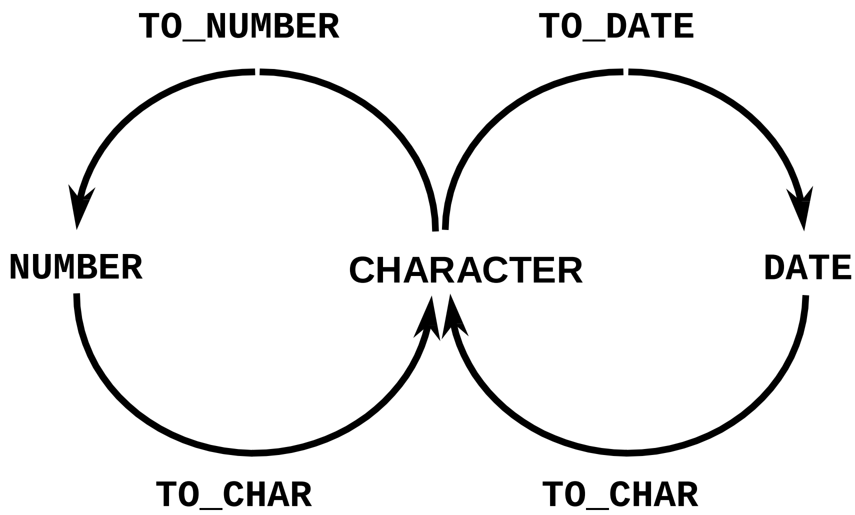


Explicit Data Type Conversion

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

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

Explicit Data Type Conversion



Explicit Data Type 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(*char*,[*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.

Using the TO_CHAR Function with Dates

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

The format model:

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

ORACLE

Displaying a Date in a Specific Format

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

Guidelines

- The format model must be enclosed in single quotation marks and is case sensitive.
- The format model can include any valid date format element. Be sure to separate the date value from the format model by a comma.
- The names of days and months in the output are automatically padded with blanks.
- To remove padded blanks or to suppress leading zeros, use the fill mode *`fm`* element.
- You can format the resulting character field with the *i*SQL*Plus `COLUMN` command covered in a later lesson.

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

EMPLOYEE_ID	MONTH
205	06/94

Elements of the Date Format Model

- **Time elements format the time portion of the date.**

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

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

`DD "of" MONTH`	`12 of OCTOBER`

- **Number suffixes spell out numbers.**

`ddspth`	`fourteenth`

ORACLE

Date Format Elements - Time Formats

Use the formats listed in the following tables to display time information and literals and to change numerals to spelled numbers.

Element	**Description**
AM or PM	Meridian indicator
A.M. or P.M.	Meridian indicator with periods
HH or HH12 or HH24	Hour of day, or hour (1–12), or hour (0–23)
MI	Minute (0–59)
SS	Second (0–59)
SSSSS	Seconds past midnight (0–86399)

Using the TO_CHAR Function with Dates

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

LAST_NAME	HIREDATE
King	17 June 1987
Kochhar	21 September 1989
De Haan	13 January 1993
Hunold	3 January 1990
Ernst	21 May 1991
Lorentz	7 February 1999
Mourgos	16 November 1999

...

20 rows selected.

ORACLE

The TO_CHAR Function with Dates

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

Example

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

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

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

20 rows selected.

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

Using the TO_CHAR Function with Numbers

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

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

9	Represents a number
10	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

The TO_CHAR Function with Numbers

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

Number Format Elements

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

Element	Description	Example	Result
9	Numeric position (number of 9s determine display width)	999999	1234
0	Display leading zeros	099999	001234
$	Floating dollar sign	$999999	$1234
L	Floating local currency symbol	L999999	FF1234
.	Decimal point in position specified	999999.99	1234.00
,	Comma in position specified	999,999	1,234
MI	Minus signs to right (negative values)	999999MI	1234-
PR	Parenthesize negative numbers	999999PR	<1234>
EEEE	Scientific notation (format must specify four Es)	99.999EEEE	1.234E+03
V	Multiply by 10 *n* times (*n* = number of 9s after V)	9999V99	123400
B	Display zero values as blank, not 0	B9999.99	1234.00

Using the TO_CHAR Function with Numbers

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

SALARY
$6,000.00

Guidelines

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

Using the TO_NUMBER and TO_DATE Functions

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

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

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

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

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

ORACLE

The TO_NUMBER and TO_DATE Functions

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

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

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

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

<table>
<tr><td colspan="2" rowspan="2"></td><td colspan="2">If the specified two-digit year is:</td></tr>
<tr><td>0–49</td><td>50–99</td></tr>
<tr><td rowspan="2">If two digits of the current year are:</td><td>0–49</td><td>The return date is in the current century</td><td>The return date is in the century before the current one</td></tr>
<tr><td>50–99</td><td>The return date is in the century after the current one</td><td>The return date is in the current century</td></tr>
</table>

The RR Date Format Element

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

Current Year	Given Date	Interpreted (RR)	Interpreted (YY)
1994	27-OCT-95	1995	1995
1994	27-OCT-17	2017	1917
2001	27-OCT-17	2017	2017

Example of RR Date Format

To find employees hired prior to 1990, use the RR format, which produces the same results whether the command is run in 1999 or now:

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

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

The RR Date Format Element Example

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

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

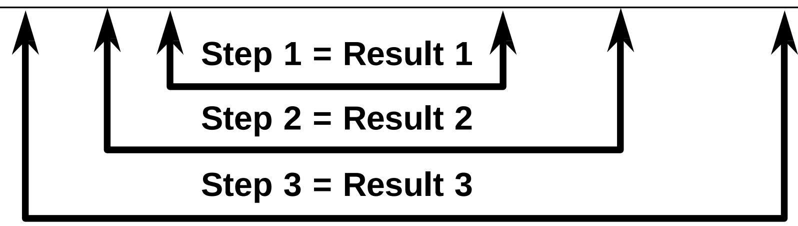
```
SELECT last_name, TO_CHAR(hire_date, 'DD-Mon-yyyy')
FROM   employees
WHERE  TO_DATE(hire_date, 'DD-Mon-yy') < '01-Jan-1990';

no rows selected
```

Nesting Functions

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

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



Nesting Functions

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

Nesting Functions

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

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

ORACLE

Nesting Functions (continued)

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

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

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

Example

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

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

General Functions

These functions work with any data type and pertain to using nulls.

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

ORACLE

General Functions

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

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

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

NVL Function

Converts a null to an actual value.

- **Data types that can be used are date, character, and number.**
- **Data types must match:**
  - **`NVL(commission_pct,0)`**
  - **`NVL(hire_date,'01-JAN-97')`**
  - **`NVL(job_id,'No Job Yet')`**

ORACLE

The `NVL` Function

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

Syntax

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

In the syntax:

`expr1` is the source value or expression that may contain a null

`expr2` is the target value for converting the null

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

`NVL` Conversions for Various Data Types

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

Using the NVL Function

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

LAST_NAME	SALARY	NVL(COMMISSION_PCT,0)	AN_SAL
King	24000	0	288000
Kochhar	17000	0	204000
De Haan	17000	0	204000
Hunold	9000	0	108000
Ernst	6000	0	72000
Lorentz	4200	0	50400
Mourgos	5800	0	69600
Rajs	3500	0	42000

...

20 rows selected.

1 2

The NVL Function

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

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

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

20 rows selected.

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

Using the NVL2 Function

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

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

8 rows selected.

The NVL2 Function

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

Syntax

NVL(*expr1*, *expr2*, *expr3*)

In the syntax:

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

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

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

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

Using the NULLIF Function

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

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

...

20 rows selected.

ORACLE

The NULLIF Function

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

Syntax

NULLIF (*expr1*, *expr2*)

In the syntax:

expr1 is the source value compared to *expr2*

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

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

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

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

Using the COALESCE Function

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

ORACLE

The COALESCE Function

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

Syntax

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

In the syntax:

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

Using the COALESCE Function

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

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

...

20 rows selected.

The COALESCEFunction

In the example shown, if the COMMISSION_PCTvalue is not null, it is shown. If the COMMISSION_PCT value is null, then the SALARY is shown. If the COMMISSION_PCTand SALARYvalues are null, then the value 10 is shown.

Conditional Expressions

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

ORACLE

Conditional Expressions

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

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

The CASE Expression

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

```
CASE expr WHEN comparison_expr1 THEN return_expr1
         [WHEN comparison_expr2 THEN return_expr2
          WHEN comparison_exprn THEN return_exprn
          ELSE else_expr]
END
```

The CASE Expression

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

In a simple CASE expression, Oracle searches for the first WHEN ... THEN pair for which expris equal to comparison_exprand returns return_expr. If none of the WHEN ... THEN pairs meet this condition, and an ELSE clause exists, then Oracle returns else_expr. Otherwise, Oracle returns null. You cannot specify the literal NULL for all the return_exprs and the else_expr.

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

Using the CASE Expression

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

```
SELECT last_name, job_id, salary,
       CASE job_id WHEN 'IT_PROG'  THEN  1.10*salary
                   WHEN 'ST_CLERK' THEN  1.15*salary
                   WHEN 'SA_REP'   THEN  1.20*salary
       ELSE      salary END     "REVISED_SALARY"
FROM   employees;
```

LAST_NAME	JOB_ID	SALARY	REVISED_SALARY
...			
Lorentz	IT_PROG	4200	4620
Mourgos	ST_MAN	5800	5800
Rajs	ST_CLERK	3500	4025
...			
Gietz	AC_ACCOUNT	8300	8300

20 rows selected.

ORACLE

Using the CASE Expression

In the preceding SQL statement, the value of JOB_ID is decoded. If JOB_ID is IT_PROG, the salary increase is 10%; if JOB_IDis ST_CLERK, the salary increase is 15%; if JOB_IDis SA_REP, the salary increase is 20%. For all other job roles, there is no increase in salary.

The same statement can be written with the DECODEfunction.

The DECODE Function

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

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

ORACLE

The DECODE Function

The DECODE function decodes an expression in a way similar to the IF-THEN-ELSE logic used in various languages. The DECODE function decodes expression after comparing it to each search value. If the expression is the same as search, result is returned.

If the default value is omitted, a null value is returned where a search value does not match any of the result values.

Using the DECODE Function

```
SELECT last_name, job_id, salary,
       DECODE(job_id, 'IT_PROG',  1.10*salary,
                      'ST_CLERK', 1.15*salary,
                      'SA_REP',   1.20*salary,
             salary)
       REVISED_SALARY
FROM   employees;
```

LAST_NAME	JOB_ID	SALARY	REVISED_SALARY
...			
Lorentz	IT_PROG	4200	4620
Mourgos	ST_MAN	5800	5800
Rajs	ST_CLERK	3500	4025
...			
Gietz	AC_ACCOUNT	8300	8300

20 rows selected.

ORACLE

Using the DECODE Function

In the preceding SQL statement, the value of JOB_ID is tested. If JOB_ID is IT_PROG, the salary increase is 10%; if JOB_ID is ST_CLERK, the salary increase is 15%; if JOB_ID is SA_REP, the salary increase is 20%. For all other job roles, there is no increase in salary.

The same statement can be expressed in pseudocode as an IF-THEN-ELSE statement:

```
IF job_id = 'IT_PROG'     THEN  salary = salary*1.10
IF job_id = 'ST_CLERK'    THEN  salary = salary*1.15
IF job_id = 'SA_REP'      THEN  salary = salary*1.20
ELSE salary = salary
```

Using the DECODE Function

Display the applicable tax rate for each employee in department 80.

```
SELECT last_name, salary,
       DECODE (TRUNC(salary/2000, 0),
                         0, 0.00,
                         1, 0.09,
                         2, 0.20,
                         3, 0.30,
                         4, 0.40,
                         5, 0.42,
                         6, 0.44,
                            0.45) TAX_RATE
FROM   employees
WHERE  department_id = 80;
```

Example

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

Monthly Salary Range	***Rate***
$0.00 - 1999.99	00%
$2,000.00 - 3,999.99	09%
$4,000.00 - 5,999.99	20%
$6,000.00 - 7,999.99	30%
$8,000.00 - 9,999.99	40%
$10,000.00 - 11,999.99	42%
$12,200.00 - 13,999.99	44%
$14,000.00 or greater	45%

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

Summary

In this lesson, you should have learned how to:

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

ORACLE

Single-Row Functions

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

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

SYSDATE and DUAL

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

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

Date
28-SEP-01

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

EMPLOYEE_ID	LAST_NAME	SALARY	New Salary
100	King	24000	27600
101	Kochhar	17000	19550
102	De Haan	17000	19550
103	Hunold	9000	10350

...

202	Fay	6000	6900
205	Higgins	12000	13800
206	Gietz	8300	9545

20 rows selected.

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

EMPLOYEE_ID	LAST_NAME	SALARY	New Salary	Increase
100	King	24000	27600	3600
101	Kochhar	17000	19550	2550
102	De Haan	17000	19550	2550
103	Hunold	9000	10350	1350
104	Ernst	6000	6900	900
107	Lorentz	4200	4830	630
124	Mourgos	5800	6670	870
141	Rajs	3500	4025	525
142	Davies	3100	3565	465
143	Matos	2600	2990	390

...

201	Hartstein	13000	14950	1950
202	Fay	6000	6900	900
205	Higgins	12000	13800	1800
206	Gietz	8300	9545	1245

20 rows selected.

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

Name	Length
Abel	4
Matos	5
Mourgos	7

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

   Note: Your results will differ.

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

20 rows selected.

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

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

20 rows selected.

If you have time, complete the following exercises:

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

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

20 rows selected.

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

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

20 rows selected.

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

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

20 rows selected.

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

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

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

20 rows selected.

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

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

20 rows selected.

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

Job	***Grade***
AD_PRES	A
ST_MAN	B
IT_PROG	C
SA_REP	D
ST_CLERK	E
None of the above	0

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

20 rows selected.

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

Chapter 5: Displaying Data from Multiple Tables

Obtaining Data from Multiple Tables

Cartesian Products

Generating a Cartesian Product

Types of Joins

Joining Tables Using Oracle Syntax

What is an Equijoin?

Retrieving Records with Equijoins

Additional Search Conditions Using the AND Operator

Qualifying Ambiguous Column Names

Using Table Aliases

Joining More than Two Tables

Non-Equijoins

Retrieving Records with Non-Equijoins

Outer Joins

Outer Joins Syntax

Using Outer Joins

Self Joins

Joining a Table to Itself

Joining Tables Using SQL: 1999 Syntax

Creating Cross Joins

Creating Natural Joins

Retrieving Records with Natural Joins

Creating Joins with the USINGClause

Retrieving Records with the USINGClause

Creating Joins with the ONClause

Retrieving Records with the ONClause

Creating Three-Way Joins with the ONClause

INNERVersus OUTERJoins

Obtaining Data from Multiple Tables

EMPLOYEES

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

DEPARTMENTS

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

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

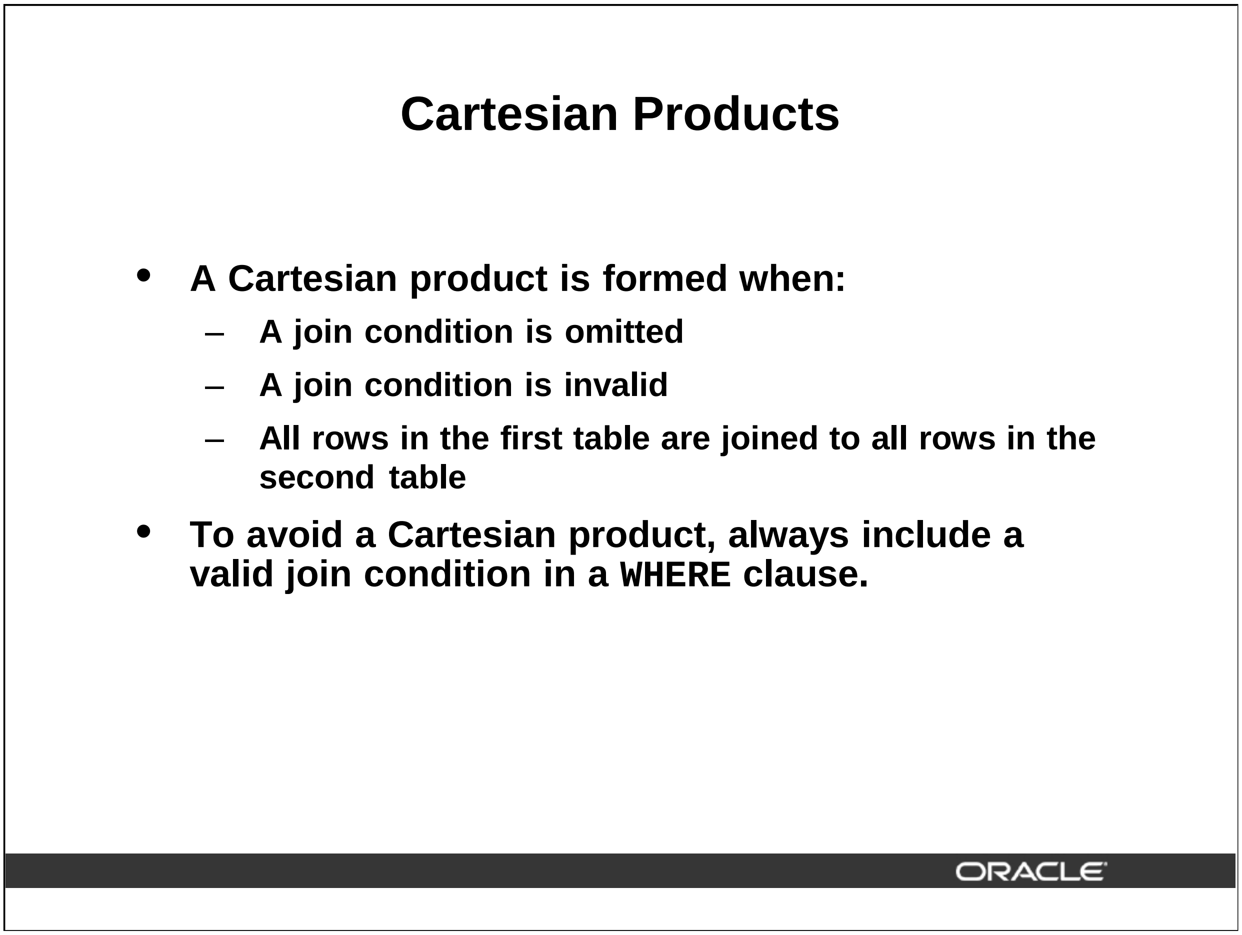
4-2

Data from Multiple Tables

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

- Employee IDs exist in the EMPLOYEES table.
- Department IDs exist in both the EMPLOYEESand DEPARTMENTStables.
- Location IDs exist in the DEPARTMENTStable.

To produce the report, you need to link the EMPLOYEESand DEPARTMENTStables and access data from both of them.



Cartesian Products

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

A Cartesian product tends to generate a large number of rows, and the result is rarely useful. You should always include a valid join condition in a WHERE clause, unless you have a specific need to combine all rows from all tables.

Cartesian products are useful for some tests when you need to generate a large number of rows to simulate a reasonable amount of data.

Generating a Cartesian Product

EMPLOYEES (20 rows)

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

20 rows selected.

DEPARTMENTS (8 rows)

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

8 rows selected.

Cartesian product: 20x8=160 rows

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

…

160 rows selected.

ORACLE

Cartesian Products (continued)

A Cartesian product is generated if a join condition is omitted. The example on the slide displays employee last name and department name from the `EMPLOYEES` and `DEPARTMENTS` tables. Because no `WHERE` clause has been specified, all rows (20 rows) from the `EMPLOYEES` table are joined with all rows (8 rows) in the `DEPARTMENTS` table, thereby generating 160 rows in the output.

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

LAST_NAME	DEPT_NAME
King	Administration
Kochhar	Administration
De Haan	Administration

…

160 rows selected.

Instructor Note

Demo: `4_cart.sql`

Purpose: To illustrate executing a Cartesian product

Types of Joins

Oracle Proprietary Joins (8*i* and prior):

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

SQL: 1999 Compliant Joins:

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

Types of Joins

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

Joining Tables Using Oracle Syntax

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

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

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

ORACLE

Defining Joins

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

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

In the syntax:

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

Guidelines

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

For more information, see *Oracle9i SQL Reference,* "`SELECT`."

What is an Equijoin?

EMPLOYEES

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

…

DEPARTMENTS

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

…

Foreign key **Primary key**

ORACLE

Equijoins

To determine an employee's department name, you compare the value in the `DEPARTMENT_ID` column in the `EMPLOYEES` table with the `DEPARTMENT_ID` values in the `DEPARTMENTS` table. The relationship between the `EMPLOYEES` and `DEPARTMENTS` tables is an *equijoin*—that is, values in the `DEPARTMENT_ID` column on both tables must be equal. Frequently, this type of join involves primary and foreign key complements.

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

Instructor Note

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

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

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

Retrieving Records with Equijoins

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

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

...

19 rows selected.

ORACLE

Retrieving Records with Equijoins

In the slide example:

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

  `EMPLOYEES.DEPARTMENT_ID = DEPARTMENTS.DEPARTMENT_ID`

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

Additional Search Conditions Using the AND Operator

EMPLOYEES

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

...

DEPARTMENTS

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

...

ORACLE

Additional Search Conditions

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

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

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
Matos	50	Shipping

Qualifying Ambiguous Column Names

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

ORACLE

Qualifying Ambiguous Column Names

You need to qualify the names of the columns in the WHERE clause with the table name to avoid ambiguity. Without the table prefixes, the DEPARTMENT_ID column could be from either the DEPARTMENTS table or the EMPLOYEES table. It is necessary to add the table prefix to execute your query.

If there are no common column names between the two tables, there is no need to qualify the columns. However, using the table prefix improves performance, because you tell the Oracle Server exactly where to find the columns.

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

Using Table Aliases

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

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

ORACLE

Table Aliases

Qualifying column names with table names can be very time consuming, particularly if table names are lengthy. You can use *table aliases* instead of table names. Just as a column alias gives a column another name, a table alias gives a table another name. Table aliases help to keep SQL code smaller, therefore using less memory.

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

Guidelines

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

Joining More than Two Tables

EMPLOYEES

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

...

20 rows selected.

DEPARTMENTS

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

8 rows selected.

LOCATIONS

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

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

ORACLE

Additional Search Conditions

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

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

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

...

19 rows selected.

Non-Equijoins

EMPLOYEES

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

…

20 rows selected.

JOB_GRADES

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

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

ORACLE

Non-Equijoins

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

Retrieving Records with Non-Equijoins

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

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

...

20 rows selected.

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 job grade table contain grades that overlap. That is, the salary value for an employee can lie only between the low salary and high salary values of one of the rows in the salary grade table.
- All of the employees' salaries lie within the limits provided by the job grade table. That is, no employee earns less than the lowest value contained in the `LOWEST_SAL` column or more than the highest value contained in the `HIGHEST_SAL` column.

Note: Other conditions, such as `<=` and `>=` can be used, but `BETWEEN` is the simplest. Remember to specify the low value first and the high value last when using `BETWEEN`.

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

Outer Joins

DEPARTMENTS

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

8 rows selected.

EMPLOYEES

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

...

20 rows selected.

There are no employees in department 190.

ORACLE

Returning Records with No Direct Match with Outer Joins

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

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

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

...

19 rows selected.

Outer Joins Syntax

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

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

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

ORACLE

Using Outer Joins to Return Records with No Direct Match

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

In the syntax:

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

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

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

20 rows selected.

ORACLE

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

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

Outer Join Restrictions

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

Self Joins

EMPLOYEES (WORKER)

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

...

EMPLOYEES (MANAGER)

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

...

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

ORACLE

Joining a Table to Itself

Sometimes you need to join a table to itself. To find the name of each employee's manager, you need to join the `EMPLOYEES` table to itself, or perform a self join. For example, to find the name of Whalen's manager, you need to:

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

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

Joining a Table to Itself

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

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

...

19 rows selected.

ORACLE

Joining a Table to Itself (continued)

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

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

Joining Tables Using SQL: 1999 Syntax

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

```
SELECT   table1.column, table2.column
FROM     table1
[CROSS JOIN table2] |
[NATURAL JOIN table2] |
[JOIN table2 USING (column_name)] |
[JOIN table2
  ON(table1.column_name = table2.column_name)] |
[LEFT|RIGHT|FULL OUTER JOIN table2
  ON (table1.column_name = table2.column_name)];
```

ORACLE

Defining Joins

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

In the syntax:

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

Creating Cross Joins

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

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

LAST_NAME	DEPARTMENT_NAME
King	Administration
Kochhar	Administration
De Haan	Administration
Hunold	Administration

...

160 rows selected.

Creating Cross Joins

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

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

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

...

160 rows selected.

Creating Natural Joins

- The `NATURAL JOIN` clause is based on all columns in the two tables that have the same name.
- It selects rows from the two tables that have equal values in all matched columns.
- If the columns having the same names have different data types, an error is returned.

ORACLE

Creating Natural Joins

It was not possible to do a join without explicitly specifying the columns in the corresponding tables in prior releases of Oracle. In Oracle9*i* it is possible to let the join be completed automatically based on columns in the two tables which have matching data types and names, using the keywords NATURAL JOINkeywords.

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

Retrieving Records with Natural Joins

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

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

8 rows selected.

ORACLE

Retrieving Records with Natural Joins

In the example on the slide, the `LOCATIONS` table is joined to the `DEPARTMENT` table by the `LOCATION_ID` column, which is the only column of the same name in both tables. If other common columns were present, the join would have used them all.

Equijoins

The natural join can also be written as an equijoin:

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

Natural Joins with a `WHERE` Clause

Additional restrictions on a natural join are implemented by using a `WHERE` clause. The example below limits the rows of output to those with a department ID equal to 20 or 50.

```
SELECT  department_id, department_name,
        location_id, city
FROM    departments
NATURAL JOIN locations
WHERE   department_id IN (20, 50);
```

Creating Joins with the `USING` Clause

- If several columns have the same names but the data types do not match, the `NATURAL JOIN` clause can be modified with the `USING` clause to specify the columns that should be used for an equijoin.
- Use the `USING` clause to match only one column when more than one column matches.
- Do not use a table name or alias in the referenced columns.
- The `NATURAL JOIN` and `USING` clauses are mutually exclusive.

ORACLE

The USING Clause

Natural joins use all columns with matching names and data types to join the tables. The `USING` clause can be used to specify only those columns that should be used for an equijoin. The columns referenced in the `USING` clause should not have a qualifier (table name or alias) anywhere in the SQL statement.

For example, this statement is valid:

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

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

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

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

Retrieving Records with the USING Clause

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

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

...

19 rows selected.

ORACLE

The USING Clause (continued)

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

This can also be written as an equijoin:

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

Retrieving Records with the ON Clause

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

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

...

19 rows selected.

ORACLE

Creating Joins with the ON Clause

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

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

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

...

19 rows selected.

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

Creating Three-Way Joins with the `ON` Clause

```
SELECT employee_id, city, department_name
FROM   employees e
JOIN   departments d
ON     d.department_id = e.department_id
JOIN   locations l
ON     d.location_id = l.location_id;
```

EMPLOYEE_ID	CITY	DEPARTMENT_NAME
103	Southlake	IT
104	Southlake	IT
107	Southlake	IT
124	South San Francisco	Shipping
141	South San Francisco	Shipping
142	South San Francisco	Shipping
143	South San Francisco	Shipping
144	South San Francisco	Shipping

...

19 rows selected.

ORACLE

Three-Way Joins

A three-way join is a join of three tables. In SQL: 1999 compliant syntax, joins are performed from left to right so the first join to be performed is `EMPLOYEES JOIN DEPARTMENTS`. The first join condition can reference columns in `EMPLOYEES` and `DEPARTMENTS` but cannot reference columns in `LOCATIONS`. The second join condition can reference columns from all three tables.

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

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

Instructor Note

The example shown can also be accomplished with the `USING` clause:

```
SELECT e.employee_id, l.city, d.department_name
FROM  employees e
JOIN departments d
USING (department_id)
JOIN locations l
USING (location_id);
```

INNER Versus OUTER Joins

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

ORACLE

Joins - Comparing SQL: 1999 to Oracle Syntax

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

LEFT OUTER JOIN

```
SELECT e.last_name, e.department_id, d.department_name
FROM   employees e
LEFT OUTER JOIN departments d
ON   (e.department_id = d.department_id) ;
```

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

20 rows selected.

ORACLE

Example of LEFT OUTER JOIN

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

This query was completed in earlier releases as follows:

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

RIGHT OUTER JOIN

```
SELECT e.last_name, e.department_id, d.department_name
FROM   employees e
RIGHT OUTER JOIN departments d
ON    (e.department_id = d.department_id) ;
```

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
King	90	Executive
Kochhar	90	Executive
...		
Whalen	10	Administration
Hartstein	20	Marketing
Fay	20	Marketing
Higgins	110	Accounting
Gietz	110	Accounting
		Contracting

20 rows selected.

ORACLE

Example of RIGHT OUTER JOIN

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

This query was completed in earlier releases as follows:

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

FULL OUTER JOIN

```
SELECT e.last_name, e.department_id, d.department_name
FROM   employees e
FULL OUTER JOIN departments d
ON   (e.department_id = d.department_id) ;
```

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
Whalen	10	Administration
Fay	20	Marketing

...

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
De Haan	90	Executive
Kochhar	90	Executive
King	90	Executive
Gietz	110	Accounting
Higgins	110	Accounting
Grant		
		Contracting

21 rows selected.

ORACLE

Example of `FULL OUTER JOIN`

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

Additional Conditions

```
SELECT e.employee_id, e.last_name, e.department_id,
       d.department_id, d.location_id
FROM   employees e JOIN departments d
ON     (e.department_id = d.department_id)
AND    e.manager_id = 149 ;
```

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

Applying Additional Conditions

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

Summary

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

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

ORACLE

Summary

There are multiple ways to join tables.

Types of Joins

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

Cartesian Products

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

Table Aliases

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

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

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

▪ ▪ ▪

19 rows selected.

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

JOB_ID	LOCATION_ID
SA_MAN	2500
SA_REP	2500

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

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

Practice 4 - Part One (continued)

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

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

10 rows selected.

Practice 4 - Part Two

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

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

6. Display the employee last name and employee number along with their manager's last name and manager number. Label the columns `Employee`, `Emp#`, `Manager`, and `Mgr#`, respectively.
   Place your SQL statement in a text file named `lab4_6.sql`.

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

19 rows selected.

Practice 4 - Part Two (continued)

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

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

▪ ▪ ▪

20 rows selected.

If you have time, complete the following exercises:

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

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

...

42 rows selected.

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

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

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

...

19 rows selected.

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

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

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

8 rows selected.

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

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

9 rows selected.

Chapter 6: Aggregating Data Using Group Functions

What Are Group Functions?

Types of Group Functions

Group Functions Syntax

Using the AVG and SUM Functions

Using the MIN and MAX Functions

Using the COUNT Function

Using the DISTINCT Keyword

Group Functions and Null Values

Using the NVL Function with Group Functions

Creating Groups of Data

Creating Groups of Data: The GROUP BY Clause Syntax

Using the GROUP BY Clause

Grouping by More Than One Column

Using the GROUP BY Clause on Multiple Columns

Illegal Queries Using Group Functions

Excluding Group Results

Excluding Group Results: The HAVING Clause Using the HAVING Clause

Nesting Group Functions

Summary

What Are Group Functions?

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

EMPLOYEES

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

...

20 rows selected.

The maximum salary in the EMPLOYEES table.

MAX(SALARY)
24000

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**

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

Group Functions Syntax

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

ORACLE

Guidelines for Using Group Functions

- `DISTINCT` makes the function consider only nonduplicate values; `ALL` makes it consider every value including duplicates. The default is `ALL` and therefore does not need to be specified.
- The data types for the functions with an `expr` argument may be `CHAR`, `VARCHAR2`, `NUMBER`, or `DATE`.
- All group functions ignore null values. To substitute a value for null values, use the `NVL`, `NVL2`, or `COALESCE` functions.
- The Oracle server implicitly sorts the result set in ascending order when using a `GROUP BY` clause. To override this default ordering, `DESC` can be used in an `ORDER BY` clause.

Using the AVG and SUM Functions

You can use AVG and SUM for numeric data.

```
SELECT AVG(salary), MAX(salary),
       MIN(salary), SUM(salary)
FROM   employees
WHERE  job_id LIKE '%REP%';
```

AVG(SALARY)	MAX(SALARY)	MIN(SALARY)	SUM(SALARY)
8150	11000	6000	32600

ORACLE

Group Functions

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

Using the MIN and MAX Functions

You can use MIN and MAX for any data type.

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

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

ORACLE

Group Functions (continued)

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

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

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

MIN(LAST_NAME)	MAX(LAST_NAME)
Abel	Zlotkey

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

Using the COUNT Function

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

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

COUNT(*)
5

ORACLE

The COUNT Function

The COUNT function has three formats:

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

COUNT(*) returns the number of rows in a table that satisfy the criteria of the SELECT statement, including duplicate rows and rows containing null values in any of the columns. If a WHERE clause is included in the SELECT statement, COUNT(*) returns the number of rows that satisfies the condition in the WHERE clause.

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

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

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

Using the COUNT Function

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

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

COUNT(COMMISSION_PCT)
3

ORACLE

The COUNT Function (continued)

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

Example

Display the number of department values in the EMPLOYEES table.

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

COUNT(DEPARTMENT_ID)
19

Using the DISTINCT Keyword

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

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

COUNT(DISTINCTDEPARTMENT_ID)
7

ORACLE

The `DISTINCT` Keyword

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

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

Group Functions and Null Values

Group functions ignore null values in the column.

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

AVG(COMMISSION_PCT)
.2125

ORACLE

Group Functions and Null Values

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

Using the NVL Function with Group Functions

The NVL function forces group functions to include null values.

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

AVG(NVL(COMMISSION_PCT,0))
.0425

Group Functions and Null Values (continued)

The NVL function forces group functions to include null values. In the slide example, the average is calculated based on all rows in the table, regardless of whether null values are stored in the COMMISSION_PCT column. The average is calculated as the total commission paid to all employees divided by the total number of employees in the company (20).

Creating Groups of Data

EMPLOYEES

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

...

20 rows selected.

4400

9500

3500

6400

10033

The average salary in EMPLOYEES table for each department.

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

ORACLE

Groups of Data

Until now, all group functions have treated the table as one large group of information. At times, you need to divide the table of information into smaller groups. This can be done by using the GROUP BY clause.

Creating Groups of Data: The GROUP BY Clause Syntax

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

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

ORACLE

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 receive an error message if you fail to include the column list in the GROUP BY clause.
- Using a WHERE clause, you can exclude rows before dividing them into groups.
- You must include the *columns* in the GROUP BY clause.
- You cannot use a column alias in the GROUP BY clause.
- By default, rows are sorted by ascending order of the columns included in the GROUP BY list. You can override this by using the ORDER BY clause.

Using the GROUP BY Clause

All columns in the SELECT list that are not in group functions must be in the GROUP BY clause.

```
SELECT   department_id, AVG(salary)
FROM     employees
GROUP BY department_id ;
```

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

8 rows selected.

ORACLE

The GROUP BY Clause (continued)

When using the GROUP BY clause, make sure that all columns in the SELECT list that are not group functions are included in the GROUP BY clause. The example on the slide displays the department number and the average salary for each department. Here is how this SELECT statement, containing a GROUP BY clause, is evaluated:

- The SELECT clause specifies the columns to be retrieved:
  - Department number column in the EMPLOYEES table
  - The average of all the salaries in the group you specified in the GROUP BY clause
- The FROM clause specifies the tables that the database must access: the EMPLOYEES table.
- The WHERE clause specifies the rows to be retrieved. Since there is no WHERE clause, all rows are retrieved by default.
- The GROUP BY clause specifies how the rows should be grouped. The rows are being grouped by department number, so the AVG function that is being applied to the salary column will calculate the *average salary for each department.*

Instructor Note

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

Using the GROUP BY Clause

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

```
SELECT   AVG(salary)
FROM     employees
GROUP BY department_id ;
```

AVG(SALARY)
4400
9500
3500
6400
10033.3333
19333.3333
10150
7000

ORACLE

The GROUP BY Clause (continued)

The GROUP BY column does not have to be in the SELECT clause. For example, the SELECT statement on the slide displays the average salaries for each department without displaying the respective department numbers. Without the department numbers, however, the results do not look meaningful.

You can use the group function in the ORDER BY clause.

```
SELECT   department_id, AVG(salary)
FROM     employees
GROUP BY department_id
```

DEPARTMENT_ID	AVG(SALARY)
50	3500
10	4400
60	6400
...	
90	19333.3333

8 rows selected.

.

Grouping by More Than One Column

EMPLOYEES

DEPARTMENT_ID	JOB_ID	SALARY
90	AD_PRES	24000
90	AD_VP	17000
90	AD_VP	17000
60	IT_PROG	9000
60	IT_PROG	6000
60	IT_PROG	4200
50	ST_MAN	5800
50	ST_CLERK	3500
50	ST_CLERK	3100
50	ST_CLERK	2600
50	ST_CLERK	2500
80	SA_MAN	10500
80	SA_REP	11000
80	SA_REP	8600
...		
20	MK_REP	6000
110	AC_MGR	12000
110	AC_ACCOUNT	8300

20 rows selected.

"Add up the salaries in the EMPLOYEES table for each job, grouped by department.

DEPARTMENT_ID	JOB_ID	SUM(SALARY)
10	AD_ASST	4400
20	MK_MAN	13000
20	MK_REP	6000
50	ST_CLERK	11700
50	ST_MAN	5800
60	IT_PROG	19200
80	SA_MAN	10500
80	SA_REP	19600
90	AD_PRES	24000
90	AD_VP	34000
110	AC_ACCOUNT	8300
110	AC_MGR	12000
	SA_REP	7000

13 rows selected.

ORACLE

Groups within Groups

Sometimes you need to see results for groups within groups. The slide shows a report that displays the total salary being paid to each job title, within each department.

The EMPLOYEES table is grouped first by department number and, within that grouping, by job title. For example, the four stock clerks in department 50 are grouped together and a single result (total salary) is produced for all stock clerks within the group.

Using the GROUP BY Clause on Multiple Columns

```
SELECT   department_id dept_id, job_id, SUM(salary)
FROM     employees
GROUP BY department_id, job_id ;
```

DEPT_ID	JOB_ID	SUM(SALARY)
10	AD_ASST	4400
20	MK_MAN	13000
20	MK_REP	6000
50	ST_CLERK	11700
50	ST_MAN	5800
60	IT_PROG	19200
80	SA_MAN	10500
80	SA_REP	19600
90	AD_PRES	24000
90	AD_VP	34000
110	AC_ACCOUNT	8300
110	AC_MGR	12000
	SA_REP	7000

13 rows selected.

ORACLE

Groups within Groups (continued)

You can return summary results for groups and subgroups by listing more than one GROUP BY column. You can determine the default sort order of the results by the order of the columns in the GROUP BY clause. Here is how the SELECT statement on the slide, containing a GROUP BY clause, is evaluated:

- The SELECT clause specifies the column to be retrieved:
  - Department number in the EMPLOYEES table
  - Job ID in the EMPLOYEES table
  - The sum of all the salaries in the group that you specified in the GROUP BY clause
- The FROM clause specifies the tables that the database must access: the EMPLOYEES table.
- The GROUP BY clause specifies how you must group the rows:
  - First, the rows are grouped by department number.
  - Second, within the department number groups, the rows are grouped by job ID.

So the SUM function is being applied to the salary column for all job IDs within each department number group.

Illegal Queries Using Group Functions

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

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

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

Column missing in the `GROUP BY` clause

ORACLE

Illegal Queries Using Group Functions

Whenever you use a mixture of individual items (DEPARTMENT_ID) and group functions (COUNT) in the same SELECT statement, you must include a GROUP BY clause that specifies the individual items (in this case, DEPARTMENT_ID). If the GROUP BY clause is missing, then the error message "not a single-group group function" appears and an asterisk (*) points to the offending column. You can correct the error on the slide by adding the GROUP BY clause.

```
SELECT   department_id, count(last_name)
FROM     employees
GROUP BY department_id;
```

DEPARTMENT_ID	COUNT(LAST_NAME)
10	1
20	2
...	
	1

8 rows selected.

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

Illegal Queries Using Group Functions

- You cannot use the WHERE clause to restrict groups.
- You use the HAVING clause to restrict groups.
- You cannot use group functions in the WHERE clause.

```
SELECT   department_id, AVG(salary)
FROM     employees
WHERE    AVG(salary) > 8000
GROUP BY department_id;
```

```
WHERE  AVG(salary) > 8000
       *
ERROR at line 3:
ORA-00934: group function is not allowed here
```

Cannot use the WHERE clause to restrict groups

ORACLE

Illegal Queries Using Group Functions (continued)

The WHERE clause cannot be used to restrict groups. The SELECT statement on the slide results in an error because it uses the WHERE clause to restrict the display of average salaries of those departments that have an average salary greater than $8,000.

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

```
SELECT   department_id, AVG(salary)
FROM     employees
HAVING   AVG(salary) > 8000
GROUP BY department_id;
```

DEPARTMENT_ID	AVG(SALARY)
20	9500
80	10033.3333
90	19333.3333
110	10150

Excluding Group Results

EMPLOYEES

DEPARTMENT_ID	SALARY
90	24000
90	17000
90	17000
60	9000
60	6000
60	4200
50	5800
50	3500
50	3100
50	2600
50	2500
80	10500
80	11000
80	8600
...	
20	6000
110	12000
110	8300

20 rows selected.

The maximum salary per department when it is greater than $10,000

DEPARTMENT_ID	MAX(SALARY)
20	13000
80	11000
90	24000
110	12000

ORACLE

Restricting Group Results

In the same way that you use the WHERE clause to restrict the rows that you select, you use the HAVING clause to restrict groups. To find the maximum salary of each department, but show only the departments that have a maximum salary of more than $10,000, you need to do the following:

1. Find the average salary for each department by grouping by department number.
2. Restrict the groups to those departments with a maximum salary greater than $10,000.

Excluding Group Results: The HAVING Clause

Use the HAVING clause to restrict groups:

1. **Rows are grouped.**
2. **The group function is applied.**
3. **Groups matching the HAVING clause are displayed.**

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

ORACLE

The HAVING Clause

You use the HAVING clause to specify which groups are to be displayed, and thus, you further restrict the groups on the basis of aggregate information.

In the syntax:

group_condition	restricts the groups of rows returned to those groups for which the specified condition is true

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

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

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

Instructor Note

The Oracle server evaluates the clauses in the following order:

- If the statement contains a WHERE clause, the server establishes the candidate rows.
- The server identifies the groups specified in the GROUP BY clause.
- The HAVING clause further restricts result groups that do not meet the group criteria in the HAVING clause.

Using the HAVING Clause

```
SELECT   department_id, MAX(salary)
FROM     employees
GROUP BY department_id
HAVING   MAX(salary)>10000 ;
```

DEPARTMENT_ID	MAX(SALARY)
20	13000
80	11000
90	24000
110	12000

ORACLE

The HAVING Clause (continued)

The slide example displays department numbers and maximum salaries for those departments whose maximum salary is greater than $10,000.

You can use the GROUP BY clause without using a group function in the SELECT list.

If you restrict rows based on the result of a group function, you must have a GROUP BY clause as well as the HAVING clause.

The following example displays the department numbers and average salaries for those departments whose maximum salary is greater than $10,000:

```
SELECT   department_id, AVG(salary)
FROM     employees
GROUP BY department_id
HAVING   max(salary)>10000;
```

DEPARTMENT_ID	AVG(SALARY)
20	9500
80	10033.3333
90	19333.3333
110	10150

Using the HAVING Clause

```
SELECT   job_id, SUM(salary) PAYROLL
FROM     employees
WHERE    job_id NOT LIKE '%REP%'
GROUP BY job_id
HAVING   SUM(salary) > 13000
ORDER BY SUM(salary);
```

JOB_ID	PAYROLL
IT_PROG	19200
AD_PRES	24000
AD_VP	34000

ORACLE

The HAVING Clause (continued)

The slide example displays the job ID and total monthly salary for each job with a total payroll exceeding $13,000. The example excludes sales representatives and sorts the list by the total monthly salary.

Nesting Group Functions

Display the maximum average salary.

```
SELECT   MAX(AVG(salary))
FROM     employees
GROUP BY department_id;
```

MAX(AVG(SALARY))
19333.3333

Nesting Group Functions

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

Summary

In this lesson, you should have learned how to:

- **Use the group functions `COUNT`, `MAX`, `MIN`, `AVG`**
- **Write queries that use the `GROUP BY` clause**
- **Write queries that use the `HAVING` clause**

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

ORACLE

Summary

Seven group functions are available in SQL:

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

You can create subgroups by using the GROUP BY clause. Groups can be excluded using the HAVING clause.

Place the HAVING and GROUP BY clauses after the WHERE clause in a statement. Place the ORDER BY clause last.

The Oracle server evaluates the clauses in the following order:

1. If the statement contains a WHERE clause, the server establishes the candidate rows.
2. The server identifies the groups specified in the GROUP BY clause.
3. The HAVING clause further restricts result groups that do not meet the group criteria in the HAVING clause.

Determine the validity of the following three statements. Circle either True or False.

1. Group functions work across many rows to produce one result per group.
   True/False
2. Group functions include nulls in calculations.
   True/False
3. The `WHERE` clause restricts rows prior to inclusion in a group calculation.
   True/False
4. Display the highest, lowest, sum, and average salary of all employees. Label the columns `Maximum`, `Minimum`, `Sum`, and `Average`, respectively. Round your results to the nearest whole number. Place your SQL statement in a text file named `lab5_4.sql`.

Maximum	Minimum	Sum	Average
24000	2500	175500	8775

5. Modify the query in `lab5_4.sql` to display the minimum, maximum, sum, and average salary for each job type. Resave `lab5_4.sql` to `lab5_5.sql`. Run the statement in `lab5_5.sql`.

JOB_ID	Maximum	Minimum	Sum	Average
AC_ACCOUNT	8300	8300	8300	8300
AC_MGR	12000	12000	12000	12000
AD_ASST	4400	4400	4400	4400
AD_PRES	24000	24000	24000	24000
AD_VP	17000	17000	34000	17000
IT_PROG	9000	4200	19200	6400
MK_MAN	13000	13000	13000	13000
MK_REP	6000	6000	6000	6000
SA_MAN	10500	10500	10500	10500
SA_REP	11000	7000	26600	8867
ST_CLERK	3500	2500	11700	2925
ST_MAN	5800	5800	5800	5800

12 rows selected.

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

JOB_ID	COUNT(*)
AC_ACCOUNT	1
AC_MGR	1
AD_ASST	1
AD_PRES	1
AD_VP	2
IT_PROG	3
MK_MAN	1
MK_REP	1
SA_MAN	1
SA_REP	3
ST_CLERK	4
ST_MAN	1

12 rows selected.

7. Determine the number of managers without listing them. Label the column `Number of Managers`. *Hint: Use the `MANAGER_ID` column to determine the number of managers.*

Number of Managers
8

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

DIFFERENCE
21500

If you have time, complete the following exercises:

9. Display the manager number and the salary of the lowest paid employee for that manager. Exclude anyone whose manager is not known. Exclude any groups where the minimum salary is $6,000 or less. Sort the output in descending order of salary.

MANAGER_ID	MIN(SALARY)
102	9000
205	8300
149	7000

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

Name	Location	Number of People	Salary
Accounting	1700	2	10150
Administration	1700	1	4400
Executive	1700	3	19333.33
IT	1400	3	6400
Marketing	1800	2	9500
Sales	2500	3	10033.33
Shipping	1500	5	3500

7 rows selected.

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

11. Create a query that will display the total number of employees and, of that total, the number of employees hired in 1995, 1996, 1997, and 1998. Create appropriate column headings.

TOTAL	1995	1996	1997	1998
20	1	2	2	3

12. Create a matrix query to display the job, the salary for that job based on department number, and the total salary for that job, for departments 20, 50, 80, and 90, giving each column an appropriate heading.

Job	Dept 20	Dept 50	Dept 80	Dept 90	Total
AC_ACCOUNT					8300
AC_MGR					12000
AD_ASST					4400
AD_PRES				24000	24000
AD_VP				34000	34000
IT_PROG					19200
MK_MAN	13000				13000
MK_REP	6000				6000
SA_MAN			10500		10500
SA_REP			19600		26600
ST_CLERK		11700			11700
ST_MAN		5800			5800

12 rows selected.

Chapter 7: Subqueries

Objectives

Using a Subquery to Solve a Problem

Subquery Syntax

Using a Subquery

Guidelines for Using Subqueries

Types of Subqueries

Single-Row Subqueries

Executing Single-Row Subqueries

Using Group Functions in a Subquery

The HAVING Clause with Subqueries

What is Wrong with this Statement?

Will this Statement Return Rows?

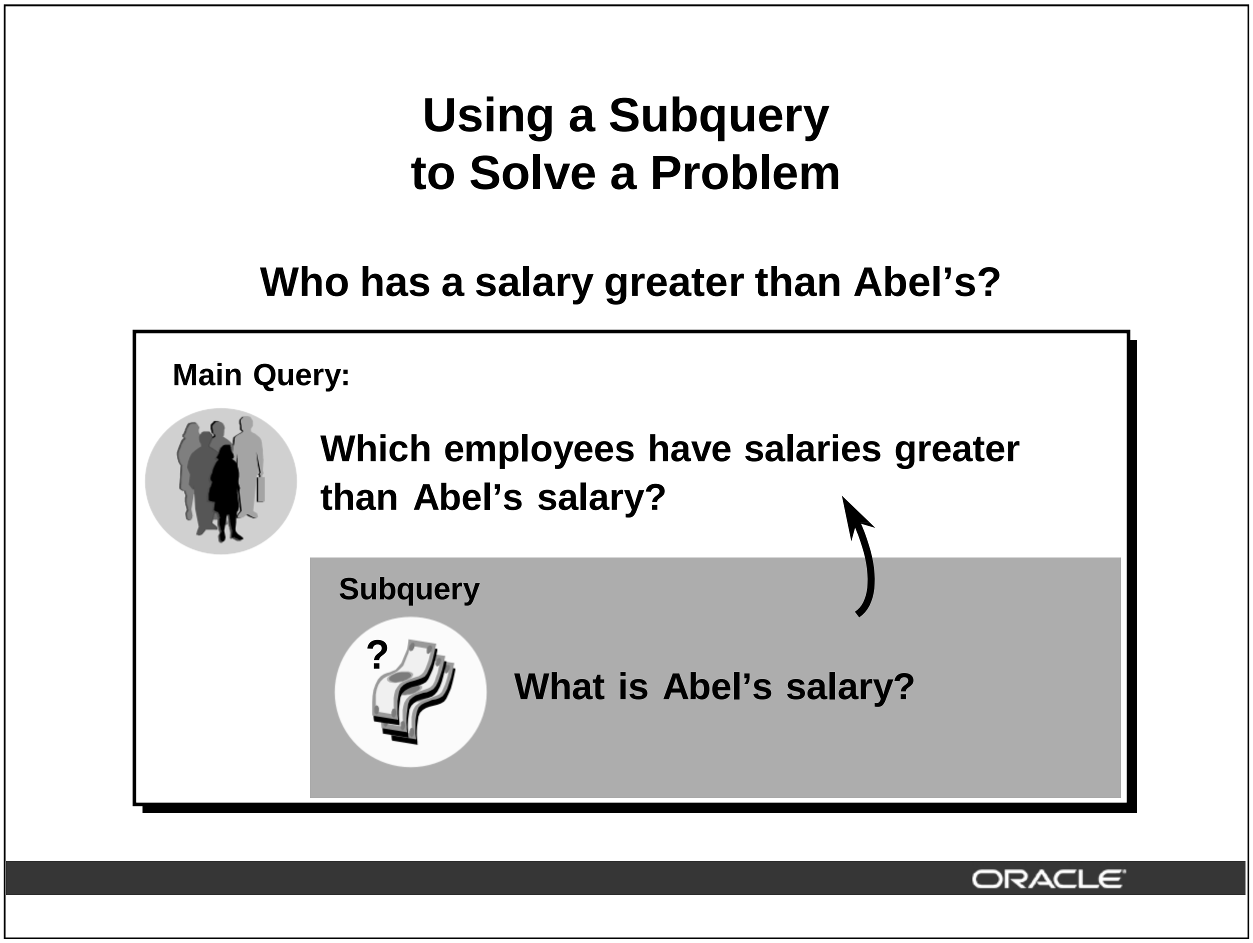
Multiple-Row Subqueries

Using the ANY Operator in Multiple-Row Subqueries

Using the ALL Operator in Multiple-Row Subqueries

Null Values in a Subquery

Summary



Using a Subquery to Solve a Problem

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

To solve this problem, you need two queries: one to find what Abel earns, and a second query to find who earns more than that amount.

You can solve this problem by combining the two queries, placing one query inside the other query. The inner query or the subquery returns a value that is used by the outer query or the main query.

Using a subquery is equivalent to performing two sequential queries and using the result of the first query as the search value in the second query.

Subquery Syntax

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

- **The subquery (inner query) executes once before the main query.**
- **The result of the subquery is used by the main query (outer query).**

ORACLE

Subqueries

A subquery is a `SELECT` statement that is embedded in a clause of another `SELECT` statement. You can build powerful statements out of simple ones by using subqueries. They can be very useful when you need to select rows from a table with a condition that depends on the data in the table itself.

You can place the subquery in a number of SQL clauses, including:

- The `WHERE` clause
- The `HAVING` clause
- The `FROM` clause

In the syntax:

`operator` includes a comparison condition such as >, =, or `IN`

Note: Comparison conditions fall into two classes: single-row operators (>, =, >=, <, <>, <=) and multiple-row operators (`IN`, `ANY`, `ALL`).

The subquery is often referred to as a nested `SELECT`, sub-`SELECT`, or inner `SELECT` statement. The subquery generally executes first, and its output is used to complete the query condition for the main or outer query.

Instructor Note

Additionally, subqueries can be placed in the `CREATE VIEW` statement, `CREATE TABLE` statement, `UPDATE` statement, `INTO` clause of an `INSERT` statement, and `SET` clause of an `UPDATE` statement.

Using a Subquery

```
SELECT  last_name
FROM    employees
WHERE   salary >
                 (SELECT  salary
                  FROM    employees
                  WHERE   last_name = 'Abel');
```

11000

LAST_NAME
King
Kochhar
De Haan
Hartstein
Higgins

Using a Subquery

In the slide, the inner query determines the salary of employee Abel. The outer query takes the result of the inner query and uses this result to display all the employees who earn more than this amount.

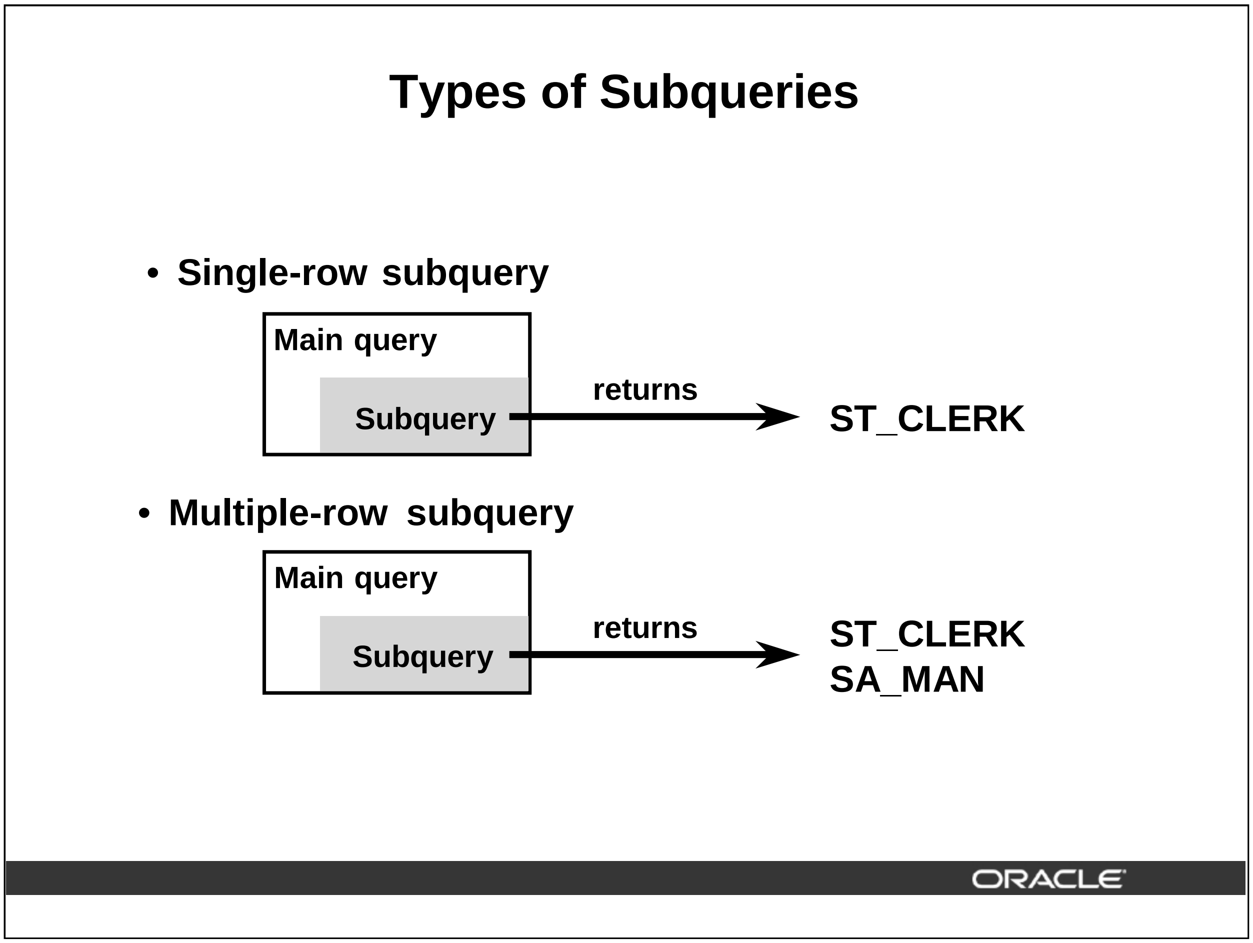
Guidelines for Using Subqueries

- **Enclose subqueries in parentheses.**
- **Place subqueries on the right side of the comparison condition.**
- **The `ORDER BY` clause in the subquery is not needed unless you are performing Top-N analysis.**
- **Use single-row operators with single-row subqueries and use multiple-row operators with multiple-row subqueries.**

ORACLE

Guidelines for Using Subqueries

- A subquery must be enclosed in parentheses.
- Place the subquery on the right side of the comparison condition for readability.
- Prior to release Oracle8*i*, subqueries could not contain an ORDER BY clause. Only one ORDER BY clause can be used for a SELECT statement, and if specified it must be the last clause in the main SELECT statement. Starting with release Oracle8*i*, an ORDER BY clause can be used and is required in the subquery to perform Top-N analysis.
- Two classes of comparison conditions are used in subqueries: single-row operators and multiple-row operators.



Types of Subqueries

- Single-row subqueries: Queries that return only one row from the inner SELECTstatement
- Multiple-row subqueries: Queries that return more than one row from the inner SELECT

statement

Note: There are also multiple-column subqueries: Queries that return more than one column from the inner SELECTstatement.

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

Single-Row Subqueries

A single-row subquery is one that returns one row from the inner `SELECT` statement. This type of subquery uses a single-row operator. The slide gives a list of single-row operators.

Example

Display the employees whose job ID is the same as that of employee 141.

```
SELECT last_name, job_id
FROM   employees
WHERE  job_id =
                (SELECT job_id
                 FROM   employees
                 WHERE  employee_id = 141);
```

LAST_NAME	JOB_ID
Rajs	ST_CLERK
Davies	ST_CLERK
Matos	ST_CLERK
Vargas	ST_CLERK

Executing Single-Row Subqueries

```
SELECT last_name, job_id, salary
FROM   employees
WHERE  job_id =                 ST_CLERK
                (SELECT job_id
                 FROM   employees
                 WHERE  employee_id = 141)
AND    salary >                  2600
                (SELECT salary
                 FROM   employees
                 WHERE  employee_id = 143);
```

LAST_NAME	JOB_ID	SALARY
Rajs	ST_CLERK	3500
Davies	ST_CLERK	3100

Executing Single-Row Subqueries

A `SELECT` statement can be considered as a query block. The example on the slide displays employees whose job ID is the same as that of employee 141 and whose salary is greater than that of employee 143.

The example consists of three query blocks: the outer query and two inner queries. The inner query blocks are executed first, producing the query results ST_CLERK and 2600, respectively. The outer query block is then processed and uses the values returned by the inner queries to complete its search conditions.

Both inner queries return single values (ST_CLERK and 2600, respectively), so this SQL statement is called a single-row subquery.

Note: The outer and inner queries can get data from different tables.

Using Group Functions in a Subquery

```
SELECT last_name, job_id, salary
FROM   employees
WHERE  salary =
                (SELECT MIN(salary)
                 FROM   employees);
```

2500

LAST_NAME	JOB_ID	SALARY
Vargas	ST_CLERK	2500

Using Group Functions in a Subquery

You can display data from a main query by using a group function in a subquery to return a single row. The subquery is in parentheses and is placed after the comparison condition.

The example on the slide displays the employee last name, job ID, and salary of all employees whose salary is equal to the minimum salary. The MINgroup function returns a single value (2500) to the outer query.

The HAVING Clause with Subqueries

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

```
SELECT   department_id, MIN(salary)
FROM     employees
GROUP BY department_id
HAVING   MIN(salary) >
                       (SELECT MIN(salary)
                        FROM   employees
                        WHERE  department_id = 50);
```

2500

ORACLE

The HAVING Clause with Subqueries

You can use subqueries not only in the WHERE clause, but also in the HAVING clause. The Oracle server executes the subquery, and the results are returned into the HAVING clause of the main query.

The SQL statement on the slide displays all the departments that have a minimum salary greater than that of department 50.

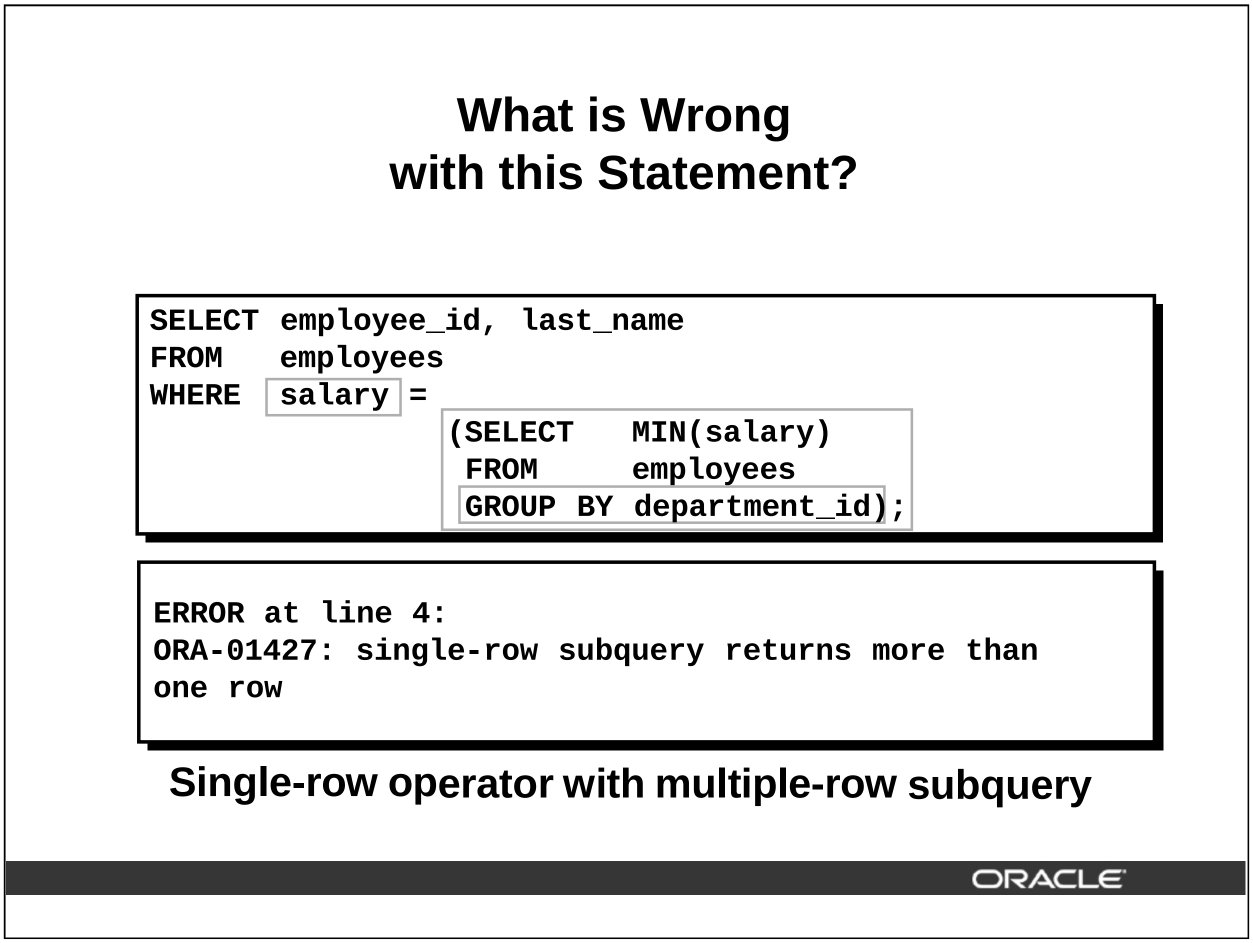
DEPARTMENT_ID	MIN(SALARY)
10	4400
20	6000
...	
	7000

7 rows selected.

Example

Find the job with the lowest average salary.

```
SELECT   job_id, AVG(salary)
FROM     employees
GROUP BY job_id
HAVING   AVG(salary) = (SELECT   MIN(AVG(salary))
                        FROM     employees
                        GROUP BY job_id);
```



Errors with Subqueries

One common error with subqueries is more than one row returned for a single-row subquery.

In the SQL statement on the slide, the subquery contains a GROUP BY clause, which implies that the subquery will return multiple rows, one for each group it finds. In this case, the result of the subquery will be 4400, 6000, 2500, 4200, 7000, 17000, and 8300.

The outer query takes the results of the subquery (4400, 6000, 2500, 4200, 7000, 17000, 8300) and uses these results in its WHERE clause. The WHERE clause contains an equal (=) operator, a single- row comparison operator expecting only one value. The = operator cannot accept more than one value from the subquery and therefore generates the error.

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

Will this Statement Return Rows?

```
SELECT last_name, job_id
FROM   employees
WHERE  job_id =
                (SELECT job_id
                 FROM   employees
                 WHERE  last_name = 'Haas');
```

```
no rows selected
```

Subquery returns no values

ORACLE

Problems with Subqueries

A common problem with subqueries is no rows being returned by the inner query.

In the SQL statement on the slide, the subquery contains a WHERE clause. Presumably, the intention is to find the employee whose name is Haas. The statement is correct but selects no rows when executed.

There is no employee named Haas. So the subquery returns no rows. The outer query takes the results of the subquery (null) and uses these results in its WHERE clause. The outer query finds no employee with a job ID equal to null, and so returns no rows. If a job existed with a value of null, the row is not returned because comparison of two null values yields a null, therefore the WHERE condition is not true.

Multiple-Row Subqueries

- **Return more than one row**
- **Use multiple-row comparison operators**

Operator	Meaning
`IN`	Equal to any member in the list
`ANY`	Compare value to each value returned by the subquery
`ALL`	Compare value to every value returned by the subquery

ORACLE

Multiple-Row Subqueries

Subqueries that return more than one row are called multiple-row subqueries. You use a multiple-row operator, instead of a single-row operator, with a multiple-row subquery. The multiple-row operator expects one or more values.

```
SELECT last_name, salary, department_id
FROM   employees
WHERE  salary IN (SELECT   MIN(salary)
                  FROM     employees
                  GROUP BY department_id);
```

Example

Find the employees who earn the same salary as the minimum salary for each department.

The inner query is executed first, producing a query result. The main query block is then processed and uses the values returned by the inner query to complete its search condition. In fact, the main query would appear to the Oracle server as follows:

```
SELECT last_name, salary, department_id
FROM   employees
WHERE  salary IN (2500, 4200, 4400, 6000, 7000, 8300, 8600, 17000);
```

Using the ANY Operator in Multiple-Row Subqueries

```
SELECT employee_id, last_name, job_id, salary
FROM   employees     9000, 6000, 4200
WHERE  salary < ANY
                    (SELECT salary
                     FROM   employees
                     WHERE  job_id = 'IT_PROG')
AND    job_id <> 'IT_PROG';
```

EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
124	Mourgos	ST_MAN	5800
141	Rajs	ST_CLERK	3500
142	Davies	ST_CLERK	3100
143	Matos	ST_CLERK	2600
144	Vargas	ST_CLERK	2500

...

10 rows selected.

ORACLE

Multiple-Row Subqueries (continued)

The ANY operator (and its synonym, the SOME operator) compares a value to each value returned by a subquery. The slide example displays employees who are not IT programmers and whose salary is less than that of any IT programmer. The maximum salary that a programmer earns is $9,000. <ANY means less than the maximum. >ANY means more than the minimum. =ANY is equivalent to IN. <ALL means less than the maximum. >ALL means more than the minimum.

Using the ALL Operator in Multiple-Row Subqueries

```
SELECT employee_id, last_name, job_id, salary
FROM   employees
WHERE  salary < ALL
                  (SELECT salary
                   FROM   employees
                   WHERE  job_id = 'IT_PROG')
AND    job_id <> 'IT_PROG';
```

9000, 6000, 4200

EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
141	Rajs	ST_CLERK	3500
142	Davies	ST_CLERK	3100
143	Matos	ST_CLERK	2600
144	Vargas	ST_CLERK	2500

Multiple-Row Subqueries (continued)

The `ALL` operator compares a value to *every* value returned by a subquery. The slide example displays employees whose salary is less than the salary of all employees with a job ID of IT_PROG and whose job is not IT_PROG.

`>ALL` means more than the maximum, and `<ALL` means less than the minimum.

The `NOT` operator can be used with `IN`, `ANY`, and `ALL` operators.

Null Values in a Subquery

```
SELECT emp.last_name
FROM   employees emp
WHERE  emp.employee_id NOT IN
                             (SELECT mgr.manager_id
                              FROM   employees mgr);

no rows selected
```

ORACLE

Returning Nulls in the Resulting Set of a Subquery

The SQL statement on the slide attempts to display all the employees who do not have any subordinates. Logically, this SQL statement should have returned 12 rows. However, the SQL statement does not return any rows. One of the values returned by the inner query is a null value, and hence the entire query returns no rows. The reason is that all conditions that compare a null value result in a null. So whenever null values are likely to be part of the results set of a subquery, do not use the `NOT IN` operator. The `NOT IN` operator is equivalent to `<> ALL`.

Notice that the null value as part of the results set of a subquery is not a problem if you use the `IN` operator. The `IN` operator is equivalent to `=ANY`. For example, to display the employees who have subordinates, use the following SQL statement:

```
SELECT emp.last_name
FROM   employees emp
WHERE  emp.employee_id  IN
                          (SELECT mgr.manager_id
                           FROM   employees mgr);
```

Alternatively, a `WHERE` clause can be included in the subquery to display all employees who do not have any subordinates:

```
SELECT last_name FROM employees
WHERE  employee_id NOT IN
                        (SELECT manager_id
                         FROM   employees
                         WHERE  manager_id IS NOT NULL);
```

Summary

In this lesson, you should have learned how to:

- **Identify when a subquery can help solve a question**
- **Write subqueries when a query is based on unknown values**

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

ORACLE

Summary

In this lesson, you should have learned how to use subqueries. A subquery is a `SELECT` statement that is embedded in a clause of another SQL statement. Subqueries are useful when a query is based on a search criteria with unknown intermediate values.

Subqueries have the following characteristics:

- Can pass one row of data to a main statement that contains a single-row operator, such as =, <>, >, >=, <, or <=
- Can pass multiple rows of data to a main statement that contains a multiple-row operator, such as `IN`
- Are processed first by the Oracle server, and the `WHERE` or `HAVING` clause uses the results
- Can contain group functions

1. Write a query to display the last name and hire date of any employee in the same department as Zlotkey. Exclude Zlotkey.

LAST_NAME	HIRE_DATE
Abel	11-MAY-96
Taylor	24-MAR-98

2. Create a query to display the employee numbers and last names of all employees who earn more than the average salary. Sort the results in ascending order of salary.

EMPLOYEE_ID	LAST_NAME	SALARY
103	Hunold	9000
149	Zlotkey	10500
174	Abel	11000
205	Higgins	12000
201	Hartstein	13000
101	Kochhar	17000
102	De Haan	17000
100	King	24000

8 rows selected.

3. Write a query that displays the employee numbers and last names of all employees who work in a department with any employee whose last name contains a *u*. Place your SQL statement in a text file named `lab6_3.sql`. Run your query.

EMPLOYEE_ID	LAST_NAME
124	Mourgos
141	Rajs
142	Davies
143	Matos
144	Vargas
103	Hunold
104	Ernst
107	Lorentz

8 rows selected.

4. Display the last name, department number, and job ID of all employees whose department location ID is 1700.

LAST_NAME	DEPARTMENT_ID	JOB_ID
Whalen	10	AD_ASST
King	90	AD_PRES
Kochhar	90	AD_VP
De Haan	90	AD_VP
Higgins	110	AC_MGR
Gietz	110	AC_ACCOUNT

6 rows selected.

5. Display the last name and salary of every employee who reports to King.

LAST_NAME	SALARY
Kochhar	17000
De Haan	17000
Mourgos	5800
Zlotkey	10500
Hartstein	13000

6. Display the department number, last name, and job ID for every employee in the Executive department.

DEPARTMENT_ID	LAST_NAME	JOB_ID
90	King	AD_PRES
90	Kochhar	AD_VP
90	De Haan	AD_VP

If you have time, complete the following exercises:

7. Modify the query in `lab6_3.sql` to display the employee numbers, last names, and salaries of all employees who earn more than the average salary and who work in a department with any employee with a *u* in their name. Resave `lab6_3.sql` to `lab6_7.sql`. Run the statement in `lab6_7.sql`.

EMPLOYEE_ID	LAST_NAME	SALARY
103	Hunold	9000

Chapter 8: Manipulating Data

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.**

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

70	Public Relations	100	1700

New row

DEPARTMENTS

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

…insert a new row into the `DEPARMENTS` table…

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

ORACLE

Adding a New Row to a Table

The slide graphic illustrates adding a new department to the `DEPARTMENTS` table.

The INSERT Statement Syntax

- **Add new rows to a table by using the INSERT statement.**

```
INSERT INTO   table [(column [, column...])]
VALUES        (value [, value...]);
```

- **Only one row is inserted at a time with this syntax.**

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.**

```
INSERT INTO departments(department_id, department_name,
                        manager_id, location_id)
VALUES      (70, 'Public Relations', 100, 1700);
1 row created.
```

- **Enclose character and date values within single quotation marks.**

ORACLE

Adding a New Row to a Table (continued)

Because you can insert a new row that contains values for each column, the column list is not required in the `INSERT` clause. However, if you do not use the column list, the values must be listed according to the default order of the columns in the table, and a value must be provided for each column.

```
DESCRIBE  departments
```

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

For clarity, use the column list in the `INSERT` clause.
Enclose character and date values within single quotation marks; it is not recommended to enclose numeric values within single quotation marks.

Number values should not be enclosed in single quotes, because implicit conversion may take place for numeric values assigned to `NUMBER` data type columns if single quotes are included.

Inserting Rows with Null Values

- **Implicit method: Omit the column from the column list.**

```
INSERT INTO   departments (department_id,
                           department_name   )
VALUES        (30, 'Purchasing');
1 row created.
```

- **Explicit method: Specify the `NULL` keyword in the `VALUES` clause.**

```
INSERT INTO   departments
VALUES        (100, 'Finance', NULL, NULL);
1 row created.
```

Methods for Inserting Null Values

Method	**Description**
Implicit	Omit the column from the column list.
Explicit	Specify the `NULL` keyword in the `VALUES` list, specify the empty string (`''`) in the `VALUES` list for character strings and dates.

Be sure that you can use null values in the targeted column by verifying the `Null?` status with the *i*SQL*Plus `DESCRIBE` command.

The Oracle Server automatically enforces all data types, data ranges, and data integrity constraints. Any column that is not listed explicitly obtains a null value in the new row.

Common errors that can occur during user input:

- Mandatory value missing for a `NOT NULL` column
- Duplicate value violates uniqueness constraint
- Foreign key constraint violated
- `CHECK` constraint violated
- Data type mismatch
- Value too wide to fit in column

Inserting Special Values

The SYSDATE function records the current date and time.

```
INSERT INTO employees (employee_id,
                 first_name, last_name,
                 email, phone_number,
                 hire_date, job_id, salary,
                 commission_pct, manager_id,
                 department_id)
VALUES          (113,
                 'Louis', 'Popp',
                 'LPOPP', '515.124.4567',
                 SYSDATE, 'AC_ACCOUNT', 6900,
                 NULL, 205, 100);
1 row created.
```

ORACLE

Inserting Special Values by Using SQL Functions

You can use functions to enter special values in your table.

The slide example records information for employee Popp in the EMPLOYEES table. It supplies the current date and time in the HIRE_DATE column. It uses the SYSDATE function for current date and time.

You can also use the USER function when inserting rows in a table. The USER function records the current username.

Confirming Additions to the Table

```
SELECT employee_id, last_name, job_id, hire_date, commission_pct
FROM   employees
WHERE  employee_id = 113;
```

EMPLOYEE_ID	LAST_NAME	JOB_ID	HIRE_DATE	COMMISSION_PCT
113	Popp	AC_ACCOUNT	27-SEP-01	

Inserting Specific Date Values

- **Add a new employee.**

```
INSERT INTO employees
VALUES      (114,
             'Den', 'Raphealy',
             'DRAPHEAL', '515.127.4561',
             TO_DATE('FEB 3, 1999', 'MON DD, YYYY'),
             'AC_ACCOUNT', 11000, NULL, 100, 30);
1 row created.
```

- **Verify your addition.**

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_F
114	Den	Raphealy	DRAPHEAL	515.127.4561	03-FEB-99	AC_ACCOUNT	11000	

Inserting Specific Date and Time Values

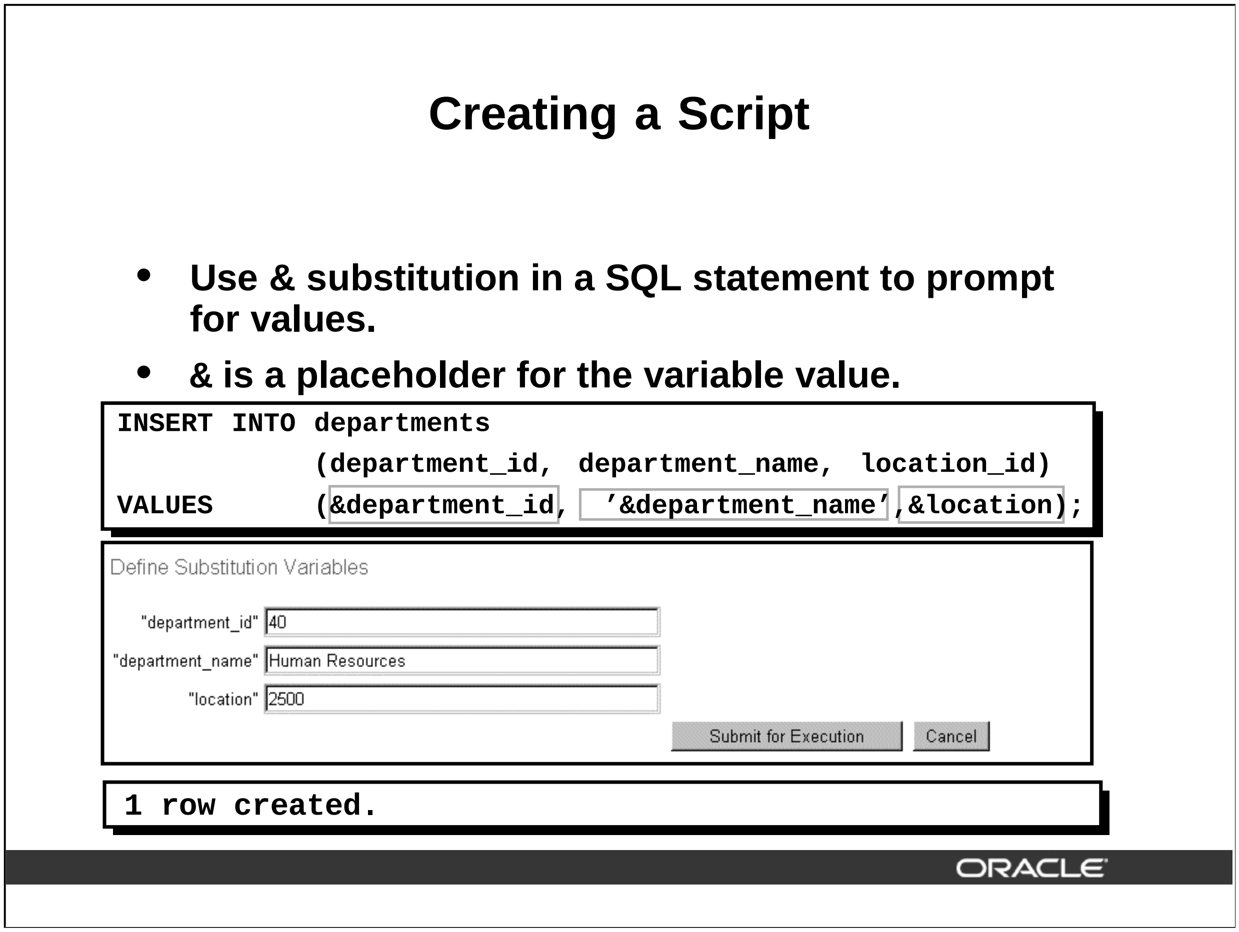
The `DD-MON-YY` format is usually used to insert a date value. With this format, recall that the century defaults to the current century. Because the date also contains time information, the default time is midnight (00:00:00).

If a date must be entered in a format other than the default format, for example, with another century, or a specific time, you must use the `TO_DATE` function.

The example on the slide records information for employee Raphealy in the `EMPLOYEES` table. It sets the `HIRE_DATE` column to be February 3, 1999. If you use the following statement instead of the one shown on the slide, the year of the hire_date is interpreted as 2099.

```
INSERT INTO employees
VALUES      (114,
             'Den', 'Raphealy',
             'DRAPHEAL', '515.127.4561',
             '03-FEB-99',
             'AC_ACCOUNT', 11000, NULL, 100, 30);
```

If the RR format is used, the system provides the correct century automatically, even if it is not the current one.



Creating a Script to Manipulate Data

You can save commands with substitution variables to a file and execute the commands in the file. The example above records information for a department in the `DEPARTMENTS` table.

Run the script file and you are prompted for input for the & substitution variables. The values you input are then substituted into the statement. This allows you to run the same script file over and over, but supply a different set of values each time you run it.

Copying Rows from Another Table

- **Write your INSERT statement with a subquery.**

```
INSERT INTO sales_reps(id, name, salary, commission_pct)
  SELECT employee_id, last_name, salary, commission_pct
  FROM   employees
  WHERE  job_id LIKE '%REP%';

4 rows created.
```

- **Do not use the VALUES clause.**
- **Match the number of columns in the INSERT clause to those in the subquery.**

ORACLE

Copying Rows from Another Table

You can use the `INSERT` statement to add rows to a table where the values are derived from existing tables. In place of the `VALUES` clause, you use a subquery.

Syntax

```
INSERT INTO table [ column (, column) ] subquery;
```

In the syntax:

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

The number of columns and their data types in the column list of the `INSERT` clause must match the number of values and their data types in the subquery. To create a copy of the rows of a table, use `SELECT *` in the subquery.

```
INSERT INTO copy_emp
  SELECT *
  FROM   employees;
```

Changing Data in a Table

EMPLOYEES

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	HIRE_DATE	JOB_ID	SALARY	DEPARTMENT_ID	COMMISSION_
100	Steven	King	SKING	17-JUN-87	AD_PRES	24000	90	
101	Neena	Kochhar	NKOCHHAR	21-SEP-89	AD_VP	17000	90	
102	Lex	De Haan	LDEHAAN	13-JAN-93	AD_VP	17000	90	
103	Alexander	Hunold	AHUNOLD	03-JAN-90	IT_PROG	9000	60	
104	Bruce	Ernst	BERNST	21-MAY-91	IT_PROG	6000	60	
107	Diana	Lorentz	DLORENTZ	07-FEB-99	IT_PROG	4200	60	
124	Kevin	Mourgos	KMOURGOS	16-NOV-99	ST_MAN	5800	50	

Update rows in the `EMPLOYEES` table.

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	HIRE_DATE	JOB_ID	SALARY	DEPARTMENT_ID	COMMISSIO
100	Steven	King	SKING	17-JUN-87	AD_PRES	24000	90	
101	Neena	Kochhar	NKOCHHAR	21-SEP-89	AD_VP	17000	90	
102	Lex	De Haan	LDEHAAN	13-JAN-93	AD_VP	17000	90	
103	Alexander	Hunold	AHUNOLD	03-JAN-90	IT_PROG	9000	30	
104	Bruce	Ernst	BERNST	21-MAY-91	IT_PROG	6000	30	
107	Diana	Lorentz	DLORENTZ	07-FEB-99	IT_PROG	4200	30	
124	Kevin	Mourgos	KMOURGOS	16-NOV-99	ST_MAN	5800	50	

ORACLE

Changing Data in a Table

The slide graphic illustrates changing the department number for employees in department 60 to department 30.

The UPDATE Statement Syntax

- **Modify existing rows with the UPDATE statement.**

```
UPDATE      table
SET         column = value [, column = value, ...]
[WHERE      condition];
```

- **Update more than one row at a time, if required.**

Updating Rows

You can modify existing rows by using the `UPDATE` 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 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 *Oracle9i SQL Reference*, "`UPDATE`."

Note: In general, use the primary key to identify a single row. Using other columns can unexpectedly cause several rows to be updated. For example, identifying a single row in the `EMPLOYEES` table by name is dangerous, because more than one employee may have the same name.

Updating Rows in a Table

- **Specific row or rows are modified if you specify the WHERE clause.**

```
UPDATE  employees
SET     department_id = 70
WHERE   employee_id = 113;
1 row updated.
```

- **All rows in the table are modified if you omit the WHERE clause.**

```
UPDATE   copy_emp
SET      department_id = 110;
22 rows updated.
```

ORACLE

Updating Rows (continued)

The `UPDATE` statement modifies specific rows if the `WHERE` clause is specified. The slide example transfers employee 113 (Popp) to department 70.

If you omit the `WHERE` clause, all the rows in the table are modified.

```
SELECT last_name, department_id
FROM   copy_emp;
```

LAST_NAME	DEPARTMENT_ID
King	110
Kochhar	110
De Haan	110
Hunold	110
Ernst	110
Lorentz	110

...

22 rows selected.

Note: The `COPY_EMP` table has the same data as the `EMPLOYEES` table.

Updating Two Columns with a Subquery

Update employee 114's job and salary to match that of employee 205.

```
UPDATE   employees
SET      job_id  = (SELECT  job_id
                    FROM    employees
                    WHERE   employee_id = 205),
         salary  = (SELECT  salary
                    FROM    employees
                    WHERE   employee_id = 205)
WHERE    employee_id    =  114;
1 row updated.
```

ORACLE

Updating Two Columns with a Subquery

You can update multiple columns in the SET clause of an UPDATE statement by writing multiple subqueries.

Syntax

```
UPDATE table
SET     column  =
                (SELECT column
                 FROM table
                 WHERE condition)
       [ ,
        column  =
                (SELECT column
                 FROM table
                 WHERE condition)]
[WHERE  condition ]     ;
```

Note: If no rows are updated, a message "`0 rows updated`." is returned.

Updating Rows Based on Another Table

Use subqueries in `UPDATE` statements to update rows in a table based on values from another table.

```
UPDATE  copy_emp
SET     department_id  =  (SELECT department_id
                           FROM employees
                           WHERE employee_id = 100)
WHERE   job_id         =  (SELECT job_id
                           FROM employees
                           WHERE employee_id = 200);
1 row updated.
```

Updating Rows Based on Another Table

You can use subqueries in `UPDATE` statements to update rows in a table. The example on the slide updates the `COPY_EMP` table based on the values from the `EMPLOYEES` table. It changes the department number of all employees with employee 200’s job ID to employee 100’s current department number.

Updating Rows: Integrity Constraint Error

```
UPDATE employees
SET    department_id = 55
WHERE  department_id = 110;
```

```
UPDATE employees
       *
ERROR at line 1:
ORA-02291: integrity constraint (HR.EMP_DEPT_FK)
violated - parent key not found
```

Department number 55 does not exist

Integrity Constraint Error

If you attempt to update a record with a value that is tied to an integrity constraint, an error is returned.

In the example on the slide, department number 55 does not exist in the parent table, `DEPARTMENTS`, and so you receive the *parent key* violation `ORA-02291`.

Note: Integrity constraints ensure that the data adheres to a predefined set of rules. A subsequent lesson covers integrity constraints in greater depth.

Removing a Row from a Table

DEPARTMENTS

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
30	Purchasing		
100	Finance		
50	Shipping	124	1500
60	IT	103	1400

Delete a row from the `DEPARTMENTS` table.

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
30	Purchasing		
50	Shipping	124	1500
60	IT	103	1400

Removing a Row from a Table

The slide graphic removes the Finance department from the `DEPARTMENTS` table (assuming that there are no constraints defined on the `DEPARTMENTS` table).

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

Note: If no rows are deleted, a message "0 rows deleted." is returned:

For more information, see *Oracle9i SQL Reference,* "DELETE."

Deleting Rows from a Table

- **Specific rows are deleted if you specify the `WHERE` clause.**

```
 DELETE FROM departments
 WHERE  department_name = 'Finance';
1 row deleted.
```

- **All rows in the table are deleted if you omit the `WHERE` clause.**

```
DELETE FROM  copy_emp;
22 rows deleted.
```

ORACLE

Deleting Rows (continued)

You can delete specific rows by specifying the `WHERE` clause in the `DELETE` statement. The slide example deletes the Finance department from the `DEPARTMENTS` table. You can confirm the delete operation by displaying the deleted rows using the `SELECT` statement.

```
SELECT   *
FROM     departments
WHERE    department_name = 'Finance';
no rows selected.
```

If you omit the `WHERE` clause, all rows in the table are deleted. The second example on the slide deletes all the rows from the `COPY_EMP` table, because no `WHERE` clause has been specified.

Example

Remove rows identified in the `WHERE` clause.

```
DELETE FROM  employees
WHERE        employee_id = 114;

1 row deleted.

DELETE FROM  departments
WHERE        department_id IN (30, 40);

2 rows deleted.
```

Deleting Rows Based on Another Table

Use subqueries in DELETE statements to remove rows from a table based on values from another table.

```
DELETE FROM employees
WHERE  department_id =
                        (SELECT department_id
                         FROM   departments
                         WHERE  department_name LIKE '%Public%');
1 row deleted.
```

ORACLE

Deleting Rows Based on Another Table

You can use subqueries to delete rows from a table based on values from another table. The example on the slide deletes all the employees who are in a department where the department name contains the string “Public.” The subquery searches the DEPARTMENTS table to find the department number based on the department name containing the string “Public.” The subquery then feeds the department number to the main query, which deletes rows of data from the EMPLOYEES table based on this department number.

Deleting Rows: Integrity Constraint Error

```
DELETE FROM departments
WHERE       department_id = 60;
```

```
DELETE FROM departments
            *
ERROR at line 1:
ORA-02292: integrity constraint (HR.EMP_DEPT_FK)
violated - child record found
```

You cannot delete a row that contains a primary key that is used as a foreign key in another table.

Integrity Constraint Error

If you attempt to delete a record with a value that is tied to an integrity constraint, an error is returned.

The example on the slide tries to delete department number 60 from the `DEPARTMENTS` table, but it results in an error because department number is used as a foreign key in the `EMPLOYEES` table. If the parent record that you attempt to delete has child records, then you receive the *child record found* violation `ORA-02292`.

The following statement works because there are no employees in department 70:

```
DELETE FROM  departments
WHERE        department_id = 70;

1 row deleted.
```

Using a Subquery in an INSERT Statement

```
INSERT INTO
        (SELECT employee_id, last_name,
                email, hire_date, job_id, salary,
                department_id
         FROM   employees
         WHERE  department_id = 50)
VALUES (99999, 'Taylor', 'DTAYLOR',
        TO_DATE('07-JUN-99', 'DD-MON-RR'),
        'ST_CLERK', 5000, 50);

1 row created.
```

ORACLE

Using a Subquery in an INSERT Statement

You can use a subquery in place of the table name in the INTO clause of the INSERT statement.

The select list of this subquery must have the same number of columns as the column list of the VALUES clause. Any rules on the columns of the base table must be followed in order for the INSERT statement to work successfully. For example, you could not put in a duplicate employee ID, nor leave out a value for a mandatory not null column.

Using a Subquery in an INSERT Statement

```
SELECT employee_id, last_name, email, hire_date,
       job_id, salary, department_id
FROM   employees
WHERE  department_id = 50;
```

EMPLOYEE_ID	LAST_NAME	EMAIL	HIRE_DATE	JOB_ID	SALARY	DEPARTMENT_ID
124	Mourgos	KMOURGOS	16-NOV-99	ST_MAN	5800	50
141	Rajs	TRAJS	17-OCT-95	ST_CLERK	3500	50
142	Davies	CDAVIES	29-JAN-97	ST_CLERK	3100	50
143	Matos	RMATOS	15-MAR-98	ST_CLERK	2600	50
144	Vargas	PVARGAS	09-JUL-98	ST_CLERK	2500	50
99999	Taylor	DTAYLOR	07-JUN-99	ST_CLERK	5000	50

6 rows selected.

Using a Subquery in an INSERT Statement

The example shows the results of the subquery that was used to identify the table for the INSERT statement.

Using the `WITH CHECK OPTION` Keyword on DML Statements

- **A subquery is used to identify the table and columns of the DML statement.**
- **The `WITH CHECK OPTION` keyword prohibits you from changing rows that are not in the subquery.**

```
INSERT INTO  (SELECT employee_id, last_name, email,
                     hire_date, job_id, salary
              FROM   employees
              WHERE  department_id = 50 WITH CHECK OPTION)
VALUES (99998, 'Smith', 'JSMITH',
        TO_DATE('07-JUN-99', 'DD-MON-RR'),
        'ST_CLERK', 5000);
INSERT INTO
         *
ERROR at line 1:
ORA-01402: view WITH CHECK OPTION where-clause violation
```

ORACLE

The `WITH CHECK OPTION` Keyword

Specify `WITH CHECK OPTION` to indicate that, if the subquery is used in place of a table in an `INSERT`, `UPDATE`, or `DELETE` statement, no changes that would produce rows that are not included in the subquery are permitted to that table.

In the example shown, the `WITH CHECK OPTION` keyword is used. The subquery identifies rows that are in department 50, but the department ID is not in the `SELECT` list, and a value is not provided for it in the `VALUES` list. Inserting this row would result in a department ID of null, which is not in the subquery.

Overview of the Explicit Default Feature

- With the explicit default feature, you can use the `DEFAULT` keyword as a column value where the column default is desired.
- The addition of this feature is for compliance with the SQL: 1999 Standard.
- This allows the user to control where and when the default value should be applied to data.
- Explicit defaults can be used in `INSERT` and `UPDATE` statements.

Explicit Defaults

The `DEFAULT` keyword can be used in `INSERT` and `UPDATE` statements to identify a default column value. If no default value exists, a null value is used.

Using Explicit Default Values

- **DEFAULT with INSERT:**

```
INSERT INTO departments
  (department_id, department_name, manager_id)
VALUES (300, 'Engineering', DEFAULT);
```

- **DEFAULT with UPDATE:**

```
UPDATE departments
SET manager_id = DEFAULT WHERE department_id = 10;
```

Using Explicit Default Values

Specify `DEFAULT` to set the column to the value previously specified as the default value for the column. If no default value for the corresponding column has been specified, Oracle sets the column to null.

In the first example shown, the `INSERT` statement uses a default value for the `MANAGER_ID` column. If there is no default value defined for the column, a null value is inserted instead.

The second example uses the `UPDATE` statement to set the `MANAGER_ID` column to a default value for department 10. If no default value is defined for the column, it changes the value to null.

Note: When creating a table, you can specify a default value for a column. This is discussed in the "Creating and Managing Tables" lesson.

The `MERGE` Statement

- **Provides the ability to conditionally update or insert data into a database table**
- **Performs an `UPDATE` if the row exists, and an `INSERT` if it is a new row:**
  - **Avoids separate updates**
  - **Increases performance and ease of use**
  - **Is useful in data warehousing applications**

ORACLE

`MERGE` Statements

SQL has been extended to include the `MERGE` statement. Using this statement, you can update or insert a row conditionally into a table, thus avoiding multiple `UPDATE` statements. The decision whether to update or insert into the target table is based on a condition in the `ON` clause.

Since the `MERGE` command combines the `INSERT` and `UPDATE` commands, you need both `INSERT` and `UPDATE` privileges on the target table and the `SELECT` privilege on the source table.

The `MERGE` statement is deterministic. You cannot update the same row of the target table multiple times in the same `MERGE` statement.

An alternative approach is to use PL/SQL loops and multiple DML statements. The `MERGE` statement, however, is easy to use and more simply expressed as a single SQL statement.

The `MERGE` statement is suitable in a number of data warehousing applications. For example, in a data warehousing application, you may need to work with data coming from multiple sources, some of which may be duplicates. With the `MERGE` statement, you can conditionally add or modify rows.

The MERGE Statement Syntax

You can conditionally insert or update rows in a table by using the MERGE statement.

```
MERGE INTO table_name table_alias
  USING (table|view|sub_query) alias
  ON (join condition)
  WHEN MATCHED THEN
    UPDATE SET
    col1 = col_val1,
    col2 = col2_val
  WHEN NOT MATCHED THEN
    INSERT (column_list)
    VALUES (column_values);
```

8-3

Merging Rows

You can update existing rows and insert new rows conditionally by using the MERGE statement.

In the syntax:

INTO clause	specifies the target table you are updating or inserting into
USING clause	identifies the source of the data to be updated or inserted; can be a table, view, or subquery
ON clause	the condition upon which the MERGE operation either updates or inserts
WHEN MATCHED |	instructs the server how to respond to the results of the join condition
WHEN NOT MATCHED	

For more information, see *Oracle9i SQL Reference*, "MERGE."

Merging Rows

Insert or update rows in the COPY_EMP table to match the EMPLOYEES table.

```
MERGE INTO copy_emp  c
  USING employees e
  ON (c.employee_id = e.employee_id)
WHEN MATCHED THEN
  UPDATE SET
     c.first_name     = e.first_name,
     c.last_name      = e.last_name,
     ...
     c.department_id  = e.department_id
WHEN NOT MATCHED THEN
 INSERT VALUES(e.employee_id, e.first_name, e.last_name,
          e.email, e.phone_number, e.hire_date, e.job_id,
          e.salary, e.commission_pct, e.manager_id,
          e.department_id);
```

Example of Merging Rows

```
MERGE INTO copy_emp c
  USING employees e
  ON (c.employee_id = e.employee_id)
WHEN MATCHED THEN
  UPDATE SET
    c.first_name     = e.first_name,
    c.last_name      = e.last_name,
    c.email          = e.email,
    c.phone_number   = e.phone_number,
    c.hire_date      = e.hire_date,
    c.job_id         = e.job_id,
    c.salary         = e.salary,
    c.commission_pct = e.commission_pct,
    c.manager_id     = e.manager_id,
    c.department_id  = e.department_id
WHEN NOT MATCHED THEN
  INSERT VALUES(e.employee_id, e.first_name, e.last_name,
       e.email, e.phone_number, e.hire_date, e.job_id,
       e.salary, e.commission_pct, e.manager_id,
       e.department_id);
```

The example shown matches the `EMPLOYEE_ID` in the `COPY_EMP` table to the `EMPLOYEE_ID` in the `EMPLOYEES` table. If a match is found, the row in the `COPY_EMP` table is updated to match the row in the `EMPLOYEES` table. If the row is not found, it is inserted into the `COPY_EMP` table.

Merging Rows

```
SELECT *
FROM COPY_EMP;

no rows selected
```

```
MERGE INTO copy_emp c
  USING employees e
  ON (c.employee_id = e.employee_id)
WHEN MATCHED THEN
  UPDATE SET
     ...
WHEN NOT MATCHED THEN
 INSERT VALUES...;
```

```
SELECT *
FROM COPY_EMP;

20 rows selected.
```

ORACLE

Example of Merging Rows

The condition `c.employee_id = e.employee_id` is evaluated. Because the `COPY_EMP` table is empty, the condition returns false: there are no matches. The logic falls into the `WHEN NOT MATCHED` clause, and the `MERGE` command inserts the rows of the `EMPLOYEES` table into the `COPY_EMP` table.

If rows existed in the `COPY_EMP` table and employee IDs matched in both tables (the `COPY_EMP` and `EMPLOYEES` tables), the existing rows in the `COPY_EMP` table would be updated to match the `EMPLOYEES` table.

Database Transactions

A database transaction consists of one of the following:

- **DML statements which constitute one consistent change to the data**
- **One DDL statement**
- **One DCL statement**

ORACLE

Database Transactions

The Oracle server ensures data consistency based on transactions. Transactions give you more flexibility and control when changing data, and they ensure data consistency in the event of user process failure or system failure.

Transactions consist of DML statements that make up one consistent change to the data. For example, a transfer of funds between two accounts should include the debit to one account and the credit to another account in the same amount. Both actions should either fail or succeed together; the credit should not be committed without the debit.

Transaction Types

Type	Description
Data manipulation language (DML)	Consists of any number of DML statements that the Oracle server treats as a single entity or a logical unit of work
Data definition language (DDL)	Consists of only one DDL statement
Data control language (DCL)	Consists of only one DCL statement

Database Transactions

- **Begin when the first DML SQL statement is executed**
- **End with one of the following events:**
  - **A `COMMIT` or `ROLLBACK` statement is issued**
  - **A DDL or DCL statement executes (automatic commit)**
  - **The user exits *i*SQL*Plus**
  - **The system crashes**

ORACLE

When Does a Transaction Start and End?

A transaction begins when the first DML statement is encountered and ends when one of the following occurs:

- A `COMMIT` or `ROLLBACK` statement is issued
- A DDL statement, such as `CREATE`, is issued
- A DCL statement is issued
- The user exits *i*SQL*Plus
- A machine fails or the system crashes

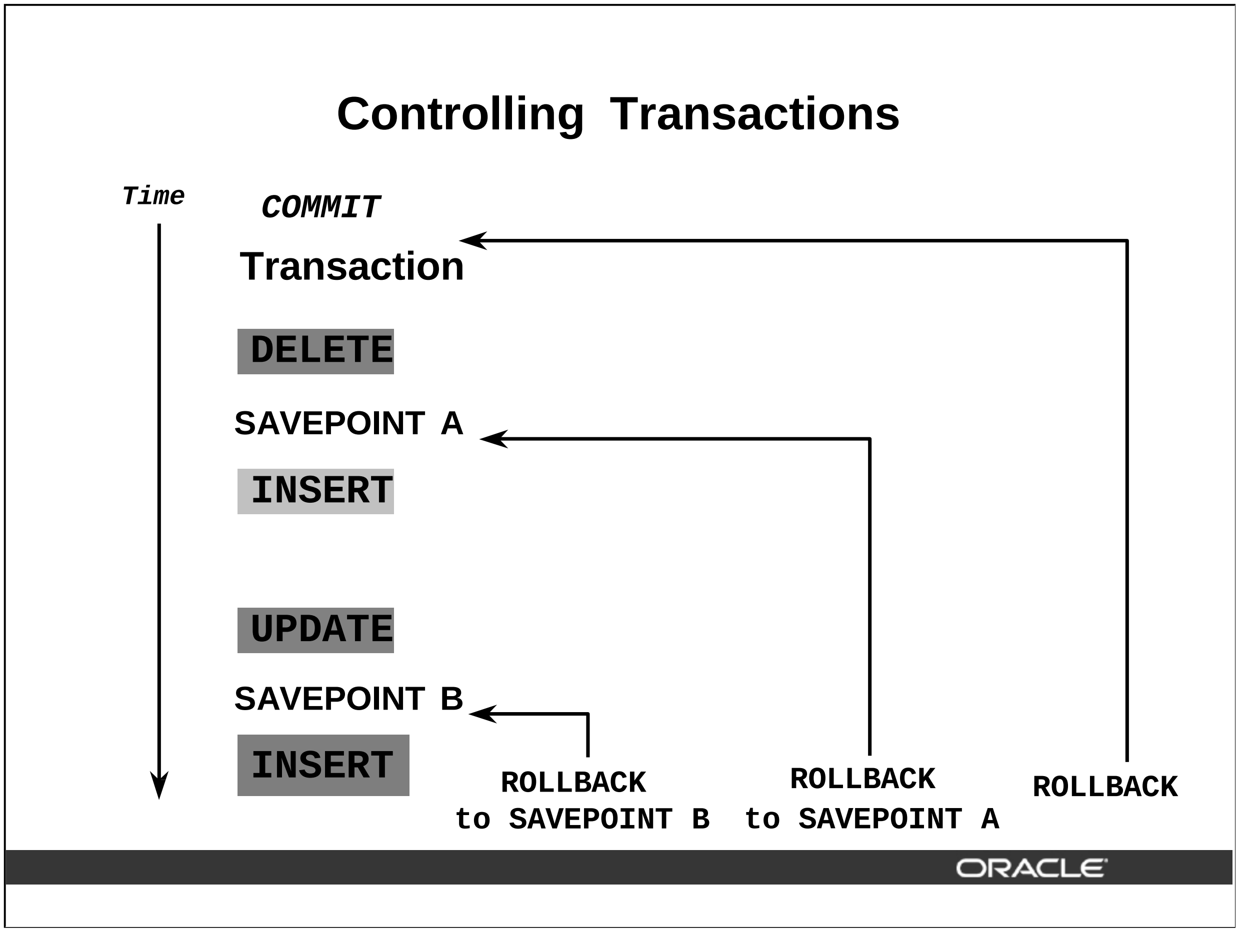
After one transaction ends, the next executable SQL statement automatically starts the next transaction.

A DDL statement or a DCL statement is automatically committed and therefore implicitly ends a transaction.

Advantages of COMMIT and ROLLBACK Statements

With COMMIT and ROLLBACK statements, you can:

- Ensure data consistency
- Preview data changes before making changes permanent
- Group logically related operations



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`	`ROLLBACK` ends the current transaction by discarding all pending data changes
`ROLLBACK TO` *`SAVEPOINT name`*	`ROLLBACK TO SAVEPOINT` rolls back the current transaction to the specified savepoint, thereby discarding any changes and or savepoints created after the savepoint to which you are rolling back. If you omit the `TO SAVEPOINT` clause, the `ROLLBACK` statement rolls back the entire transaction. As savepoints are logical, there is no way to list the savepoints you have created.

Note: `SAVEPOINT` is not ANSI standard SQL.

Rolling Back Changes to a Marker

- Create a marker in a current transaction by using the `SAVEPOINT` statement.
- Roll back to that marker by using the `ROLLBACK TO SAVEPOINT` statement.

```
UPDATE...
SAVEPOINT update_done;
Savepoint created.
INSERT...
ROLLBACK TO update_done;
Rollback complete.
```

ORACLE

Rolling Back Changes to a Savepoint

You can create a marker in the current transaction by using the `SAVEPOINT` statement which divides the transaction into smaller sections. You can then discard pending changes up to that marker by using the `ROLLBACK TO SAVEPOINT` statement.

If you create a second savepoint with the same name as an earlier savepoint, the earlier savepoint is deleted.

Implicit Transaction Processing

- **An automatic commit occurs under the following circumstances:**
  - **DDL statement is issued**
  - **DCL statement is issued**
  - **Normal exit from *i*SQL*Plus, without explicitly issuing `COMMIT` or `ROLLBACK` statements**
- **An automatic rollback occurs under an abnormal termination of *i*SQL*Plus or a system failure.**

ORACLE

Implicit Transaction Processing

Status	Circumstances
Automatic commit	DDL statement or DCL statement is issued. *i*SQL*Plus exited normally, without explicitly issuing `COMMIT` or `ROLLBACK` commands.
Automatic rollback	Abnormal termination of *i*SQL*Plus or system failure.

Note: A third command is available in *i*SQL*Plus. The `AUTOCOMMIT` command can be toggled on or off. If set to *on*, each individual DML statement is committed as soon as it is executed. You cannot roll back the changes. If set to *off*, the `COMMIT` statement can still be issued explicitly. Also, the `COMMIT` statement is issued when a DDL statement is issued or when you exit from *i*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.

From *i*SQL*Plus, a normal exit from the session is accomplished by clicking the Exit button. With SQL*Plus, a normal exit is accomplished by typing the command `EXIT` at the prompt. Closing the window is interpreted as an abnormal exit.

State of the Data
Before `COMMIT` or `ROLLBACK`

- **The previous state of the data can be recovered.**
- **The current user can review the results of the DML operations by using the `SELECT` statement.**
- **Other users *cannot* view the results of the DML statements by the current user.**
- **The affected rows are *locked*; other users cannot change the data within the affected rows.**

Committing Changes

Every data change made during the transaction is temporary until the transaction is committed.

State of the data before `COMMIT` or `ROLLBACK` statements are issued:

- Data manipulation operations primarily affect the database buffer; therefore, the previous state of the data can be recovered.
- The current user can review the results of the data manipulation operations by querying the tables.
- Other users cannot view the results of the data manipulation operations made by the current user. The Oracle server institutes read consistency to ensure that each user sees data as it existed at the last commit.
- The affected rows are locked; other users cannot change the data in the affected rows.

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` statement:

- Data changes are written to the database.
- The previous state of the data is permanently lost.
- All users can view the results of the transaction.
- The locks on the affected rows are released; the rows are now available for other users to perform new data changes.
- All savepoints are erased.

Committing Data

- **Make the changes.**

```
DELETE FROM employees
WHERE  employee_id = 99999;
1 row deleted.

INSERT INTO departments
VALUES (290, 'Corporate Tax', NULL, 1700);
1 row inserted.
```

- **Commit the changes.**

```
COMMIT;
Commit complete.
```

ORACLE

Committing Changes (continued)

The slide example deletes a row from the `EMPLOYEES` table and inserts a new row into the `DEPARTMENTS` table. It then makes the change permanent by issuing the `COMMIT` statement.

Example

Remove departments 290 and 300 in the `DEPARTMENTS` table, and update a row in the `COPY_EMP` table. Make the data change permanent.

```
DELETE FROM departments
WHERE  department_id IN (290, 300);

2 rows deleted.

 UPDATE  copy_emp
   SET   department_id = 80
   WHERE employee_id = 206;

 1 row updated.

COMMIT;

Commit Complete.
```

State of the Data After ROLLBACK

Discard all pending changes by using the `ROLLBACK` statement:

- **Data changes are undone.**
- **Previous state of the data is restored.**
- **Locks on the affected rows are released.**

```
DELETE FROM copy_emp;
22 rows deleted.
ROLLBACK;
Rollback complete.
```

ORACLE

Rolling Back Changes

Discard all pending changes by using the `ROLLBACK` statement. Following a `ROLLBACK` statement:

- Data changes are undone.
- The previous state of the data is restored.
- The locks on the affected rows are released.

Example

While attempting to remove a record from the `TEST` table, you can accidentally empty the table. You can correct the mistake, reissue the proper statement, and make the data change permanent.

```
DELETE FROM test;
25,000 rows deleted.
ROLLBACK;
Rollback complete.
DELETE FROM test
WHERE  id = 100;
1 row deleted.
SELECT *
FROM   test
WHERE  id = 100;
No rows selected.
COMMIT;
Commit complete.
```

Statement-Level Rollback

- **If a single DML statement fails during execution, only that statement is rolled back.**
- **The Oracle server implements an implicit savepoint.**
- **All other changes are retained.**
- **The user should terminate transactions explicitly by executing a `COMMIT` or `ROLLBACK` statement.**

ORACLE

Statement-Level Rollbacks

Part of a transaction can be discarded by an implicit rollback if a statement execution error is detected. If a single DML statement fails during execution of a transaction, its effect is undone by a statement-level rollback, but the changes made by the previous DML statements in the transaction are not discarded. They can be committed or rolled back explicitly by the user.

Oracle issues an implicit commit before and after any data definition language (DDL) statement. So, even if your DDL statement does not execute successfully, you cannot roll back the previous statement because the server issued a commit.

Terminate your transactions explicitly by executing a `COMMIT` or `ROLLBACK` statement.

Summary

In this lesson, you should have learned how to use DML statements and control transactions.

Statement	Description
`INSERT`	Adds a new row to the table
`UPDATE`	Modifies existing rows in the table
`DELETE`	Removes existing rows from the table
`MERGE`	Conditionally inserts or updates data in a table
`COMMIT`	Makes all pending changes permanent
`SAVEPOINT`	Is used to rollback to the savepoint marker
`ROLLBACK`	Discards all pending data changes

ORACLE

8-4

Summary

In this lesson, you should have learned how to manipulate data in the Oracle database by using the `INSERT`, `UPDATE`, and `DELETE` statements. Control data changes by using the `COMMIT`, `SAVEPOINT`, and `ROLLBACK` statements.

The Oracle server guarantees a consistent view of data at all times.

Locking can be implicit or explicit.

Insert data into the `MY_EMPLOYEE` table.

1. Run the statement in the `lab8_1.sql` script to build the `MY_EMPLOYEE` table to be used for the lab.
2. Describe the structure of the `MY_EMPLOYEE` table to identify the column names.

Name	Null?	Type
ID	NOT NULL	NUMBER(4)
LAST_NAME		VARCHAR2(25)
FIRST_NAME		VARCHAR2(25)
USERID		VARCHAR2(8)
SALARY		NUMBER(9,2)

3. Add the first row of data to the `MY_EMPLOYEE` table from the following sample data. Do not list the columns in the `INSERT` clause.

ID	LAST_NAME	FIRST_NAME	USERID	SALARY
1	Patel	Ralph	rpatel	895
2	Dancs	Betty	bdancs	860
3	Biri	Ben	bbiri	1100
4	Newman	Chad	cnewman	750
5	Ropeburn	Audrey	aropebur	1550

4. Populate the `MY_EMPLOYEE` table with the second row of sample data from the preceding list. This time, list the columns explicitly in the `INSERT` clause.
5. Confirm your addition to the table.

ID	LAST_NAME	FIRST_NAME	USERID	SALARY
1	Patel	Ralph	rpatel	895
2	Dancs	Betty	bdancs	860

6. Write an insert statement in a text file named `loademp.sql` to load rows into the `MY_EMPLOYEE` table. Concatenate the first letter of the first name and the first seven characters of the last name to produce the user ID.
7. Populate the table with the next two rows of sample data by running the insert statement in the script that you created.
8. Confirm your additions to the table.

ID	LAST_NAME	FIRST_NAME	USERID	SALARY
1	Patel	Ralph	rpatel	895
2	Dancs	Betty	bdancs	860
3	Biri	Ben	bbiri	1100
4	Newman	Chad	cnewman	750

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.

ID	LAST_NAME	FIRST_NAME	USERID	SALARY
1	Patel	Ralph	rpatel	1000
2	Dancs	Betty	bdancs	1000
3	Drexler	Ben	bbiri	1100
4	Newman	Chad	cnewman	1000

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

15. Commit all pending changes.

Control data transaction to the `MY_EMPLOYEE` table.

16. Populate the table with the last row of sample data by modifying the statements in the script that you created in step 6. Run the statements in the script.
17. Confirm your addition to the table.

ID	LAST_NAME	FIRST_NAME	USERID	SALARY
1	Patel	Ralph	rpatel	1000
3	Drexler	Ben	bbiri	1100
4	Newman	Chad	cnewman	1000
5	Ropeburn	Audrey	aropebur	1550

18. Mark an intermediate point in the processing of the transaction.
19. Empty the entire table.
20. Confirm that the table is empty.

21. Discard the most recent `DELETE` operation without discarding the earlier `INSERT` operation.
22. Confirm that the new row is still intact.

ID	LAST_NAME	FIRST_NAME	USERID	SALARY
1	Patel	Ralph	rpatel	1000
3	Drexler	Ben	bbiri	1100
4	Newman	Chad	cnewman	1000
5	Ropeburn	Audrey	aropebur	1550

23. Make the data addition permanent.

Chapter 9: Creating and Managing Tables

Database Objects

Object	Description
Table	Basic unit of storage; composed of rows and columns
View	Logically represents subsets of data from one or more tables
Sequence	Numeric value generator
Index	Improves the performance of some queries
Synonym	Gives alternative names to objects

ORACLE

Database Objects

An Oracle database can contain multiple data structures. Each structure should be outlined in the database design so that it can be created during the build stage of database development.

- Table: Stores data
- View: Subset of data from one or more tables
- Sequence: Numeric value generator
- Index: Improves the performance of some queries
- Synonym: Gives alternative names to objects

Oracle9*i* 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 Rules

Table names and column names:

- Must begin with a letter
- Must be 1–30 characters long
- Must contain only A–Z, a–z, 0–9, _, $, and #
- Must not duplicate the name of another object owned by the same user
- Must not be an Oracle server reserved word

ORACLE

Naming Rules

Name database tables and columns according to the standard rules for naming any Oracle database object:

- Table names and column names must begin with a letter and be 1–30 characters long.
- Names must contain only the characters A–Z, a–z, 0–9, _ (underscore), $, and # (legal characters, but their use is discouraged).
- Names must not duplicate the name of another object owned by the same Oracle server user.
- Names must not be an Oracle server reserved word.

Naming Guidelines

Use descriptive names for tables and other database objects.

Note: Names are case insensitive. For example, `EMPLOYEES` is treated as the same name as `eMPloyees` or `eMpLOYEES`.

For more information, see *Oracle9i SQL Reference*,“Object Names and Qualifiers.”

The CREATE TABLE Statement

- **You must have:**
  - **CREATE TABLE privilege**
  - **A storage area**

```
CREATE TABLE [schema.]table
             (column datatype [DEFAULT expr][, ...]);
```

- **You specify:**
  - **Table name**
  - **Column name, column data type, and column size**

The CREATE TABLE Statement

Create tables to store data by executing the SQL `CREATE TABLE` statement. This statement is one of the data definition language (DDL) statements, that are covered in subsequent lessons. DDL statements are a subset of SQL statements used to create, modify, or remove Oracle9*i* 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:

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 data type 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 those tables.**

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. For example, if there is a schema named `USER_B`, and `USER_B` has an `EMPLOYEES` table, then specify the following to retrieve data from that table:

```
SELECT *
FROM   user_b.employees;
```

The DEFAULT Option

- Specify a default value for a column during an insert.

```
... hire_date DATE DEFAULT SYSDATE, ...
```

- Literal values, expressions, or SQL functions are legal values.
- Another column's name or a pseudocolumn are illegal values.
- The default data type must match the column data type.

ORACLE

The DEFAULT Option

A column can be given a default value by using the DEFAULT option. This option prevents null values from entering the columns if a row is inserted without a value for the column. The default value can be a literal, an expression, or a SQL function, such as SYSDATE and USER, but the value cannot be the name of another column or a pseudocolumn, such as NEXTVAL or CURRVAL. The default expression must match the data type of the column.

Note: CURRVAL and NEXTVAL are explained later.

Creating Tables

- **Create the table.**

```
CREATE TABLE dept
            (deptno  NUMBER(2),
             dname   VARCHAR2(14),
             loc     VARCHAR2(13));
Table created.
```

- **Confirm table creation.**

```
DESCRIBE dept
```

Name	Null?	Type
DEPTNO		NUMBER(2)
DNAME		VARCHAR2(14)
LOC		VARCHAR2(13)

ORACLE

Creating Tables

The example on the slide creates the `DEPT` table, with three columns: `DEPTNO`, `DNAME`, and `LOC`. It further confirms the creation of the table by issuing the `DESCRIBE` command.

Since creating a table is a DDL statement, an automatic commit takes place when this statement is executed.

Tables in the Oracle Database

- **User Tables:**
  - **Are a collection of tables created and maintained by the user**
  - **Contain user information**
- **Data Dictionary:**
  - **Is a collection of tables created and maintained by the Oracle Server**
  - **Contain database information**

Tables in the Oracle Database

User tables are tables created by the user, such as `EMPLOYEES`. There is another collection of tables and views in the Oracle database known as the *data dictionary*. This collection is created and maintained by the Oracle server and contains information about the database.

All data dictionary tables are owned by the `SYS` user. The base tables are rarely accessed by the user because the information in them is not easy to understand. Therefore, users typically access data dictionary views because the information is presented in a format that is easier to understand. Information stored in the data dictionary includes names of the Oracle server users, privileges granted to users, database object names, table constraints, and auditing information.

There are four categories of data dictionary views; each category has a distinct prefix that reflects its intended use.

Prefix	**Description**
`USER_`	These views contain information about objects owned by the user
`ALL_`	These views contain information about all of the tables (object tables and relational tables) accessible to the user.
`DBA_`	These views are restricted views, which can be accessed only by people who have been assigned the DBA role.
`V$`	These views are dynamic performance views, database server performance, memory, and locking.

Querying the Data Dictionary

- **See the names of tables owned by the user.**

```
SELECT table_name
FROM   user_tables ;
```

- **View distinct object types owned by the user.**

```
SELECT DISTINCT object_type
FROM   user_objects ;
```

- **View tables, views, synonyms, and sequences owned by the user.**

```
SELECT *
FROM   user_catalog ;
```

ORACLE

Querying the Data Dictionary

You can query the data dictionary tables to view various database objects owned by you. The data dictionary tables frequently used are these:

- USER_TABLES
- USER_OBJECTS
- USER_CATALOG

Note: USER_CATALOG has a synonym called CAT. You can use this synonym instead of USER_CATALOG in SQL statements.

```
SELECT   *
FROM     CAT;
```

Data Types

Data Type	Description
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	**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**
ROWID	**A 64 base number system representing the unique address of a row in its table.**

ORACLE

Data Types

Data type	**Description**
VARCHAR2(*size*)	Variable-length character data (a maximum *size* must be specified: 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 to the nearest second between January 1, 4712 B.C., and December 31, 9999 A.D.
LONG	Variable-length character data up to 2 gigabytes
CLOB	Character data up to 4 gigabytes

Data Types (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
`ROWID`	A 64 base number system representing the unique address of a row in its table.

- A `LONG` column is not copied when a table is created using a subquery.
- A `LONG` column cannot be included in a `GROUP BY` or an `ORDER BY` clause.
- Only one `LONG` column can be used per table.
- No constraints can be defined on a `LONG` column.
- You may want to use a `CLOB` column rather than a `LONG` column.

DateTime Data Types

Datetime enhancements with Oracle9*i*:

- **New Datetime data types have been introduced.**
- **New data type storage is available.**
- **Enhancements have been made to time zones and local time zone.**

Data Type	Description
`TIMESTAMP`	Date with fractional seconds
`INTERVAL YEAR TO MONTH`	Stored as an interval of years and months
`INTERVAL DAY TO SECOND`	Stored as an interval of days to hours minutes and seconds

ORACLE

Other DateTime Data Types

Data Type	Description
TIMESTAMP	Allows the time to be stored as a date with fractional seconds. There are several variations of the data type.
INTERVAL YEAR TO MONTH	Allows time to be stored as an interval of years and months. Used to represent the difference between two datetime values, where the only significant portions are the year and month.
INTERVAL DAY TO SECOND	Allows time to be stored as an interval of days to hours, minutes, and seconds. Useful in representing the precise difference between two datetime values.

DateTime Data Types

- **The `TIMESTAMP` data type is an extension of the `DATE` data type.**
- **It stores the year, month, and day of the `DATE` data type, plus hour, minute, and second values as well as the fractional second value.**
- **The `TIMESTAMP` data type is specified as follows:**

```
TIMESTAMP[(fractional_seconds_precision)]
```

DateTime Data Types

The `fractional_seconds_precision` optionally specifies the number of digits in the fractional part of the `SECOND` datetime field and can be a number in the range 0 to 9. The default is 6.

Example

```
CREATE TABLE new_employees
(employee_id NUMBER,
 first_name VARCHAR2(15),
 last_name VARCHAR2(15),
 ...
 start_date TIMESTAMP(7),
 ...);
```

In the preceding example, we are creating a table `NEW_EMPLOYEES` with a column `start_date` with a data type of `TIMESTAMP`. The precision of '`7`' indicates the fractional seconds precision which if not specified defaults to '`6`'.

Assume that two rows are inserted into the `NEW_EMPLOYEES` table. The output shows the differences in the display. (A `DATE` data type defaults to display the format of `DD-MON-RR`):

```
SELECT start_date
FROM   new_employees;

17-JUN-87 12.00.00.000000 AM
21-P-89 12.00.00.000000 AM
```

TIMESTAMP WITH TIME ZONE Data Type

- **TIMESTAMP WITH TIME ZONE is a variant of TIMESTAMP that includes a time zone displacement in its value.**
- **The time zone displacement is the difference, in hours and minutes, between local time and UTC.**

```
TIMESTAMP[(fractional_seconds_precision)]
WITH TIME ZONE
```

Datetime Data Types

UTC stands for Coordinated Universal Time—formerly Greenwich Mean Time. Two `TIMESTAMP WITH TIME ZONE` values are considered identical if they represent the same instant in UTC, regardless of the `TIME ZONE` offsets stored in the data.

Because `TIMESTAMP WITH TIME ZONE` can also store time zone information, it is particularly suited for recording date information that must be gathered or coordinated across geographic regions.

For example,

```
TIMESTAMP ’1999-04-15 8:00:00 -8:00’
```

is the same as

```
TIMESTAMP ’1999-04-15 11:00:00 -5:00’
```

That is, 8:00 a.m. Pacific Standard Time is the same as 11:00 a.m. Eastern Standard Time.

This can also be specified as

```
TIMESTAMP ’1999-04-15 8:00:00 US/Pacific’
```

Note: `fractional_seconds_precision` optionally specifies the number of digits in the fractional part of the `SECOND` datetime field and can be a number in the range 0 to 9. The default is 6.

TIMESTAMP WITH LOCAL TIME Data Type

- **TIMESTAMP WITH LOCAL TIME ZONE is another variant of TIMESTAMP that includes a time zone displacement in its value.**
- **Data stored in the database is normalized to the database time zone.**
- **The time zone displacement is not stored as part of the column data; Oracle returns the data in the users' local session time zone.**
- **TIMESTAMP WITH LOCAL TIME ZONE data type is specified as follows:**

```
TIMESTAMP[(fractional_seconds_precision)]
WITH LOCAL TIME ZONE
```

ORACLE

DateTime Data Types

Unlike TIMESTAMP WITH TIME ZONE, you can specify columns of type TIMESTAMP WITH LOCAL TIME ZONE as part of a primary or unique key. The time zone displacement is the difference (in hours and minutes) between local time and UTC. There is no literal for TIMESTAMP WITH LOCAL TIME ZONE.

Note: fractional_seconds_precision optionally specifies the number of digits in the fractional part of the SECOND datetime field and can be a number in the range 0 to 9. The default is 6.

Example

```
CREATE TABLE time_example
 (order_date TIMESTAMP WITH LOCAL TIME ZONE);

INSERT INTO time_example VALUES('15-NOV-00 09:34:28 AM');

SELECT *
FROM   time_example;

order_date
----------------------------
15-NOV-00 09.34.28.000000 AM
```

The TIMESTAMP WITH LOCAL TIME ZONE type is appropriate for two-tier applications where you want to display dates and times using the time zone of the client system.

INTERVAL YEAR TO MONTH Data Type

- **INTERVAL YEAR TO MONTH stores a period of time using the YEAR and MONTH datetime fields.**

```
INTERVAL YEAR [(year_precision)] TO MONTH
```

```
INTERVAL ’123-2’ YEAR(3) TO MONTH
Indicates an interval of 123 years, 2 months.

INTERVAL ’123’ YEAR(3)
Indicates an interval of 123 years 0 months.

INTERVAL ’300’ MONTH(3)
Indicates an interval of 300 months.

INTERVAL ’123’ YEAR
Returns an error, because the default precision is 2,
and ’123’ has 3 digits.
```

ORACLE

INTERVAL YEAR TO MONTH Data Type

INTERVAL YEAR TO MONTH stores a period of time using the YEAR and MONTH datetime fields. Use INTERVAL YEAR TO MONTH to represent the difference between two datetime values, where the only significant portions are the year and month. For example, you might use this value to set a reminder for a date 120 months in the future, or check whether 6 months have elapsed since a particular date.

Specify INTERVAL YEAR TO MONTH as follows:

```
INTERVAL YEAR [(year_precision)] TO MONTH
```

In the syntax:

year_precision	is the number of digits in the YEAR datetime field. The default value of year_precision is 2.

Example

```
CREATE TABLE time_example2
(loan_duration INTERVAL YEAR (3) TO MONTH);

INSERT INTO time_example2 (loan_duration)
VALUES (INTERVAL ’120’ MONTH(3));

SELECT TO_CHAR( sysdate+loan_duration, ’dd-mon-yyyy’)
FROM   time_example2;            --today’s date is 26-Sep-2001
```

INTERVAL DAY TO SECOND Data Type

- **INTERVAL DAY TO SECOND stores a period of time in terms of days, hours, minutes, and seconds.**

```
INTERVAL DAY [(day_precision)]
   TO SECOND [(fractional_seconds_precision)]
```

```
INTERVAL '4 5:12:10.222' DAY TO SECOND(3)
Indicates 4 days, 5 hours, 12 minutes, 10 seconds,
and 222 thousandths of a second.INTERVAL '123' YEAR(3).

INTERVAL '7' DAY
Indicates 7 days.

INTERVAL '180' DAY(3)
Indicates 180 days.
```

ORACLE

INTERVAL DAY TO SECOND Data Type

INTERVAL DAY TO SECOND stores a period of time in terms of days, hours, minutes, and seconds.

Use INTERVAL DAY TO SECOND to represent the precise difference between two datetime values. For example, you might use this value to set a reminder for a time 36 hours in the future, or to record the time between the start and end of a race. To represent long spans of time, including multiple years, with high precision, you can use a large value for the days portion.

Specify INTERVAL DAY TO SECOND as follows:

```
INTERVAL DAY [(day_precision)]
TO SECOND [(fractional_seconds_precision)]
```

In the syntax:

day_precision	is the number of digits in the DAY datetime field. Accepted values are 0 to 9. The default is 2.
fractional_seconds_precision	is the number of digits in the fractional part of the SECOND datetime field. Accepted values are 0 to 9. The default is 6.

INTERVAL DAY TO SECOND Data Type

- **INTERVAL DAY TO SECOND stores a period of time in terms of days, hours, minutes, and seconds.**

```
INTERVAL '4 5:12:10.222' DAY TO SECOND(3)
Indicates 4 days, 5 hours, 12 minutes, 10 seconds,
and 222 thousandths of a second.

INTERVAL '4 5:12' DAY TO MINUTE
Indicates 4 days, 5 hours and 12 minutes.

INTERVAL '400 5' DAY(3) TO HOUR
Indicates 400 days 5 hours.

INTERVAL '11:12:10.2222222' HOUR TO SECOND(7)
indicates 11 hours, 12 minutes, and 10.2222222 seconds.
```

INTERVAL DAY TO SECOND Data Type

Example

```
CREATE TABLE time_example3
(day_duration INTERVAL DAY (3) TO SECOND);

INSERT INTO time_example3 (day_duration)
VALUES (INTERVAL '180' DAY(3));

SELECT sysdate + day_duration "Half Year"
FROM   time_example3;        --today's date is 26-Sep-2001
```

Half Year
25-MAR-02

Creating a Table by Using a Subquery Syntax

- **Create a table and insert rows by combining the `CREATE TABLE` statement and the `AS` *`subquery`* option.**

```
CREATE TABLE table
       [(column, column...)]
AS subquery;
```

- **Match the number of specified columns to the number of subquery columns.**
- **Define columns with column names and default values.**

ORACLE

Creating a Table from Rows in Another Table

A second method for creating a table is to apply the `AS` *`subquery`* clause, which both creates the table and inserts rows returned from the subquery.

In the syntax:

`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 is created with the specified column names, and the rows retrieved by the `SELECT` statement are inserted into the table.
- The column definition can contain only the column name and default value.
- If column specifications are given, the number of columns must equal the number of columns in the subquery `SELECT` list.
- If no column specifications are given, the column names of the table are the same as the column names in the subquery.
- The integrity rules are not passed onto the new table, only the column data type definitions.

Creating a Table by Using a Subquery

```
CREATE TABLE  dept80
  AS
    SELECT  employee_id, last_name,
            salary*12 ANNSAL,
            hire_date
    FROM    employees
    WHERE   department_id = 80;
Table created.
```

```
DESCRIBE dept80
```

Name	Null?	Type
EMPLOYEE_ID		NUMBER(6)
LAST_NAME	NOT NULL	VARCHAR2(25)
ANNSAL		NUMBER
HIRE_DATE	NOT NULL	DATE

Creating a Table from Rows in Another Table (continued)

The slide example creates a table named DEPT80, which contains details of all the employees working in department 80. Notice that the data for the DEPT80 table comes from the EMPLOYEES table.

You can verify the existence of a database table and check column definitions by using the *i*SQL*Plus DESCRIBE command.

Be sure to give a column alias when selecting an expression. The expression SALARY*12 is given the alias ANNSAL. Without the alias, this error is generated:

```
ERROR at line 3:
ORA-00998: must name this expression with a column alias
```

The ALTER TABLE Statement

Use the ALTER TABLE statement to:

- **Add a new column**
- **Modify an existing column**
- **Define a default value for the new column**
- **Drop a column**

ORACLE

The ALTER TABLE Statement

After you create a table, you may need to change the table structure because: you omitted a column, your column definition needs to be changed, or you need to remove columns. You can do this by using the ALTER TABLE statement.

The ALTER TABLE Statement

Use the ALTER TABLE statement to add, modify, or drop columns.

```
ALTER TABLE table
ADD         (column datatype [DEFAULT expr]
            [, column datatype]...);
```

```
ALTER TABLE table
MODIFY      (column datatype [DEFAULT expr]
            [, column datatype]...);
```

```
ALTER TABLE table
DROP       (column);
```

ORACLE®

The ALTER TABLE Statement (continued)

You can add, modify, and drop columns to a table by using the `ALTER TABLE` statement.

In the syntax:

table	is the name of the table
ADD|MODIFY|DROP	is the type of modification
column	is the name of the new column
datatype	is the data type and length of the new column
DEFAULT *expr*	specifies the default value for a new column

Note: The slide gives the abridged syntax for `ALTER TABLE`. More about `ALTER TABLE` is covered in a subsequent lesson.

Adding a Column

New column

DEPT80

EMPLOYEE_ID	LAST_NAME	ANNSAL	HIRE_DATE
149	Zlotkey	126000	29-JAN-00
174	Abel	132000	11-MAY-96
176	Taylor	103200	24-MAR-98

JOB_ID

"Add a new column to the DEPT80 table."

DEPT80

EMPLOYEE_ID	LAST_NAME	ANNSAL	HIRE_DATE	JOB_ID
149	Zlotkey	126000	29-JAN-00	
174	Abel	132000	11-MAY-96	
176	Taylor	103200	24-MAR-98	

ORACLE

Adding a Column

The graphic adds the `JOB_ID` column to the `DEPT80` table. Notice that the new column becomes the last column in the table.

Adding a Column

- **You use the ADD clause to add columns.**

```
ALTER TABLE dept80
ADD         (job_id VARCHAR2(9));
Table altered.
```

- **The new column becomes the last column.**

EMPLOYEE_ID	LAST_NAME	ANNSAL	HIRE_DATE	JOB_ID
149	Zlotkey	126000	29-JAN-00	
174	Abel	132000	11-MAY-96	
176	Taylor	103200	24-MAR-98	

ORACLE

Guidelines for Adding a Column

- You can add or modify columns.
- You cannot specify where the column is to appear. The new column becomes the last column.

The example on the slide adds a column named `JOB_ID` to the `DEPT80` table. The `JOB_ID` column becomes the last column in the table.

Note: If a table already contains rows when a column is added, then the new column is initially null for all the rows.

Modifying a Column

- **You can change a column's data type, size, and default value.**

```
ALTER TABLE   dept80
MODIFY        (last_name VARCHAR2(30));
Table altered.
```

- **A change to the default value affects only subsequent insertions to the table.**

ORACLE

Modifying a Column

You can modify a column definition by using the `ALTER TABLE` statement with the `MODIFY` clause. Column modification can include changes to a column's data type, size, and default value.

Guidelines

- You can increase the width or precision of a numeric column.
- You can increase the width of numeric or character columns.
- You can decrease the width of a column only if the column contains only null values or if the table has no rows.
- You can change the data type only if the column contains null values.
- You can convert a `CHAR` column to the `VARCHAR2` data type or convert a `VARCHAR2` column to the `CHAR` data type only if the column contains null values or if you do not change the size.
- A change to the default value of a column affects only subsequent insertions to the table.

Dropping a Column

Use the `DROP COLUMN` clause to drop columns you no longer need from the table.

```
ALTER TABLE  dept80
DROP COLUMN  job_id;
Table altered.
```

Dropping a Column

You can drop a column from a table by using the `ALTER TABLE` statement with the `DROP COLUMN` clause. This is a feature available in Oracle8*i* and later.

Guidelines

- The column may or may not contain data.
- Using the `ALTER TABLE` statement, only one column can be dropped at a time.
- The table must have at least one column remaining in it after it is altered.
- Once a column is dropped, it cannot be recovered.

The SET UNUSED Option

- You use the SET UNUSED option to mark one or more columns as unused.
- You use the DROP UNUSED COLUMNS option to remove the columns that are marked as unused.

```
ALTER TABLE    table
SET   UNUSED  (column);
OR
ALTER TABLE  table
SET   UNUSED COLUMN column;
```

```
ALTER TABLE table
DROP  UNUSED COLUMNS;
```

ORACLE

The SET UNUSED Option

The SET UNUSED option marks one or more columns as unused so that they can be dropped when the demand on system resources is lower. This is a feature available in Oracle8*i* and later. Specifying this clause does not actually remove the target columns from each row in the table (that is, it does not restore the disk space used by these columns). Therefore, the response time is faster than if you executed the DROP clause. Unused columns are treated as if they were dropped, even though their column data remains in the table's rows. After a column has been marked as unused, you have no access to that column. A SELECT * query will not retrieve data from unused columns. In addition, the names and types of columns marked unused will not be displayed during a DESCRIBE, and you can add to the table a new column with the same name as an unused column. SET UNUSED information is stored in the USER_UNUSED_COL_TABS dictionary view.

The DROP UNUSED COLUMNS Option

DROP UNUSED COLUMNS removes from the table all columns currently marked as unused. You can use this statement when you want to reclaim the extra disk space from unused columns in the table. If the table contains no unused columns, the statement returns with no errors.

```
ALTER TABLE  dept80
SET   UNUSED (last_name);
Table altered.

ALTER TABLE  dept80
DROP  UNUSED COLUMNS;
Table altered.
```

Dropping a Table

- **All data and structure in the table is deleted.**
- **Any pending transactions are committed.**
- **All indexes are dropped.**
- **You *cannot* roll back the `DROP TABLE` statement.**

```
DROP TABLE dept80;
Table dropped.
```

ORACLE

Dropping a Table

The `DROP TABLE` statement removes the definition of an Oracle table. When you drop a table, the database loses all the data in the table and all the indexes associated with it.

Syntax

```
DROP TABLE table
```

In the syntax:

`table` is the name of the table

Guidelines

- All data is deleted from the table.
- Any views and synonyms remain but are invalid.
- Any pending transactions are committed.
- Only the creator of the table or a user with the `DROP ANY TABLE` privilege can remove a table.

Note: The `DROP TABLE` statement, once executed, is irreversible. The Oracle server does not question the action when you issue the `DROP TABLE` statement. If you own that table or have a high-level privilege, then the table is immediately removed. As with all DDL statements, `DROP TABLE` is committed automatically.

Changing the Name of an Object

- **To change the name of a table, view, sequence, or synonym, you execute the `RENAME` statement.**

```
RENAME dept TO detail_dept;
Table renamed.
```

- **You must be the owner of the object.**

Renaming a Table

Additional DDL statements include the `RENAME` statement, which is used to rename a table, view, sequence, or a synonym.

Syntax

`RENAME` *`old_name`* `TO` *`new_name;`*

In the syntax:

`old_name`	is the old name of the table, view, sequence, or synonym.
`new_name`	is the new name of the table, view, sequence, or synonym.

You must be the owner of the object that you rename.

Truncating a Table

- The `TRUNCATE TABLE` statement:
  - Removes all rows from a table
  - Releases the storage space used by that table

```
TRUNCATE TABLE detail_dept;
Table truncated.
```

- You cannot roll back row removal when using `TRUNCATE`.
- Alternatively, you can remove rows by using the `DELETE` statement.

ORACLE

Truncating a Table

Another DDL statement is the `TRUNCATE TABLE` statement, which is used to remove all rows from a table and to release the storage space used by that table. When using the `TRUNCATE TABLE` statement, you cannot roll back row removal.

Syntax

```
TRUNCATE  TABLE   table;
```

In the syntax:

table is the name of the table

You must be the owner of the table or have `DELETE TABLE` system privileges to truncate a table.

The `DELETE` statement can also remove all rows from a table, but it does not release storage space. The `TRUNCATE` command is faster. Removing rows with the `TRUNCATE` statement is faster than removing them with the `DELETE` statement for the following reasons:

- The `TRUNCATE` statement is a data definition language (DDL) statement and generates no rollback information.
- Truncating a table does not fire the delete triggers of the table.
- If the table is the parent of a referential integrity constraint, you cannot truncate the table. Disable the constraint before issuing the `TRUNCATE` statement.

Adding Comments to a Table

- **You can add comments to a table or column by using the `COMMENT` statement.**

```
COMMENT ON TABLE employees
IS 'Employee Information';
Comment created.
```

- **Comments can be viewed through the data dictionary views:**
  - **`ALL_COL_COMMENTS`**
  - **`USER_COL_COMMENTS`**
  - **`ALL_TAB_COMMENTS`**
  - **`USER_TAB_COMMENTS`**

ORACLE

Adding a Comment to a Table

You can add a comment of up to 2,000 bytes about a column, table, view, or snapshot by using the `COMMENT` statement. The comment is stored in the data dictionary and can be viewed in one of the following data dictionary views in the `COMMENTS` column:

- `ALL_COL_COMMENTS`
- `USER_COL_COMMENTS`
- `ALL_TAB_COMMENTS`
- `USER_TAB_COMMENTS`

Syntax

```
COMMENT ON TABLE table | COLUMN table.column
 IS 'text';
```

In the syntax:

table	is the name of the table
column	is the name of the column in a table
text	is the text of the comment

You can drop a comment from the database by setting it to empty string (`''`):

```
COMMENT ON TABLE  employees IS ' ';
```

Summary

In this lesson, you should have learned how to use DDL statements to create, alter, drop, and rename tables.

Statement	Description
`CREATE TABLE`	Creates a table
`ALTER TABLE`	Modifies table structures
`DROP TABLE`	Removes the rows and table structure
`RENAME`	Changes the name of a table, view, sequence, or synonym
`TRUNCATE`	Removes all rows from a table and releases the storage space
`COMMENT`	Adds comments to a table or view

ORACLE

Summary

In this lesson, you should have learned how to use DDL commands to create, alter, drop, and rename tables. You also learned how to truncate a table and add comments to a table.

`CREATE TABLE`

- Create a table.
- Create a table based on another table by using a subquery.

`ALTER TABLE`

- Modify table structures.
- Change column widths, change column data types, and add columns.

`DROP TABLE`

- Remove rows and a table structure.
- Once executed, this statement cannot be rolled back.

`RENAME`

- Rename a table, view, sequence, or synonym.

`TRUNCATE`

- Remove all rows from a table and release the storage space used by the table.
- The `DELETE` statement removes only rows.

`COMMENT`

- Add a comment to a table or a column.
- Query the data dictionary to view the comment.

1. Create the `DEPT` table based on the following table instance chart. Place the syntax in a script called `lab9_1.sql`, then execute the statement in the script to create the table. Confirm that the table is created.

Column Name	ID	NAME
Key Type		
Nulls/Unique		
FK Table		
FK Column		
Data type	NUMBER	VARCHAR2
Length	7	25

Name	Null?	Type
ID		NUMBER(7)
NAME		VARCHAR2(25)

2. Populate the `DEPT` table with data from the `DEPARTMENTS` table. Include only columns that you need.
3. Create the `EMP` table based on the following table instance chart. Place the syntax in a script called `lab9_3.sql`, and then execute the statement in the script to create the table. Confirm that the table is created.

Column Name	ID	LAST_NAME	FIRST_NAME	DEPT_ID
Key Type				
Nulls/Unique				
FK Table				
FK Column				
Data type	NUMBER	VARCHAR2	VARCHAR2	NUMBER
Length	7	25	25	7

Name	Null?	Type
ID		NUMBER(7)
LAST_NAME		VARCHAR2(25)
FIRST_NAME		VARCHAR2(25)
DEPT_ID		NUMBER(7)

4. Modify the EMP table to allow for longer employee last names. Confirm your modification.

Name	Null?	Type
ID		NUMBER(7)
LAST_NAME		VARCHAR2(50)
FIRST_NAME		VARCHAR2(25)
DEPT_ID		NUMBER(7)

5. Confirm that both the DEPT and EMP tables are stored in the data dictionary. (*Hint:* USER_TABLES)

TABLE_NAME
DEPT
EMP

6. Create the EMPLOYEES2 table based on the structure of the EMPLOYEES table. Include only the EMPLOYEE_ID, FIRST_NAME, LAST_NAME, SALARY, and DEPARTMENT_ID columns. Name the columns in your new table ID, FIRST_NAME, LAST_NAME, SALARY , and DEPT_ID, respectively.
7. Drop the EMP table.
8. Rename the EMPLOYEES2 table as EMP.
9. Add a comment to the DEPT and EMP table definitions describing the tables. Confirm your additions in the data dictionary.
10. Drop the FIRST_NAME column from the EMP table. Confirm your modification by checking the description of the table.
11. In the EMP table, mark the DEPT_ID column in the EMP table as UNUSED. Confirm your modification by checking the description of the table.
12. Drop all the UNUSED columns from the EMP table. Confirm your modification by checking the description of the table.

Chapter 10: Including Constraints

What are Constraints?
Constraint Guidelines
Defining Constraints
The NOT NULL Constraint
The UNIQUE Constraint
The PRIMARY KEY Constraint
The FOREIGN KEY Constraint
FOREIGN KEY Constraint Keywords
The CHECK Constraint
Adding a Constraint Syntax
Adding a Constraint
Dropping a Constraint
Disabling Constraints
Enabling Constraints
Cascading Constraints
Viewing Constraints
Viewing the Columns Associated with Constraints
Summary

What are Constraints?

- **Constraints enforce rules at the table level.**
- **Constraints prevent the deletion of a table if there are dependencies.**
- **The following constraint types are valid:**
  - **`NOT NULL`**
  - **`UNIQUE`**
  - **`PRIMARY KEY`**
  - **`FOREIGN KEY`**
  - **`CHECK`**

ORACLE

Constraints

The Oracle Server uses *constraints* to prevent invalid data entry into tables.

You can use constraints to do the following:

- Enforce rules on the data in a table whenever a row is inserted, updated, or deleted from that table. The constraint must be satisfied for the operation to succeed.
- Prevent the deletion of a table if there are dependencies from other tables
- Provide rules for Oracle tools, such as Oracle Developer

Data Integrity Constraints

Constraint	**Description**
`NOT NULL`	Specifies that the column cannot contain a null value
`UNIQUE`	Specifies a column or combination of columns whose values must be unique for all rows in the table
`PRIMARY KEY`	Uniquely identifies each row of the table
`FOREIGN KEY`	Establishes and enforces a foreign key relationship between the column and a column of the referenced table
`CHECK`	Specifies a condition that must be true

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

Constraint Guidelines

- **Name a constraint or the Oracle server generates a name by using the `SYS_C`*n* format.**
- **Create a constraint either:**
  - **At the same time as the table is created, or**
  - **After the table has been created**
- **Define a constraint at the column or table level.**
- **View a constraint in the data dictionary.**

Constraint Guidelines

All constraints are stored in the data dictionary. Constraints are easy to reference if you give them a meaningful name. Constraint names must follow the standard object-naming rules. If you do not name your constraint, the Oracle server generates a name with the format `SYS_C`*n*, where *n* is an integer so that the constraint name is unique.

Constraints can be defined at the time of table creation or after the table has been created.

You can view the constraints defined for a specific table by looking at the `USER_CONSTRAINTS` data dictionary table.

Defining Constraints

```
CREATE TABLE [schema.]table
            (column datatype [DEFAULT expr]
             [column_constraint],
             ...
             [table_constraint][,...]);
```

```
CREATE TABLE employees(
             employee_id  NUMBER(6),
             first_name   VARCHAR2(20),
             ...
             job_id       VARCHAR2(10) NOT NULL,
             CONSTRAINT emp_emp_id_pk
                          PRIMARY KEY (EMPLOYEE_ID));
```

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 to use if a value is omitted in the INSERT statement
column	is the name of the column
datatype	is the column's data type 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 *Oracle9i SQL Reference*, "CREATE TABLE."

Defining Constraints

- **Column constraint level**

```
column [CONSTRAINT constraint_name] constraint_type,
```

- **Table constraint level**

```
column,...
  [CONSTRAINT constraint_name] constraint_type
  (column, ...),
```

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:

EMPLOYEE_ID	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	DEPARTMENT_ID
100	King	SKING	515.123.4567	17-JUN-87	AD_PRES	24000	90
101	Kochhar	NKOCHHAR	515.123.4568	21-SEP-89	AD_VP	17000	90
102	De Haan	LDEHAAN	515.123.4569	13-JAN-93	AD_VP	17000	90
103	Hunold	AHUNOLD	590.423.4567	03-JAN-90	IT_PROG	9000	60
104	Ernst	BERNST	590.423.4568	21-MAY-91	IT_PROG	6000	60
178	Grant	KGRANT	011.44.1644.429263	24-MAY-99	SA_REP	7000	
200	Whalen	JWHALEN	515.123.4444	17-SEP-87	AD_ASST	4400	10

...

20 rows selected.

NOT NULL constraint (No row can contain a null value for this column.)

NOT NULL constraint

Absence of NOT NULL constraint (Any row can contain null for this column.)

ORACLE

The NOT NULL Constraint

The NOT NULL constraint ensures that the column contains no null values. Columns without the NOT NULL constraint can contain null values by default.

The NOT NULL Constraint

Is defined at the column level:

```
CREATE TABLE employees(
    employee_id    NUMBER(6),
    last_name      VARCHAR2(25) NOT NULL,
    salary         NUMBER(8,2),
    commission_pct NUMBER(2,2),
    hire_date      DATE
                   CONSTRAINT emp_hire_date_nn
                   NOT NULL,
...
```

System named

User named

ORACLE

The `NOT NULL` Constraint (continued)

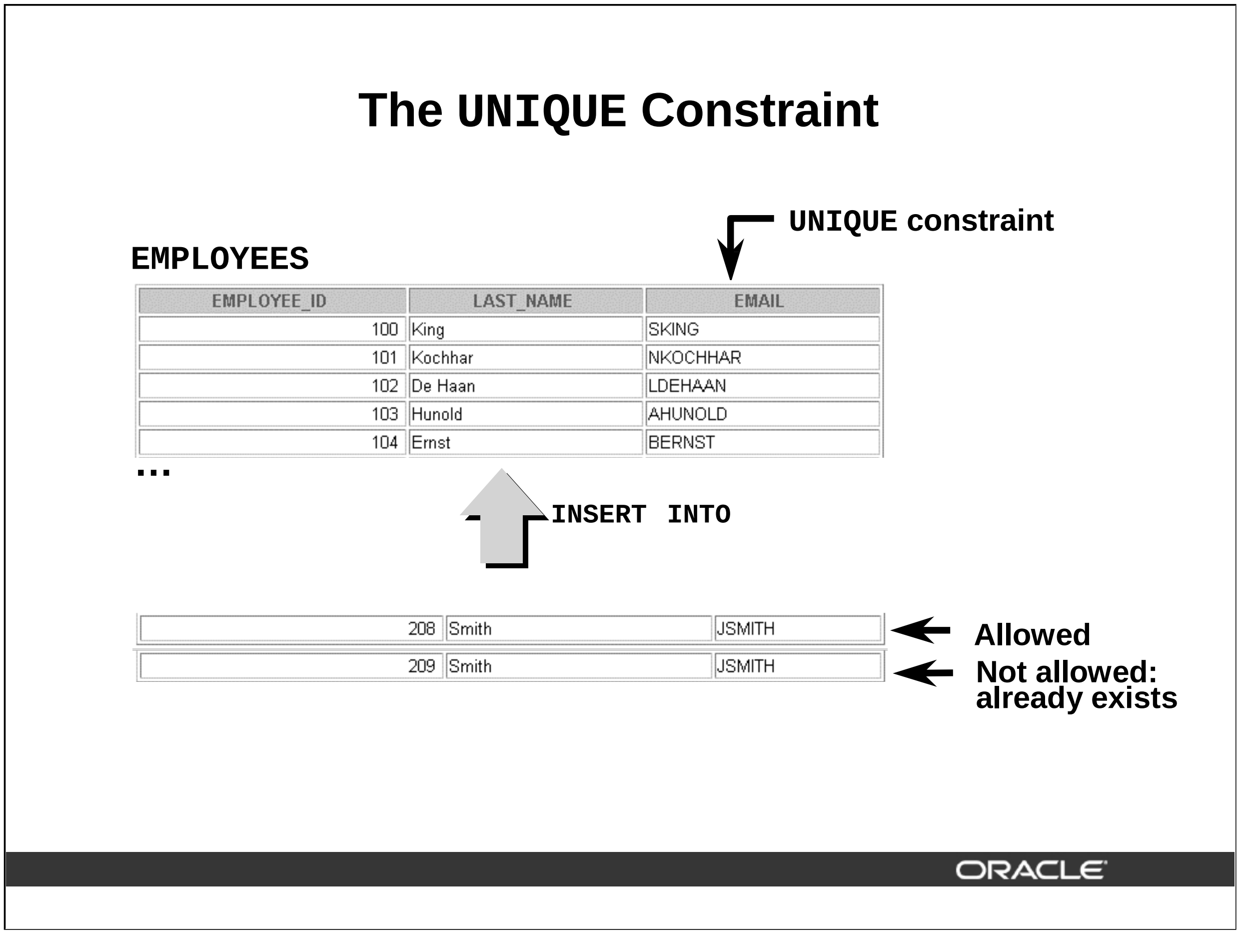
The `NOT NULL` constraint can be specified only at the column level, not at the table level.

The slide example applies the `NOT NULL` constraint to the `LAST_NAME` and `HIRE_DATE` columns of the `EMPLOYEES` table. Because these constraints are unnamed, the Oracle server creates names for them.

You can specify the name of the constraint when you specify the constraint:

```
... last_name VARCHAR2(25)
      CONSTRAINT emp_last_name_nn NOT NULL...
```

Note: The constraint examples described in this lesson may not be present in the sample tables provided with the course. If desired, these constraints can be added to the tables.



The UNIQUE Constraint

A UNIQUE key integrity constraint requires that every value in a column or set of columns (key) be unique—that is, no two rows of a table can have duplicate values in a specified column or set of columns. The column (or set of columns) included in the definition of the UNIQUE key constraint is called the *unique key*. If the UNIQUE constraint comprises more than one column, that group of columns is called a *composite unique key*.

UNIQUE constraints allow the input of nulls unless you also define NOT NULL constraints for the same columns. In fact, any number of rows can include nulls for columns without NOT NULL constraints because nulls are not considered equal to anything. A null in a column (or in all columns of a composite UNIQUE key) always satisfies a UNIQUE constraint.

Note: Because of the search mechanism for UNIQUE constraints on more than one column, you cannot have identical values in the non-null columns of a partially null composite UNIQUE key constraint.

The UNIQUE Constraint

Defined at either the table level or the column level:

```
CREATE TABLE employees(
    employee_id      NUMBER(6),
    last_name        VARCHAR2(25) NOT NULL,
    email            VARCHAR2(25),
    salary           NUMBER(8,2),
    commission_pct   NUMBER(2,2),
    hire_date        DATE NOT NULL,
...
    CONSTRAINT emp_email_uk UNIQUE(email));
```

ORACLE

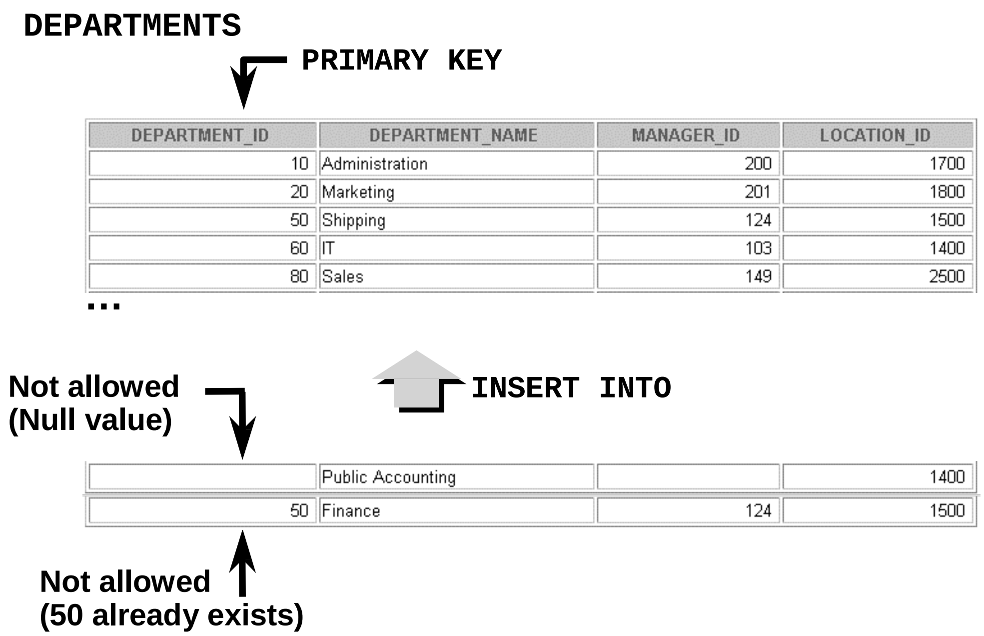
The UNIQUE Constraint (continued)

UNIQUE constraints can be defined at the column or table level. A composite unique key is created by using the table level definition.

The example on the slide applies the UNIQUE constraint to the EMAIL column of the EMPLOYEES table. The name of the constraint is EMP_EMAIL_UK..

Note: The Oracle server enforces the UNIQUE constraint by implicitly creating a unique index on the unique key column or columns.

The PRIMARY KEY Constraint



DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
50	Shipping	124	1500
60	IT	103	1400
80	Sales	149	2500

...

	Public Accounting		1400
50	Finance	124	1500

The PRIMARY KEY Constraint

A `PRIMARY KEY` constraint creates a primary key for the table. Only one primary key can be created for each table. The `PRIMARY KEY` constraint is a column or set of columns that uniquely identifies each row in a table. This constraint enforces uniqueness of the column or column combination and ensures that no column that is part of the primary key can contain a null value.

The PRIMARY KEY Constraint

Defined at either the table level or the column level:

```
CREATE TABLE   departments(
    department_id        NUMBER(4),
    department_name      VARCHAR2(30)
      CONSTRAINT dept_name_nn NOT NULL,
    manager_id           NUMBER(6),
    location_id          NUMBER(4),
      CONSTRAINT dept_id_pk PRIMARY KEY(department_id));
```

ORACLE

The PRIMARY KEY Constraint (continued)

PRIMARY KEY constraints can be defined at the column level or table level. A composite PRIMARY KEY is created by using the table-level definition.

A table can have only one PRIMARY KEY constraint but can have several UNIQUE constraints.

The example on the slide defines a PRIMARY KEY constraint on the DEPARTMENT_ID column of the DEPARTMENTS table. The name of the constraint is DEPT_ID_PK.

Note: A UNIQUE index is automatically created for a PRIMARY KEY column.

The FOREIGN KEY Constraint

DEPARTMENTS

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
50	Shipping	124	1500
60	IT	103	1400
80	Sales	149	2500

...

PRIMARY KEY

EMPLOYEES

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
100	King	90
101	Kochhar	90
102	De Haan	90
103	Hunold	60
104	Ernst	60
107	Lorentz	60

...

FOREIGN KEY

INSERT INTO

200	Ford	9
201	Ford	60

Not allowed (9 does not exist)

Allowed

ORACLE

The FOREIGN KEY Constraint

The `FOREIGN KEY`, or referential integrity constraint, designates a column or combination of columns as a foreign key and establishes a relationship between a primary key or a unique key in the same table or a different table. In the example on the slide, `DEPARTMENT_ID` has been defined as the foreign key in the `EMPLOYEES` table (dependent or child table); it references the `DEPARTMENT_ID` column of the `DEPARTMENTS` table (the referenced or parent table).

A foreign key value must match an existing value in the parent table or be `NULL`.

Foreign keys are based on data values and are purely logical, not physical, pointers.

The FOREIGN KEY Constraint

Defined at either the table level or the column level:

```
CREATE TABLE employees(
    employee_id      NUMBER(6),
    last_name        VARCHAR2(25) NOT NULL,
    email            VARCHAR2(25),
    salary           NUMBER(8,2),
    commission_pct   NUMBER(2,2),
    hire_date        DATE NOT NULL,
...
    department_id    NUMBER(4),
    CONSTRAINT emp_dept_fk FOREIGN KEY (department_id)
      REFERENCES departments(department_id),
    CONSTRAINT emp_email_uk UNIQUE(email));
```

ORACLE

The FOREIGN KEY Constraint (continued)

FOREIGN KEY constraints can be defined at the column or table constraint level. A composite foreign key must be created by using the table-level definition.

The example on the slide defines a FOREIGN KEY constraint on the DEPARTMENT_ID column of the EMPLOYEES table, using table-level syntax. The name of the constraint is EMP_DEPTID_FK.

The foreign key can also be defined at the column level, provided the constraint is based on a single column. The syntax differs in that the keywords FOREIGN KEY do not appear. For example:

```
CREATE TABLE employees
(...
department_id NUMBER(4) CONSTRAINT emp_deptid_fk
    REFERENCES departments(department_id),
...
)
```

FOREIGN KEY Constraint Keywords

- **FOREIGN KEY: Defines the column in the child table at the table constraint level**
- **REFERENCES: Identifies the table and column in the parent table**
- **ON DELETE CASCADE: Deletes the dependent rows in the child table when a row in the parent table is deleted.**
- **ON DELETE SET NULL: Converts dependent foreign key values to null**

The FOREIGN KEY Constraint (continued)

The foreign key is defined in the child table, and the table containing the referenced column is the parent table. The foreign key is defined using a combination of the following keywords:

- `FOREIGN KEY` is used to define the column in the child table at the table constraint level.
- `REFERENCES` identifies the table and column in the parent table.
- `ON DELETE CASCADE` indicates that when the row in the parent table is deleted, the dependent rows in the child table will also be deleted.
- `ON DELETE SET NULL` converts foreign key values to null when the parent value is removed.

The default behavior is called the restrict rule, which disallows the update or deletion of referenced data.

Without the `ON DELETE CASCADE` or the `ON DELETE SET NULL` options, the row in the parent table cannot be deleted if it is referenced in the child table.

The CHECK Constraint

- **Defines a condition that each row must satisfy**
- **The following expressions are not allowed:**
  - **References to CURRVAL, NEXTVAL, LEVEL, and ROWNUM pseudocolumns**
  - **Calls to SYSDATE, UID, USER, and USERENV functions**
  - **Queries that refer to other values in other rows**

```
..., salary  NUMBER(2)
     CONSTRAINT emp_salary_min
            CHECK (salary > 0),...
```

ORACLE

The CHECK Constraint

The CHECK constraint defines a condition that each row must satisfy. The condition can use the same constructs as query conditions, with the following exceptions:

- References to the CURRVAL, NEXTVAL, LEVEL, and ROWNUM pseudocolumns
- Calls to SYSDATE, UID, USER, and USERENV functions
- Queries that refer to other values in other rows

A single column can have multiple CHECK constraints which refer to the column in its definition. There is no limit to the number of CHECK constraints which you can define on a column.

CHECK constraints can be defined at the column level or table level.

```
CREATE TABLE employees
      (...
       salary NUMBER(8,2) CONSTRAINT emp_salary_min
                          CHECK (salary > 0),
      ...
```

Adding a Constraint Syntax

Use the `ALTER TABLE` statement to:

- **Add or drop a constraint, but not modify its structure**
- **Enable or disable constraints**
- **Add a `NOT NULL` constraint by using the `MODIFY` clause**

```
ALTER TABLE table
ADD [CONSTRAINT constraint] type (column);
```

ORACLE

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 is empty or if the column has a value for every row.

Adding a Constraint

Add a `FOREIGN KEY` constraint to the `EMPLOYEES` table indicating that a manager must already exist as a valid employee in the `EMPLOYEES` table.

```
ALTER TABLE     employees
ADD CONSTRAINT  emp_manager_fk
  FOREIGN KEY(manager_id)
  REFERENCES employees(employee_id);
Table altered.
```

ORACLE

Adding a Constraint (continued)

The example on the slide creates a `FOREIGN KEY` constraint on the `EMPLOYEES` table. The constraint ensures that a manager exists as a valid employee in the `EMPLOYEES` table.

Dropping a Constraint

- **Remove the manager constraint from the `EMPLOYEES` table.**

```
ALTER TABLE      employees
DROP CONSTRAINT  emp_manager_fk;
Table altered.
```

- **Remove the `PRIMARY KEY` constraint on the `DEPARTMENTS` table and drop the associated `FOREIGN KEY` constraint on the `EMPLOYEES.DEPARTMENT_ID` column.**

```
ALTER TABLE   departments
DROP PRIMARY KEY CASCADE;
Table altered.
```

ORACLE

Dropping a Constraint

To drop a constraint, you can identify the constraint name from the `USER_CONSTRAINTS` and `USER_CONS_COLUMNS` data dictionary views. Then use the `ALTER TABLE` statement with the `DROP` clause. The `CASCADE` option of the `DROP` clause causes any dependent constraints also to be dropped.

Syntax

```
ALTER TABLE  table
 DROP  PRIMARY KEY | UNIQUE (column) |
       CONSTRAINT   constraint  [CASCADE];
```

In the syntax:

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.**

```
ALTER TABLE          employees
DISABLE CONSTRAINT   emp_emp_id_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];
```

In the syntax:

table	is the name of the table
constraint	is the name of the constraint

Guidelines

- You can use the DISABLE clause in both the CREATE TABLE statement and the ALTER TABLE statement.
- The CASCADE clause disables dependent integrity constraints.
- Disabling a unique or primary key constraint removes the unique index.

Enabling Constraints

- Activate an integrity constraint currently disabled in the table definition by using the `ENABLE` clause.

```
ALTER TABLE          employees
ENABLE CONSTRAINT    emp_emp_id_pk;
Table altered.
```

- A `UNIQUE` or `PRIMARY KEY` index is automatically created if you enable a `UNIQUE` key or `PRIMARY KEY` constraint.

ORACLE

Enabling a Constraint

You can enable a constraint without dropping it or re-creating it by using the `ALTER TABLE` statement with the `ENABLE` clause.

Syntax

```
ALTER   TABLE      table
ENABLE  CONSTRAINT constraint;
```

In the syntax:

table	is the name of the table
constraint	is the name of the constraint

Guidelines

- If you enable a constraint, that constraint applies to all the data in the table. All the data in the table must fit the constraint.
- If you enable a `UNIQUE` key or `PRIMARY KEY` constraint, a `UNIQUE` or `PRIMARY KEY` index is created automatically.
- You can use the `ENABLE` clause in both the `CREATE TABLE` statement and the `ALTER TABLE` statement.
- Enabling a primary key constraint that was disabled with the `CASCADE` option does not enable any foreign keys that are dependent upon the primary key.

Cascading Constraints

- The `CASCADE CONSTRAINTS` clause is used along with the `DROP COLUMN` clause.
- The `CASCADE CONSTRAINTS` clause drops all referential integrity constraints that refer to the primary and unique keys defined on the dropped columns.
- The `CASCADE CONSTRAINTS` clause also drops all multicolumn constraints defined on the dropped columns.

ORACLE

Cascading Constraints

This statement illustrates the usage of the `CASCADE CONSTRAINTS` clause. Assume table `TEST1` is created as follows:

```
CREATE TABLE test1 (
  pk NUMBER PRIMARY KEY,
  fk NUMBER,
  col1 NUMBER,
  col2 NUMBER,
  CONSTRAINT fk_constraint FOREIGN KEY (fk) REFERENCES test1,
  CONSTRAINT ck1 CHECK (pk > 0 and col1 > 0),
  CONSTRAINT ck2 CHECK (col2 > 0));
```

An error is returned for the following statements:

```
ALTER TABLE test1 DROP (pk);   -- pk is a parent key
ALTER TABLE test1 DROP (col1); -- col1 is referenced by multicolumn constraint ck1
```

Cascading Constraints

Example:

```
ALTER TABLE test1
DROP (pk) CASCADE CONSTRAINTS;
Table altered.
```

```
ALTER TABLE test1
DROP (pk, fk, col1) CASCADE CONSTRAINTS;
Table altered.
```

ORACLE

Cascading Constraints (continued)

Submitting the following statement drops column `PK`, the primary key constraint, the `fk_constraint` foreign key constraint, and the check constraint, `CK1`:

```
ALTER TABLE test1 DROP (pk) CASCADE CONSTRAINTS;
```

If all columns referenced by the constraints defined on the dropped columns are also dropped, then `CASCADE CONSTRAINTS` is not required. For example, assuming that no other referential constraints from other tables refer to column `PK`, it is valid to submit the following statement without the `CASCADE CONSTRAINTS` clause:

```
ALTER TABLE test1 DROP (pk, fk, col1);
```

Viewing Constraints

Query the USER_CONSTRAINTS table to view all constraint definitions and names.

```
SELECT   constraint_name, constraint_type,
         search_condition
FROM     user_constraints
WHERE    table_name = 'EMPLOYEES';
```

CONSTRAINT_NAME	C	SEARCH_CONDITION
EMP_LAST_NAME_NN	C	"LAST_NAME" IS NOT NULL
EMP_EMAIL_NN	C	"EMAIL" IS NOT NULL
EMP_HIRE_DATE_NN	C	"HIRE_DATE" IS NOT NULL
EMP_JOB_NN	C	"JOB_ID" IS NOT NULL
EMP_SALARY_MIN	C	salary > 0
EMP_EMAIL_UK	U	

...

ORACLE

Viewing Constraints

After creating a table, you can confirm its existence by issuing a `DESCRIBE` command. The only constraint that you can verify is the `NOT NULL` constraint. To view all constraints on your table, query the `USER_CONSTRAINTS` table.

The example on the slide displays the constraints on the `EMPLOYEES` table.

Note: Constraints that are not named by the table owner receive the system-assigned constraint name. In constraint type, `C` stands for `CHECK`, `P` for `PRIMARY KEY`, `R` for referential integrity, and `U` for `UNIQUE` key. Notice that the `NOT NULL` constraint is really a `CHECK` constraint.

Viewing the Columns Associated with Constraints

View the columns associated with the constraint names in the `USER_CONS_COLUMNS` view.

```
SELECT   constraint_name, column_name
FROM     user_cons_columns
WHERE    table_name = 'EMPLOYEES';
```

CONSTRAINT_NAME	COLUMN_NAME
EMP_DEPT_FK	DEPARTMENT_ID
EMP_EMAIL_NN	EMAIL
EMP_EMAIL_UK	EMAIL
EMP_EMP_ID_PK	EMPLOYEE_ID
EMP_HIRE_DATE_NN	HIRE_DATE
EMP_JOB_FK	JOB_ID
EMP_JOB_NN	JOB_ID

...

ORACLE

Viewing Constraints (continued)

You can view the names of the columns involved in constraints by querying the `USER_CONS_COLUMNS` data dictionary view. This view is especially useful for constraints that use system-assigned names.

Summary

In this lesson, you should have learned how to create constraints.

- **Types of constraints:**
  - **`NOT NULL`**
  - **`UNIQUE`**
  - **`PRIMARY KEY`**
  - **`FOREIGN KEY`**
  - **`CHECK`**
- **You can query the `USER_CONSTRAINTS` table to view all constraint definitions and names.**

ORACLE

Summary

In this lesson, you should have learned how the Oracle server uses constraints to prevent invalid data entry into tables. You also learned how to implement the constraints in DDL statements.

The following constraint types are valid:

- `NOT NULL`
- `UNIQUE`
- `PRIMARY KEY`
- `FOREIGN KEY`
- `CHECK`

You can query the `USER_CONSTRAINTS` table to view all constraint definitions and names.

1. Add a table-level PRIMARY KEY constraint to the EMP table on the ID column. The constraint should be named at creation. Name the constraint my_emp_id_pk.

   Hint: The constraint is enabled as soon as the ALTER TABLE command executes successfully.

2. Create a PRIMARY KEY constraint to the DEPT table using the ID column. The constraint should be named at creation. Name the constraint my_dept_id_pk.

   Hint: The constraint is enabled as soon as the ALTER TABLE command executes successfully.

3. Add a column DEPT_ID to the EMP table. Add a foreign key reference on the EMP table that ensures that the employee is not assigned to a nonexistent department. Name the constraint my_emp_dept_id_fk.

4. Confirm that the constraints were added by querying the USER_CONSTRAINTS view. Note the types and names of the constraints. Save your statement text in a file called lab10_4.sql.

CONSTRAINT_NAME	C
MY_DEPT_ID_PK	P
SYS_C002541	C
MY_EMP_ID_PK	P
MY_EMP_DEPT_ID_FK	R

5. Display the object names and types from the USER_OBJECTS data dictionary view for the
   EMP and DEPT tables. Notice that the new tables and a new index were created.

If you have time, complete the following exercise:

6. Modify the EMP table. Add a COMMISSION column of NUMBER data type, precision 2, scale
   2. Add a constraint to the commission column that ensures that a commission value is greater than zero.

What is a View?

EMPLOYEES Table:

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALA
100	Steven	King	SKING	515.123.4567	17-JUN-87	AD_PRES	240
101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-SEP-89	AD_VP	170
102	Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-93	AD_VP	170
103	Alexander	Hunold	AHUNOLD	590.423.4567	03-JAN-90	IT_PROG	90
104	Bruce	Ernst	BERNST	590.423.4568	21-MAY-91	IT_PROG	60
107	Diana	Lorentz	DLORENTZ	590.423.5567	07-FEB-99	IT_PROG	42
124	Kevin	Mourgos	KMOURGOS	650.123.5234	16-NOV-99	ST_MAN	58
141	Trenna	Rajs	TRAJS	650.121.8009	17-OCT-95	ST_CLERK	35
142	Curtis	Davies	CDAVIES	650.121.2994	29-JAN-97	ST_CLERK	31
143	Randall	Matos	RMATOS	650.121.2874	15-MAR-98	ST_CLERK	26
					JUL-98	ST_CLERK	25
					JAN-00	SA_MAN	105
					MAY-96	SA_REP	110
					MAR-98	SA_REP	86
178	Kimberely	Grant	KGRANT	011.44.1644.429263	24-MAY-99	SA_REP	70
200	Jennifer	Whalen	JWHALEN	515.123.4444	17-SEP-87	AD_ASST	44
201	Michael	Hartstein	MHARTSTE	515.123.5555	17-FEB-96	MK_MAN	130
202	Pat	Fay	PFAY	603.123.6666	17-AUG-97	MK_REP	60
205	Shelley	Higgins	SHIGGINS	515.123.8080	07-JUN-94	AC_MGR	120
206	William	Gietz	WGIETZ	515.123.8181	07-JUN-94	AC_ACCOUNT	83

EMPLOYEE_ID	LAST_NAME	SALARY
149	Zlotkey	10500
174	Abel	11000
176	Taylor	8600

20 rows selected.

ORACLE

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 provide 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 can be used to make simple queries to retrieve the results of complicated queries. For example, views can be used to query information from multiple tables without the user knowing how to write a join statement.
- Views provide data independence for ad hoc users and application programs. One view can be used to retrieve data from several tables.
- Views provide groups of users access to data according to their particular criteria.

Simple Views and Complex Views

Feature	Simple Views	Complex Views
Number of tables	One	One or more
Contain functions	No	Yes
Contain groups of data	No	Yes
DML operations through a view	Yes	Not always

ORACLE

Simple Views versus Complex Views

There are two classifications for views: simple and complex. The basic difference is related to the DML (`INSERT`, `UPDATE`, and `DELETE`) operations.

- A simple view is one that:
  - Derives data from only one table
  - Contains no functions or groups of data
  - Can perform DML operations through the view
- A complex view is one that:
  - Derives data from many tables
  - Contains functions or groups of data
  - Does not always allow DML operations through the view

Creating a View

- **You embed a subquery within the `CREATE VIEW` statement.**

```
CREATE [OR REPLACE] [FORCE|NOFORCE] VIEW view
  [(alias[, alias]...)]
 AS subquery
[WITH CHECK OPTION [CONSTRAINT constraint]]
[WITH READ ONLY [CONSTRAINT constraint]];
```

- **The subquery can contain complex `SELECT` syntax.**

ORACLE

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, `EMPVU80`, that contains details of employees in department 80.**

```
CREATE VIEW  empvu80
 AS SELECT  employee_id, last_name, salary
    FROM    employees
    WHERE   department_id = 80;
View created.
```

- **Describe the structure of the view by using the *i*SQL*Plus `DESCRIBE` command.**

```
DESCRIBE empvu80
```

Creating a View (continued)

The example on the slide creates a view that contains the employee number, last name, and salary for each employee in department 80.

You can display the structure of the view by using the *i*SQL*Plus `DESCRIBE` command.

Name	Null?	Type
EMPLOYEE_ID	NOT NULL	NUMBER(6)
LAST_NAME	NOT NULL	VARCHAR2(25)
SALARY		NUMBER(8,2)

Guidelines for creating a view:

- The subquery that defines a view can contain complex `SELECT` syntax, including joins, groups, and subqueries.
- The subquery that defines the view cannot contain an `ORDER BY` clause. The `ORDER BY` clause is specified when you retrieve data from the view.
- If you do not specify a constraint name for a view created with the `WITH CHECK OPTION`, the system assigns a default name in the format `SYS_C`*n*.
- You can use the `OR REPLACE` option to change the definition of the view without dropping and re-creating it or regranting object privileges previously granted on it.

Creating a View

- **Create a view by using column aliases in the subquery.**

```
CREATE VIEW  salvu50
 AS SELECT  employee_id ID_NUMBER, last_name NAME,
            salary*12 ANN_SALARY
    FROM    employees
    WHERE   department_id = 50;
View created.
```

- **Select the columns from this view by the given alias names.**

ORACLE

Creating a View (continued)

You can control the column names by including column aliases within the subquery.

The example on the slide creates a view containing the employee number (`EMPLOYEE_ID`) with the alias `ID_NUMBER`, name (`LAST_NAME`) with the alias `NAME`, and annual salary (`SALARY`) with the alias `ANN_SALARY` for every employee in department 50.

As an alternative, you can use an alias after the `CREATE` statement and prior to the `SELECT` subquery. The number of aliases listed must match the number of expressions selected in the subquery.

```
CREATE VIEW   salvu50 (ID_NUMBER, NAME, ANN_SALARY)
  AS SELECT  employee_id, last_name, salary*12
     FROM    employees
     WHERE   department_id = 50;
View created.
```

Retrieving Data from a View

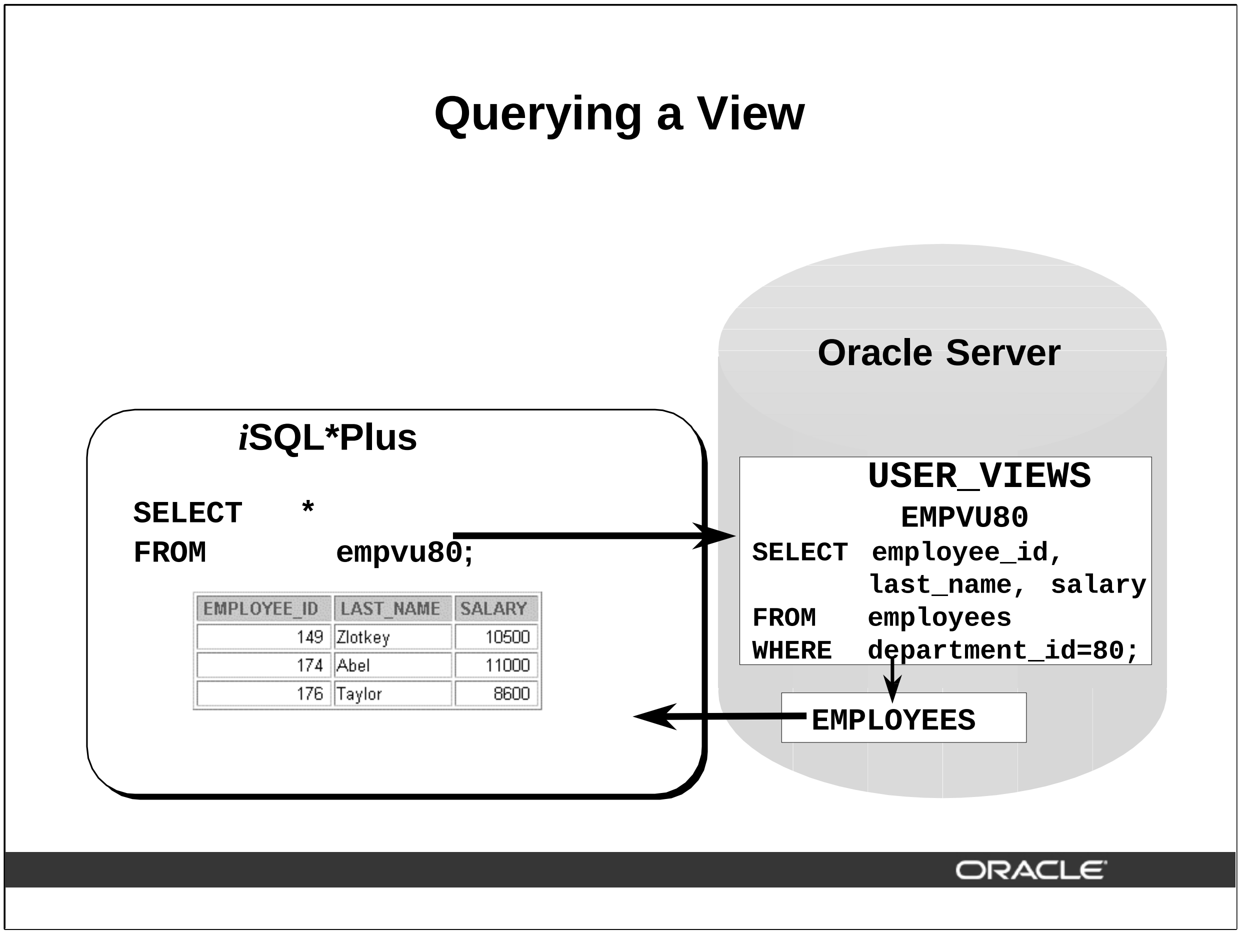
```
SELECT *
FROM salvu50;
```

ID_NUMBER	NAME	ANN_SALARY
124	Mourgos	69600
141	Rajs	42000
142	Davies	37200
143	Matos	31200
144	Vargas	30000

ORACLE

Retrieving Data from a View

You can retrieve data from a view as you would from any table. You can display either the contents of the entire view or just specific rows and columns.



Views in the Data Dictionary

Once your view has been created, you can query the data dictionary view called `USER_VIEWS` to see the name of the view and the view definition. The text of the `SELECT` statement that constitutes your view is stored in a `LONG` column.

Data Access Using Views

When you access data using a view, the Oracle server performs the following operations:

1. It retrieves the view definition from the data dictionary table `USER_VIEWS`.
2. It checks access privileges for the view base table.
3. It converts the view query into an equivalent operation on the underlying base table or tables. In other words, data is retrieved from, or an update is made to, the base tables.

Modifying a View

- **Modify the `EMPVU80` view by using `CREATE OR REPLACE VIEW` clause. Add an alias for each column name.**

```
CREATE OR REPLACE VIEW empvu80
  (id_number, name, sal, department_id)
AS SELECT  employee_id, first_name || ’ ’ || last_name,
           salary, department_id
   FROM    employees
   WHERE   department_id = 80;
View created.
```

- **Column aliases in the `CREATE VIEW` clause are listed in the same order as the columns in the subquery.**

ORACLE

Modifying a View

With the `OR REPLACE` option, a view can be created even if one exists with this name already, thus replacing the old version of the view for its owner. This means that the view can be altered without dropping, re-creating, and regranting object privileges.

Note: When assigning column aliases in the `CREATE VIEW` clause, remember that the aliases are listed in the same order as the columns in the subquery.

Creating a Complex View

Create a complex view that contains group functions to display values from two tables.

```
CREATE VIEW dept_sum_vu
  (name, minsal, maxsal, avgsal)
AS SELECT    d.department_name, MIN(e.salary),
             MAX(e.salary),AVG(e.salary)
   FROM      employees e, departments d
   WHERE     e.department_id = d.department_id
   GROUP BY  d.department_name;
View created.
```

ORACLE

Creating a Complex View

The example on the slide creates a complex view of department names, minimum salaries, maximum salaries, and average salaries by department. Note that alternative names have been specified for the view. This is a requirement if any column of the view is derived from a function or an expression.

You can view the structure of the view by using the *i*SQL*Plus `DESCRIBE` command. Display the contents of the view by issuing a `SELECT` statement.

```
SELECT   *
FROM     dept_sum_vu;
```

NAME	MINSAL	MAXSAL	AVGSAL
Accounting	8300	12000	10150
Administration	4400	4400	4400
Executive	17000	24000	19333.3333
IT	4200	9000	6400
Marketing	6000	13000	9500
Sales	8600	11000	10033.3333
Shipping	2500	5800	3500

7 rows selected.

Rules for Performing DML Operations on a View

- **You can perform DML operations on simple views.**
- **You cannot remove a row if the view contains the following:**
  - **Group functions**
  - **A `GROUP BY` clause**
  - **The `DISTINCT` keyword**
  - **The pseudocolumn `ROWNUM` keyword**

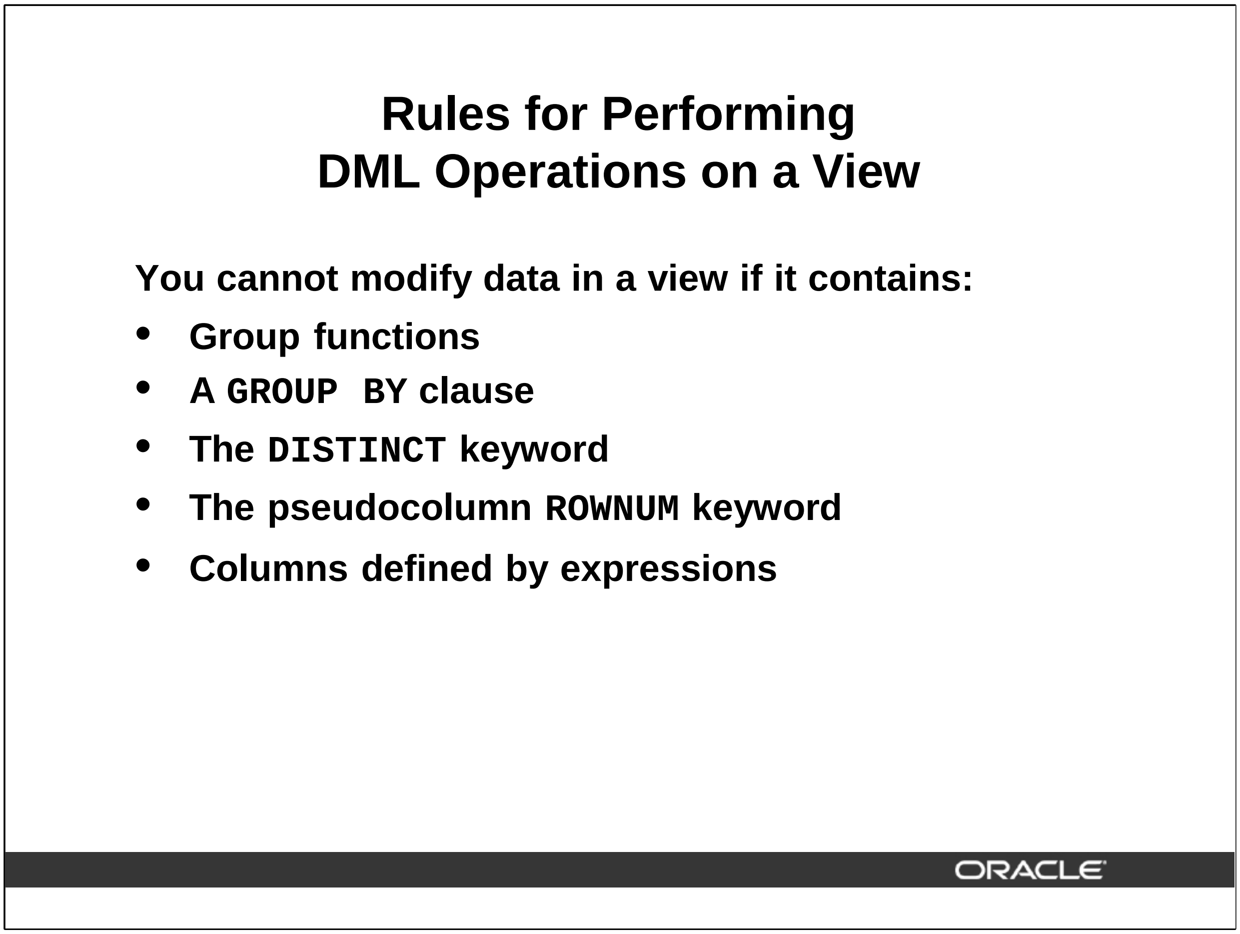
ORACLE

Performing DML Operations on a View

You can perform DML operations on data through a view if those operations follow certain rules.

You can remove a row from a view unless it contains any of the following:

- Group functions
- A `GROUP BY` clause
- The `DISTINCT` keyword
- The pseudocolumn `ROWNUM` keyword



Performing DML Operations on a View (continued)

You can modify data through a view unless it contains any of the conditions mentioned in the previous slide or columns defined by expressions—for example, `SALARY * 12`.

Rules for Performing DML Operations on a View

You cannot add data through a view if the view includes:

- **Group functions**
- **A `GROUP BY` clause**
- **The `DISTINCT` keyword**
- **The pseudocolumn `ROWNUM` keyword**
- **Columns defined by expressions**
- **`NOT NULL` columns in the base tables that are not selected by the view**

Performing DML Operations on a View (continued)

You can add data through a view unless it contains any of the items listed in the slide or there are `NOT NULL` columns without default values in the base table that are not selected by the view. All required values must be present in the view. Remember that you are adding values directly into the underlying table *through* the view.

Using the WITH CHECK OPTION Clause

- **You can ensure that DML operations performed on the view stay within the domain of the view by using the `WITH CHECK OPTION` clause.**

```
CREATE OR REPLACE VIEW empvu20
AS SELECT   *
   FROM     employees
   WHERE    department_id = 20
   WITH CHECK OPTION CONSTRAINT empvu20_ck ;
View created.
```

- **Any attempt to change the department number for any row in the view fails because it violates the `WITH CHECK OPTION` constraint.**

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 `INSERT`s and `UPDATE`s performed through the view cannot create rows which the view cannot select, and therefore it allows integrity constraints and data validation checks to be enforced on data being inserted or updated.

If there is an attempt to perform DML operations on rows that the view has not selected, an error is displayed, with the constraint name if that has been specified.

```
UPDATE empvu20
  SET    department_id = 10
  WHERE  employee_id = 201;
UPDATE empvu20
       *
ERROR at line 1:
ORA-01402: view WITH CHECK OPTION where-clause violation
```

Note: No rows are updated because if the department number were to change to 10, the view would no longer be able to see that employee. Therefore, with the `WITH CHECK OPTION` clause, the view can see only employees in department 20 and does not allow the department number for those employees to be changed through the view.

Denying DML Operations

- **You can ensure that no DML operations occur by adding the `WITH READ ONLY` option to your view definition.**
- **Any attempt to perform a DML on any row in the view results in an Oracle server error.**

ORACLE

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.

Denying DML Operations

```
CREATE OR REPLACE VIEW empvu10
    (employee_number, employee_name, job_title)
AS SELECT   employee_id, last_name, job_id
   FROM     employees
   WHERE    department_id = 10
   WITH READ ONLY;
View created.
```

ORACLE

Denying DML Operations

Any attempts to remove a row from a view with a read-only constraint results in an error.

```
DELETE FROM empvu10
WHERE  employee_number = 200;
DELETE FROM empvu10
         *
ERROR at line 1:
ORA-01752: cannot delete from view without exactly one key-
preserved table
```

Any attempt to insert a row or modify a row using the view with a read-only constraint results in Oracle server error:

```
01733: virtual column not allowed here.
```

Removing a View

You can remove a view without losing data because a view is based on underlying tables in the database.

```
DROP VIEW view;
```

```
DROP VIEW empvu80;
View dropped.
```

ORACLE

Removing a View

You use the `DROP VIEW` statement to remove a view. The statement removes the view definition from the database. Dropping views has no effect on the tables on which the view was based. Views or other applications based on deleted views become invalid. Only the creator or a user with the `DROP ANY VIEW` privilege can remove a view.

In the syntax:

view	is the name of the view

Inline Views

- **An inline view is a subquery with an alias (or correlation name) that you can use within a SQL statement.**
- **A named subquery in the `FROM` clause of the main query is an example of an inline view.**
- **An inline view is not a schema object.**

Inline Views

An inline view is created by placing a subquery in the `FROM` clause and giving that subquery an alias. The subquery defines a data source that can be referenced in the main query. In the following example, the inline view `b` returns the details of all department numbers and the maximum salary for each department from the `EMPLOYEES` table. The `WHERE a.department_id = b.department_id AND a.salary < b.maxsal` clause of the main query displays employee names, salaries, department numbers, and maximum salaries for all the employees who earn less than the maximum salary in their department.

```
SELECT  a.last_name, a.salary, a.department_id, b.maxsal
FROM    employees a, (SELECT   department_id, max(salary) maxsal
                      FROM     employees
                      GROUP BY department_id) b
WHERE   a.department_id = b.department_id
AND     a.salary < b.maxsal;
```

LAST_NAME	SALARY	DEPARTMENT_ID	MAXSAL
Fay	6000	20	13000
Rajs	3500	50	5800
Davies	3100	50	5800
Matos	2600	50	5800
Vargas	2500	50	5800

...

Top-N Analysis

- **Top-N queries ask for the *n* largest or smallest values of a column. For example:**
  - **What are the ten best selling products?**
  - **What are the ten worst selling products?**
- **Both largest values and smallest values sets are considered Top-N queries.**

ORACLE

"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:

```
SELECT [column_list], ROWNUM
FROM   (SELECT [column_list]
        FROM table
        ORDER  BY Top-N_column)
WHERE  ROWNUM <=  N;
```

ORACLE

Performing "Top-N" Analysis

Top-N queries use a consistent nested query structure with the elements described below:

- A subquery or an inline view to generate the sorted list of data. The subquery or the inline view includes the `ORDER BY` clause to ensure that the ranking is in the desired order. For results retrieving the largest values, a `DESC` parameter is needed.
- An outer query to limit the number of rows in the final result set. The outer query includes the following components:
  - The `ROWNUM` pseudocolumn, which assigns a sequential value starting with 1 to each of the rows returned from the subquery.
  - A `WHERE` clause, which specifies the *n* rows to be returned. The outer `WHERE` clause must use a < or <= operator.

Example of Top-N Analysis

To display the top three earner names and salaries from the EMPLOYEES table:

```
SELECT ROWNUM as RANK, last_name, salary
FROM  (SELECT last_name,salary FROM employees
       ORDER BY salary DESC)
WHERE ROWNUM <= 3;
```

RANK	LAST_NAME	SALARY
1	King	24000
2	Kochhar	17000
3	De Haan	17000

ORACLE

Example of "Top-N" Analysis

The example on the slide illustrates how to display the names and salaries of the top three earners from the EMPLOYEES table. The subquery returns the details of all employee names and salaries from the EMPLOYEES table, sorted in the descending order of the salaries. The WHERE ROWNUM < 3 clause of the main query ensures that only the first three records from this result set are displayed.

Here is another example of Top-N analysis that uses an inline view. The example below uses the inline view E to display the four most senior employees in the company.

```
SELECT ROWNUM as SENIOR,E.last_name, E.hire_date
FROM  (SELECT last_name,hire_date FROM employees
       ORDER BY hire_date)E
WHERE rownum <= 4;
```

SENIOR	LAST_NAME	HIRE_DATE
1	King	17-JUN-87
2	Whalen	17-SEP-87
3	Kochhar	21-SEP-89
4	Hunold	03-JAN-90

Summary

In this lesson, you should have learned that a view is derived from data in other tables or views and provides the following advantages:

- **Restricts database access**
- **Simplifies queries**
- **Provides data independence**
- **Provides multiple views of the same data**
- **Can be dropped without removing the underlying data**
- **An inline view is a subquery with an alias name.**
- **Top-N analysis can be done using subqueries and outer queries.**

ORACLE

What Is a View?

A view is based on a table or another view and acts as a window through which data on tables can be viewed or changed. A view does not contain data. The definition of the view is stored in the data dictionary. You can see the definition of the view in the `USER_VIEWS` data dictionary table.

Advantages of Views

- Restrict database access
- Simplify queries
- Provide data independence
- Provide multiple views of the same data
- Can be removed without affecting the underlying data

View Options

- Can be a simple view, based on one table
- Can be a complex view based on more than one table or can contain groups of functions
- Can replace other views with the same name
- Can contain a check constraint
- Can be read-only

Practice 11

1. Create a view called `EMPLOYEES_VU` based on the employee numbers, employee names, and department numbers from the `EMPLOYEES` table. Change the heading for the employee name to `EMPLOYEE`.
2. Display the contents of the `EMPLOYEES_VU` view.

EMPLOYEE_ID	EMPLOYEE	DEPARTMENT_ID
100	King	90
101	Kochhar	90
102	De Haan	90
103	Hunold	60
104	Ernst	60
107	Lorentz	60
...		
206	Gietz	110

20 rows selected.

3. Select the view name and text from the `USER_VIEWS` data dictionary view.

   Note: Another view already exists. The `EMP_DETAILS_VIEW` was created as part of your schema.

   Note: To see more contents of a `LONG` column, use the *i*SQL*Plus command `SET LONG` *n*, where *n* is the value of the number of characters of the `LONG` column that you want to see.

VIEW_NAME	TEXT
EMPLOYEES_VU	SELECT employee_id, last_name employee, department_id FROM employees
EMP_DETAILS_VIEW	SELECT e.employee_id, e.job_id, e.manager_id, e.department_id, d.locat ion_id, l.country_id, e.first_name, e.last_name, e.salary, e.commissio n_pct, d.department_name, j.job_title, l.city, l.state_province, c.cou ntry_name, r.region_name FROM employees e, departments d, jobs j, loca tions l, countries c, regions r WHERE e.department_id = d.department_id AN D d.location_id = l.location_id AND l.country_id = c.country_id AND c.region _id = r.region_id AND j.job_id = e.job_id WITH READ ONLY

4. Using your `EMPLOYEES_VU` view, enter a query to display all employee names and department numbers.

EMPLOYEE	DEPARTMENT_ID
King	90
Kochhar	90
...	
Gietz	110

20 rows selected.

5. Create a view named `DEPT50` that contains the employee numbers, employee last names, and department numbers for all employees in department 50. Label the view columns `EMPNO`, `EMPLOYEE`, and `DEPTNO`. Do not allow an employee to be reassigned to another department through the view.
6. Display the structure and contents of the `DEPT50` view.

Name	Null?	Type
EMPNO	NOT NULL	NUMBER(6)
EMPLOYEE	NOT NULL	VARCHAR2(25)
DEPTNO		NUMBER(4)

EMPNO	EMPLOYEE	DEPTNO
124	Mourgos	50
141	Rajs	50
142	Davies	50
143	Matos	50
144	Vargas	50

7. Attempt to reassign Matos to department 80.

If you have time, complete the following exercise:

8. Create a view called `SALARY_VU` based on the employee last names, department names, salaries, and salary grades for all employees. Use the `EMPLOYEES`, `DEPARTMENTS`, and `JOB_GRADES` tables. Label the columns `Employee`, `Department`, `Salary`, and `Grade`, respectively.

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?

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**

What Is a Sequence?

A sequence is a user created database object that can be shared by multiple users to generate unique integers.

A typical usage for sequences is to create a primary key value, which must be unique for each row. The sequence is generated and incremented (or decremented) by an internal Oracle routine. This can be a time-saving object because it can reduce the amount of application code needed to write a sequence-generating routine.

Sequence numbers are stored and generated independently of tables. Therefore, the same sequence can be used for multiple tables.

The CREATE SEQUENCE Statement Syntax

Define a sequence to generate sequential numbers automatically:

```
CREATE SEQUENCE sequence
       [INCREMENT BY n]
       [START WITH n]
       [{MAXVALUE n | NOMAXVALUE}]
       [{MINVALUE n | NOMINVALUE}]
       [{CYCLE | NOCYCLE}]
       [{CACHE n | NOCACHE}];
```

ORACLE

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 increments by 1.)
START WITH *n*	specifies the first sequence number to be generated (If this clause is omitted, the sequence starts 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 whether the sequence continues to generate values after reaching its maximum or minimum value (`NOCYCLE` is the default option.)
CACHE *n* | NOCACHE	specifies how many values the Oracle server preallocates and keep in memory (By default, the Oracle server caches 20 values.)

Creating a Sequence

- **Create a sequence named `DEPT_DEPTID_SEQ` to be used for the primary key of the `DEPARTMENTS` table.**
- **Do not use the `CYCLE` option.**

```
CREATE SEQUENCE dept_deptid_seq
                INCREMENT BY 10
                START WITH 120
                MAXVALUE 9999
                NOCACHE
                NOCYCLE;
Sequence created.
```

ORACLE

Creating a Sequence (continued)

The example on the slide creates a sequence named `DEPT_DEPTID_SEQ` to be used for the `DEPARTMENT_ID` column of the `DEPARTMENTS` table. The sequence starts at 120, does not allow caching, and does not cycle.

Do not use the `CYCLE` option if the sequence is used to generate primary key values, unless you have a reliable mechanism that purges old rows faster than the sequence cycles.

Confirming Sequences

- **Verify your sequence values in the `USER_SEQUENCES` data dictionary table.**

```
SELECT   sequence_name, min_value, max_value,
         increment_by, last_number
FROM     user_sequences;
```

- **The `LAST_NUMBER` column displays the next available sequence number if `NOCACHE` is specified.**

ORACLE

Confirming Sequences

Once you have created your sequence, it is documented in the data dictionary. Since a sequence is a database object, you can identify it in the `USER_OBJECTS` data dictionary table.

You can also confirm the settings of the sequence by selecting from the `USER_SEQUENCES` data dictionary view.

SEQUENCE_NAME	MIN_VALUE	MAX_VALUE	INCREMENT_BY	LAST_NUMBER
DEPARTMENTS_SEQ	1	9990	10	280
DEPT_DEPTID_SEQ	1	9999	10	120
EMPLOYEES_SEQ	1	1.0000E+27	1	207
LOCATIONS_SEQ	1	9900	100	3300

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

After you create your sequence, it generates sequential numbers for use in your tables. Reference the sequence values by using the NEXTVAL and CURRVAL pseudocolumns.

NEXTVAL and CURRVAL Pseudocolumns

The NEXTVAL pseudocolumn is used to extract successive sequence numbers from a specified sequence. You must qualify NEXTVAL with the sequence name. When you reference *sequence*.NEXTVAL, a new sequence number is generated and the current sequence number is placed in CURRVAL.

The CURRVAL pseudocolumn is used to refer to a sequence number that the current user has just generated. NEXTVAL must be used to generate a sequence number in the current user's session before CURRVAL can be referenced. You must qualify CURRVAL with the sequence name. When *sequence*.CURRVAL is referenced, the last value returned to that user's process is displayed.

Rules for Using NEXTVAL and CURRVAL

You can use NEXTVAL and CURRVAL in the following contexts:

- The SELECT list of a SELECT statement that is not part of a subquery
- The SELECT list of a subquery in an INSERT statement
- The VALUES clause of an INSERT statement
- The SET clause of an UPDATE statement

You cannot use NEXTVAL and CURRVAL in the following contexts:

- The SELECT list of a view
- A SELECT statement with the DISTINCT keyword
- A SELECT statement with GROUP BY, HAVING, or ORDER BY clauses
- A subquery in a SELECT, DELETE, or UPDATE statement
- The DEFAULT expression in a CREATE TABLE or ALTER TABLE statement

Using a Sequence

- **Insert a new department named "Support" in location ID 2500.**

```
INSERT INTO departments(department_id,
            department_name, location_id)
VALUES      (dept_deptid_seq.NEXTVAL,
            'Support', 2500);
1 row created.
```

- **View the current value for the `DEPT_DEPTID_SEQ` sequence.**

```
SELECT   dept_deptid_seq.CURRVAL
FROM     dual;
```

Using a Sequence

The example on the slide inserts a new department in the `DEPARTMENTS` table. It uses the `DEPT_DEPTID_SEQ` sequence for generating a new department number as follows:

You can view the current value of the sequence:

```
SELECT dept_deptid_seq.CURRVAL
FROM   dual;
```

CURRVAL
120

Suppose now you want to hire employees to staff the new department. The `INSERT` statement to be executed for all new employees can include the following code:

```
INSERT INTO employees (employee_id, department_id, ...)
VALUES (employees_seq.NEXTVAL, dept_deptid_seq .CURRVAL, ...);
```

Note: The preceding example assumes that a sequence called `EMPLOYEE_SEQ` has already been created for generating new employee numbers.

Using a Sequence

- **Caching sequence values in memory gives faster access to those values.**
- **Gaps in sequence values can occur when:**
  - **A rollback occurs**
  - **The system crashes**
  - **A sequence is used in another table**
- **If the sequence was created with `NOCACHE`, view the next available value, by querying the `USER_SEQUENCES` table.**

ORACLE

Caching Sequence Values

Cache sequences in memory to provide faster access to those sequence values. The cache is populated the first time you refer to the sequence. Each request for the next sequence value is retrieved from the cached sequence. After the last sequence value is used, the next request for the sequence pulls another cache of sequences into memory.

Gaps in the Sequence

Although sequence generators issue sequential numbers without gaps, this action occurs independent of a commit or rollback. Therefore, if you roll back a statement containing a sequence, the number is lost.

Another event that can cause gaps in the sequence is a system crash. If the sequence caches values in the memory, then those values are lost if the system crashes.

Because sequences are not tied directly to tables, the same sequence can be used for multiple tables. If you do so, each table can contain gaps in the sequential numbers.

Viewing the Next Available Sequence Value without Incrementing It

If the sequence was created with `NOCACHE`, it is possible to view the next available sequence value without incrementing it by querying the `USER_SEQUENCES` table.

Modifying a Sequence

Change the increment value, maximum value, minimum value, cycle option, or cache option.

```
ALTER SEQUENCE dept_deptid_seq
               INCREMENT BY 20
               MAXVALUE 999999
               NOCACHE
               NOCYCLE;
Sequence altered.
```

Altering a Sequence

If you reach the `MAXVALUE` limit for your sequence, no additional values from the sequence are allocated and you will receive an error indicating that the sequence exceeds the `MAXVALUE`. To continue to use the sequence, you can modify it by using the `ALTER SEQUENCE` statement.

Syntax

```
ALTER  SEQUENCE   sequence
       [INCREMENT BY n]
       [{MAXVALUE n | NOMAXVALUE}]
       [{MINVALUE n | NOMINVALUE}]
       [{CYCLE | NOCYCLE}]
       [{CACHE n | NOCACHE}];
```

In the syntax:

sequence is the name of the sequence generator

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.**

Guidelines for Modifying Sequences

- You must be the owner or have the ALTER privilege for the sequence in order to modify it.
- Only future sequence numbers are affected by the ALTER SEQUENCE statement.
- The START WITH option cannot be changed using ALTER SEQUENCE. The sequence must be dropped and re-created in order to restart the sequence at a different number.
- Some validation is performed. For example, a new MAXVALUE that is less than the current sequence number cannot be imposed.

```
ALTER SEQUENCE dept_deptid_seq
      INCREMENT BY 20
      MAXVALUE 90
      NOCACHE
      NOCYCLE;
ALTER SEQUENCE dept_deptid_seq
*
ERROR at line 1:
ORA-04009: MAXVALUE cannot be made to be less than the current
           value
```

Removing a Sequence

- Remove a sequence from the data dictionary by using the `DROP SEQUENCE` statement.
- Once removed, the sequence can no longer be referenced.

```
DROP SEQUENCE dept_deptid_seq;
Sequence dropped.
```

Removing a Sequence

To remove a sequence from the data dictionary, use the `DROP SEQUENCE` statement. You must be the owner of the sequence or have the `DROP ANY SEQUENCE` privilege to remove it.

Syntax

```
DROP   SEQUENCE    sequence;
```

In the syntax:

sequence is the name of the sequence generator

What is an Index?

An index:

- **Is a schema object**
- **Is used by the Oracle server to speed up the retrieval of rows by using a pointer**
- **Can reduce disk I/O by using a rapid path access method to locate data quickly**
- **Is independent of the table it indexes**
- **Is used and maintained automatically by the Oracle server**

Indexes

An Oracle server index is a schema object that can speed up the retrieval of rows by using a pointer. Indexes can be created explicitly or automatically. If you do not have an index on the column, then a full table scan occurs.

An index provides direct and fast access to rows in a table. Its purpose is to reduce the necessity of disk I/O by using an indexed path to locate data quickly. The index is used and maintained automatically by the Oracle server. Once an index is created, no direct activity is required by the user.

Indexes are logically and physically independent of the table they index. This means that they can be created or dropped at any time and have no effect on the base tables or other indexes.

Note: When you drop a table, corresponding indexes are also dropped.

For more information, see *Oracle9i Concepts*, “Schema Objects” section, “Indexes” topic.

How Are Indexes Created?

- **Automatically: A unique index is created automatically when you define a `PRIMARY KEY` or `UNIQUE` constraint in a table definition.**
- **Manually: Users can create nonunique indexes on columns to speed up access to the rows.**

Types of Indexes

Two types of indexes can be created. One type is a unique index: the Oracle server automatically creates this index when you define a column in a table to have a `PRIMARY KEY` or a `UNIQUE` key constraint. The name of the index is the name given to the constraint.

The other type of index is a nonunique index, which a user can create. For example, you can create a `FOREIGN KEY` column index for a join in a query to improve retrieval speed.

Note: You can manually create a unique index, but it is recommended that you create a unique constraint, which implicitly creates a unique index.

Creating an Index

- **Create an index on one or more columns.**

```
CREATE INDEX index
ON table (column[, column]...);
```

- **Improve the speed of query access to the LAST_NAME column in the EMPLOYEES table.**

```
CREATE INDEX  emp_last_name_idx
ON            employees(last_name);
Index created.
```

ORACLE

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

When to Create an Index

You should create an index if:

- A column contains a wide range of values
- A column contains a large number of null values
- One or more columns are frequently used together in a `WHERE` clause or a join condition
- The table is large and most queries are expected to retrieve less than 2 to 4 percent of the rows

ORACLE

More Is Not Always Better

More indexes on a table does not mean faster queries. Each DML operation that is committed on a table with indexes means that the indexes must be updated. The more indexes you have associated with a table, the more effort the Oracle server must make to update all the indexes after a DML operation.

When to Create an Index

Therefore, you should create indexes only if:

- The column contains a wide range of values
- The column contains a large number of null values
- One or more columns are frequently used together in a `WHERE` clause or join condition
- The table is large and most queries are expected to retrieve less than 2–4% of the rows

Remember that if you want to enforce uniqueness, you should define a unique constraint in the table definition. Then a unique index is created automatically.

When Not to Create an Index

It is usually not worth creating an index if:

- The table is small
- The columns are not often used as a condition in the query
- Most queries are expected to retrieve more than 2 to 4 percent of the rows in the table
- The table is updated frequently
- The indexed columns are referenced as part of an expression

Confirming Indexes

- The `USER_INDEXES` data dictionary view contains the name of the index and its uniqueness.
- The `USER_IND_COLUMNS` view contains the index name, the table name, and the column name.

```
SELECT   ic.index_name, ic.column_name,
         ic.column_position col_pos,ix.uniqueness
FROM     user_indexes ix, user_ind_columns ic
WHERE    ic.index_name = ix.index_name
AND      ic.table_name = 'EMPLOYEES';
```

ORACLE

Confirming Indexes

Confirm the existence of indexes from the `USER_INDEXES` data dictionary view. You can also check the columns involved in an index by querying the `USER_IND_COLUMNS` view.

The example on the slide displays all the previously created indexes, with the names of the affected column, and the index's uniqueness, on the `EMPLOYEES` table.

INDEX_NAME	COLUMN_NAME	COL_POS	UNIQUENES
EMP_EMAIL_UK	EMAIL	1	UNIQUE
EMP_EMP_ID_PK	EMPLOYEE_ID	1	UNIQUE
EMP_DEPARTMENT_IX	DEPARTMENT_ID	1	NONUNIQUE
EMP_JOB_IX	JOB_ID	1	NONUNIQUE
EMP_MANAGER_IX	MANAGER_ID	1	NONUNIQUE
EMP_NAME_IX	LAST_NAME	1	NONUNIQUE
EMP_NAME_IX	FIRST_NAME	2	NONUNIQUE
EMP_LAST_NAME_IDX	LAST_NAME	1	NONUNIQUE

8 rows selected.

Function-Based Indexes

- **A function-based index is an index based on expressions.**
- **The index expression is built from table columns, constants, SQL functions, and user-defined functions.**

```
CREATE INDEX upper_dept_name_idx
ON departments(UPPER(department_name));

Index created.

SELECT *
FROM   departments
WHERE  UPPER(department_name) = 'SALES';
```

ORACLE

Function-Based Index

Function-based indexes defined with the `UPPER(column_name)` or `LOWER(column_name)` keywords allow case-insensitive searches. For example, the following index:

```
CREATE INDEX upper_last_name_idx ON employees (UPPER(last_name));
```

Facilitates processing queries such as:

```
SELECT * FROM employees WHERE UPPER(last_name) = 'KING';
```

To ensure that the Oracle server uses the index rather than performing a full table scan, be sure that the value of the function is not null in subsequent queries. For example, the following statement is guaranteed to use the index, but without the `WHERE` clause the Oracle server may perform a full table scan:

```
SELECT   *
FROM     employees
WHERE    UPPER (last_name) IS NOT NULL
ORDER BY UPPER (last_name);
```

Removing an Index

- Remove an index from the data dictionary by using the `DROP INDEX` command.

```
DROP INDEX index;
```

- Remove the `UPPER_LAST_NAME_IDX` index from the data dictionary.

```
DROP INDEX upper_last_name_idx;
Index dropped.
```

- To drop an index, you must be the owner of the index or have the `DROP ANY INDEX` privilege.

ORACLE

Removing an Index

You cannot modify indexes. To change an index, you must drop it and then re-create it. Remove an index definition from the data dictionary by issuing the `DROP INDEX` statement. To drop an index, you must be the owner of the index or have the `DROP ANY INDEX` privilege.

In the syntax:

index is the name of the index

Note: If you drop a table, indexes and constraints are automatically dropped, but views and sequences remain.

Synonyms

Simplify access to objects by creating a synonym (another name for an object). With synonyms, you can:

- **Ease referring to a table owned by another user**
- **Shorten lengthy object names**

```
CREATE [PUBLIC] SYNONYM synonym
FOR    object;
```

ORACLE

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.

Creating and Removing Synonyms

- **Create a shortened name for the `DEPT_SUM_VU` view.**

```
CREATE SYNONYM  d_sum
FOR  dept_sum_vu;
Synonym Created.
```

- **Drop a synonym.**

```
DROP SYNONYM d_sum;
Synonym dropped.
```

Creating a Synonym for an Object (continued)

The slide example creates a synonym for the `DEPT_SUM_VU` view for quicker reference.

The database administrator can create a public synonym accessible to all users. The following example creates a public synonym named `DEPT` for Alice’s `DEPARTMENTS` table:

```
CREATE PUBLIC SYNONYM  dept
FOR    alice.departments;
Synonym created.
```

Removing a Synonym

To drop a synonym, use the `DROP SYNONYM` statement. Only the database administrator can drop a public synonym.

```
DROP PUBLIC SYNONYM  dept;
Synonym dropped.
```

Summary

In this lesson, you should have learned how to:

- **Automatically generate sequence numbers by using a sequence generator**
- **View sequence information in the `USER_SEQUENCES` data dictionary table**
- **Create indexes to improve query retrieval speed**
- **View index information in the `USER_INDEXES` dictionary table**
- **Use synonyms to provide alternative names for objects**

ORACLE

Summary

In this lesson you should have learned about some of the other database objects including sequences, indexes, and views.

Sequences

The sequence generator can be used to automatically generate sequence numbers for rows in tables. This can save time and can reduce the amount of application code needed.

A sequence is a database object that can be shared with other users. Information about the sequence can be found in the `USER_SEQUENCES` table of the data dictionary.

To use a sequence, reference it with either the `NEXTVAL` or the `CURRVAL` pseudocolumns.

- Retrieve the next number in the sequence by referencing *`sequence`*`.NEXTVAL`.
- Return the current available number by referencing *`sequence`*`.CURRVAL`.

Indexes

Indexes are used to improve query retrieval speed. Users can view the definitions of the indexes in the `USER_INDEXES` data dictionary view. An index can be dropped by the creator, or a user with the `DROP ANY INDEX` privilege, by using the `DROP INDEX` statement.

Synonyms

Database administrators can create public synonyms and users can create private synonyms for convenience, by using the `CREATE SYNONYM` statement. Synonyms permit short names or alternative names for objects. Remove synonyms by using the `DROP SYNONYM` statement.

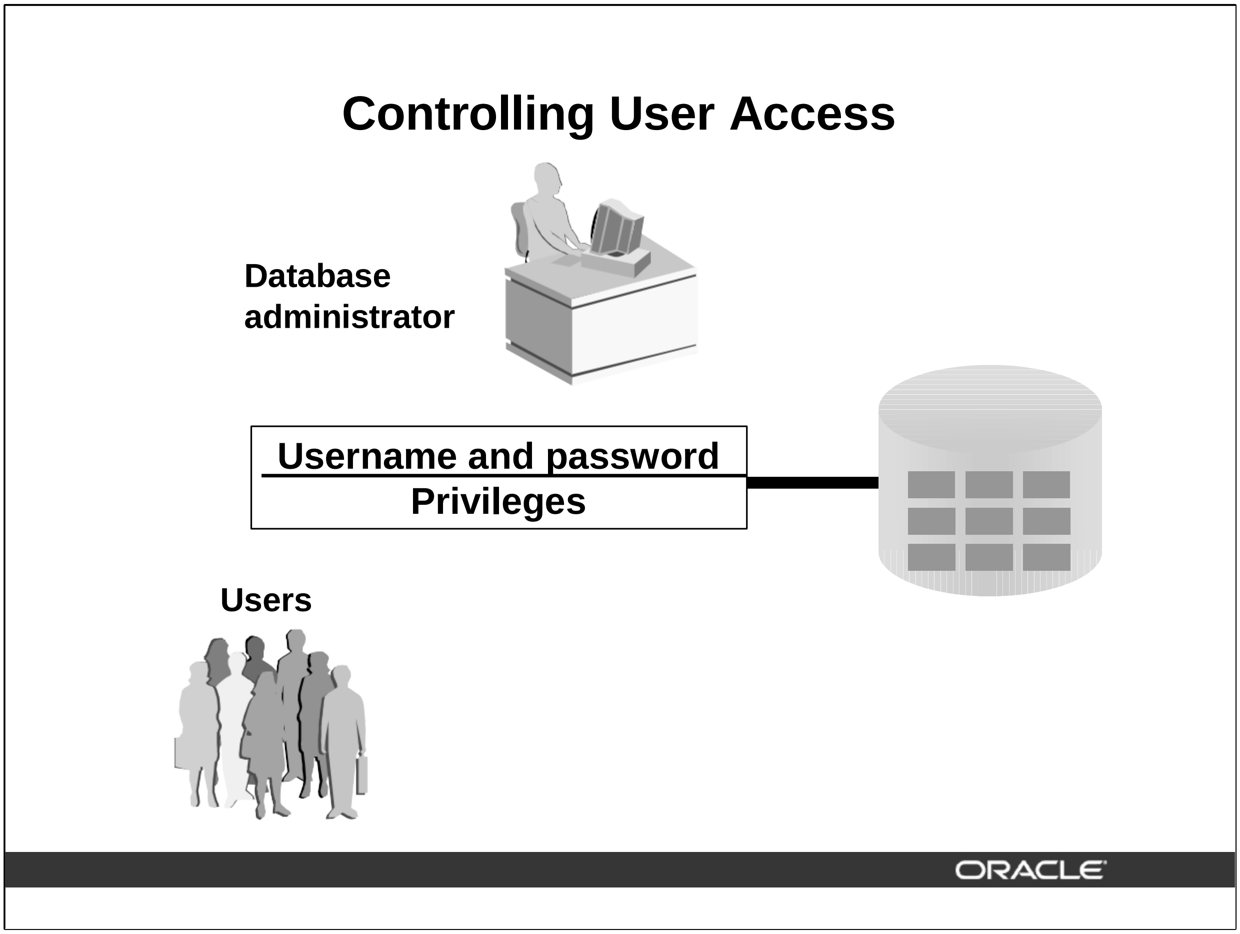
Practice

1. Create a sequence to be used with the primary key column of the `DEPT` table. The sequence should start at 200 and have a maximum value of 1000. Have your sequence increment by ten numbers. Name the sequence `DEPT_ID_SEQ`.
2. Write a query in a script to display the following information about your sequences: sequence name, maximum value, increment size, and last number. Name the script `lab12_2.sql`. Run the statement in your script.

SEQUENCE_NAME	MAX_VALUE	INCREMENT_BY	LAST_NUMBER
DEPARTMENTS_SEQ	9990	10	280
DEPT_ID_SEQ	1000	10	200
EMPLOYEES_SEQ	1.0000E+27	1	207
LOCATIONS_SEQ	9900	100	3300

3. Write a script to insert two rows into the `DEPT` table. Name your script `lab12_3.sql`. Be sure to use the sequence that you created for the ID column. Add two departments named Education and Administration. Confirm your additions. Run the commands in your script.
4. Create a nonunique index on the foreign key column (`DEPT_ID`) in the `EMP` table.
5. Display the indexes and uniqueness that exist in the data dictionary for the `EMP` table. Save the statement into a script named `lab12_5.sql`.

INDEX_NAME	TABLE_NAME	UNIQUENES
EMP_DEPT_ID_IDX	EMP	NONUNIQUE
MY_EMP_ID_PK	EMP	UNIQUE



Controlling User Access

In a multiple-user environment, you want to maintain security of the database access and use. With Oracle server database security, you can do the following:

- Control database access
- Give access to specific objects in the database
- Confirm given and received *privileges* with the Oracle data dictionary
- Create synonyms for database objects

Database security can be classified into two categories: system security and data security. System security covers access and use of the database at the system level, such as the username and password, the disk space allocated to users, and the system operations that users can perform. Database security covers access and use of the database objects and the actions that those users can have on the objects.

Privileges

- **Database security:**
  - **System security**
  - **Data security**
- **System privileges: Gaining access to the database**
- **Object privileges: Manipulating the content of the database objects**
- **Schemas: Collections of objects, such as tables, views, and sequences**

Privileges

Privileges are the right to execute particular SQL statements. The database administrator (DBA) is a high-level user with the ability to grant users access to the database and its objects. The users require *system privileges* to gain access to the database and *object privileges* to manipulate the content of the objects in the database. Users can also be given the privilege to grant additional privileges to other users or to *roles*, which are named groups of related privileges.

Schemas

A *schema* is a collection of objects, such as tables, views, and sequences. The schema is owned by a database user and has the same name as that user.

System Privileges

- **More than 100 privileges are available.**
- **The database administrator has high-level system privileges for tasks such as:**
  - **Creating new users**
  - **Removing users**
  - **Removing tables**
  - **Backing up tables**

ORACLE

System Privileges

More than 100 distinct system privileges are available for users and roles. System privileges typically are provided by the database administrator.

Typical DBA Privileges

System Privilege	Operations Authorized
`CREATE USER`	Grantee can create other Oracle users (a privilege required for a DBA role).
`DROP USER`	Grantee can drop another user.
`DROP ANY TABLE`	Grantee can drop a table in any schema.
`BACKUP ANY TABLE`	Grantee can back up any table in any schema with the export utility.
`SELECT ANY TABLE`	Grantee can query tables, views, or snapshots in any schema.
`CREATE ANY TABLE`	Grantee can create tables in any schema.

Creating Users

The DBA creates users by using the `CREATE USER` statement.

```
CREATE USER user
IDENTIFIED BY   password;
```

```
CREATE USER  scott
IDENTIFIED BY   tiger;
User created.
```

ORACLE

Creating a User

The DBA creates the user by executing the `CREATE USER` statement. The user does not have any privileges at this point. The DBA can then grant privileges to that user. These privileges determine what the user can do at the database level.

The slide gives the abridged syntax for creating a user.

In the syntax:

`user`	is the name of the user to be created
`password`	specifies that the user must log in with this password

User System Privileges

- **Once a user is created, the DBA can grant specific system privileges to a user.**

```
GRANT privilege [, privilege...]
TO user [, user| role, PUBLIC...];
```

- **An application developer, for example, may have the following system privileges:**
  - **`CREATE SESSION`**
  - **`CREATE TABLE`**
  - **`CREATE SEQUENCE`**
  - **`CREATE VIEW`**
  - **`CREATE PROCEDURE`**

Typical User Privileges

Now that the DBA has created a user, the DBA can assign privileges to that user.

System Privilege	Operations Authorized
`CREATE SESSION`	Connect to the database
`CREATE TABLE`	Create tables in the user's schema
`CREATE SEQUENCE`	Create a sequence in the user's schema
`CREATE VIEW`	Create a view in the user's schema
`CREATE PROCEDURE`	Create a stored procedure, function, or package in the user's schema

In the syntax:

`privilege`	is the system privilege to be granted
`user` `|role|PUBLIC`	is the name of the user, the name of the role, or `PUBLIC` designates that every user is granted the privilege

Note: Current system privileges can be found in the dictionary view `SESSION_PRIVS`.

Granting System Privileges

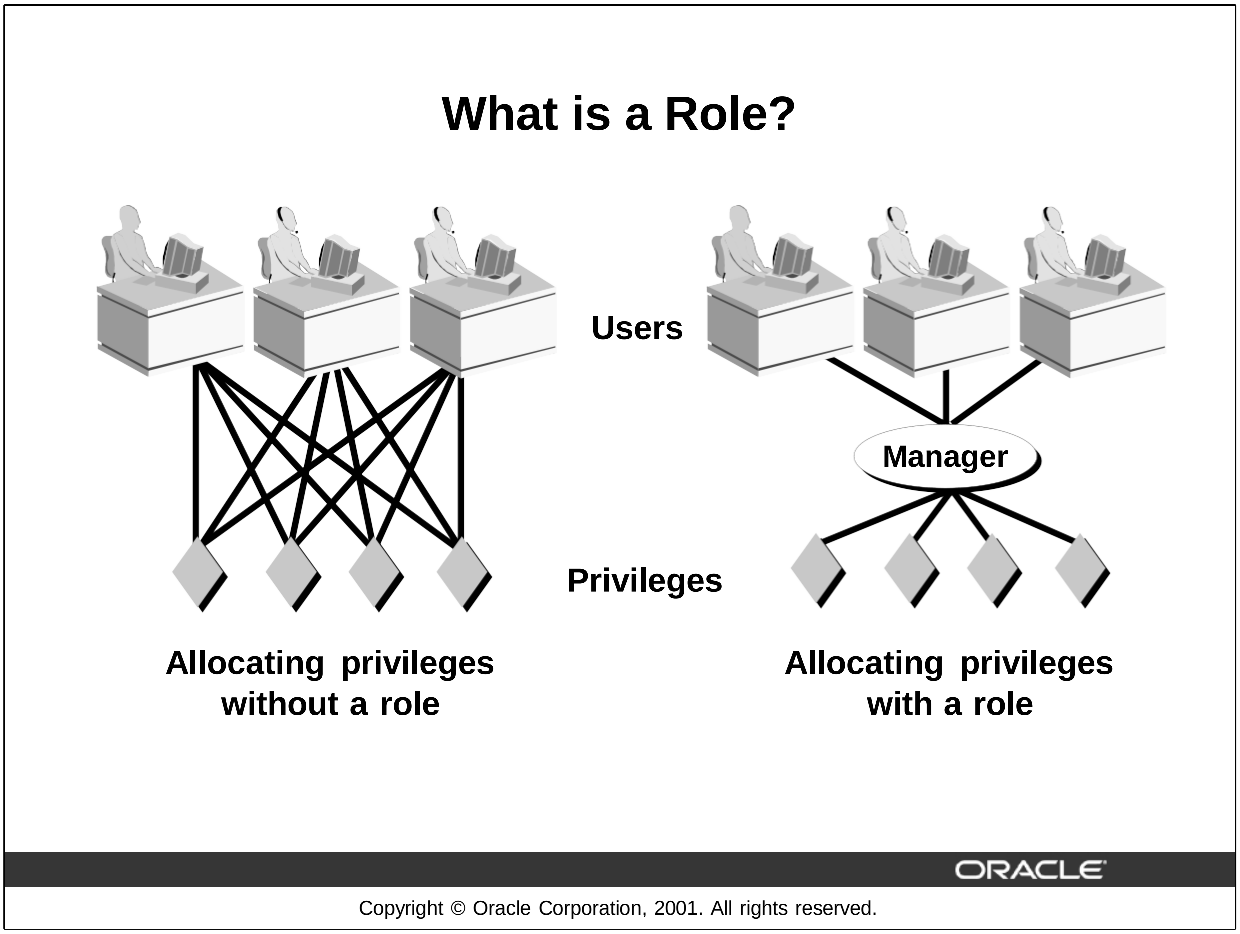
The DBA can grant a user specific system privileges.

```
GRANT  create session, create table,
       create sequence, create view
TO     scott;
Grant succeeded.
```

Granting System Privileges

The DBA uses the `GRANT` statement to allocate system privileges to the user. Once the user has been granted the privileges, the user can immediately use those privileges.

In the example on the slide, user Scott has been assigned the privileges to create sessions, tables, sequences, and views.



What is a Role?

A role is a named group of related privileges that can be granted to the user. This method makes it easier to revoke and maintain privileges.

A user can have access to several roles, and several users can be assigned the same role. Roles are typically created for a database application.

Creating and Assigning a Role

First, the DBA must create the role. Then the DBA can assign privileges to the role and users to the role.

Syntax

```
CREATE   ROLE  role;
```

In the syntax:

role is the name of the role to be created

Now that the role is created, the DBA can use the `GRANT` statement to assign users to the role as well as assign privileges to the role.

Creating and Granting Privileges to a Role

- **Create a role**

```
CREATE ROLE manager;
Role created.
```

- **Grant privileges to a role**

```
GRANT create table, create view
TO manager;
Grant succeeded.
```

- **Grant a role to users**

```
GRANT manager TO DEHAAN, KOCHHAR;
Grant succeeded.
```

Creating a Role

The example on the slide creates a manager role and then allows managers to create tables and views. It then grants DeHaan and Kochhar the role of managers. Now DeHaan and Kochhar can create tables and views.

If users have multiple roles granted to them, they receive all of the privileges associated with all of the roles.

Changing Your Password

- **The DBA creates your user account and initializes your password.**
- **You can change your password by using the `ALTER USER` statement.**

```
ALTER USER scott
IDENTIFIED BY lion;
User altered.
```

ORACLE

Changing Your Password

The DBA creates an account and initializes a password for every user. You can change your password by using the `ALTER USER` statement.

Syntax

```
ALTER USER user IDENTIFIED BY password;
```

In the syntax:

user	is the name of the user
password	specifies the new password

Although this statement can be used to change your password, there are many other options. You must have the `ALTER USER` privilege to change any other option.

Object Privileges

Object Privilege	Table	View	Sequence	Procedure
ALTER				
DELETE				
EXECUTE				
INDEX				
INSERT				
REFERENCES				
SELECT				
UPDATE				

Object Privileges

An *object privilege* is a privilege or right to perform a particular action on a specific table, view, sequence, or procedure. Each object has a particular set of grantable privileges. The table on the slide lists the privileges for various objects. Note that the only privileges that apply to a sequence are `SELECT` and `ALTER`. `UPDATE`, `REFERENCES`, and `INSERT` can be restricted by specifying a subset of updateable columns. A `SELECT` privilege can be restricted by creating a view with a subset of columns and granting the `SELECT` privilege only on the view. A privilege granted on a synonym is converted to a privilege on the base table referenced by the synonym.

Object Privileges

- **Object privileges vary from object to object.**
- **An owner has all the privileges on the object.**
- **An owner can give specific privileges on that owner's object.**

```
GRANT        object_priv [(columns)]
ON           object
TO           {user|role|PUBLIC}
[WITH GRANT OPTION];
```

ORACLE

Granting Object Privileges

Different object privileges are available for different types of schema objects. A user automatically has all object privileges for schema objects contained in the user's schema. A user can grant any object privilege on any schema object that the user owns to any other user or role. If the grant includes `WITH GRANT OPTION`, then the grantee can further grant the object privilege to other users; otherwise, the grantee can use the privilege but cannot grant it to other users.

In the syntax:

object_priv	is an object privilege to be granted
ALL	specifies all object privileges
columns	specifies the column from a table or view on which privileges are granted
ON *object*	is the object on which the privileges are granted
TO	identifies to whom the privilege is granted
PUBLIC	grants object privileges to all users
WITH GRANT OPTION	allows the grantee to grant the object privileges to other users and roles

Granting Object Privileges

- **Grant query privileges on the `EMPLOYEES` table.**

```
GRANT  select
ON     employees
TO     sue, rich;
Grant succeeded.
```

- **Grant privileges to update specific columns to users and roles.**

```
GRANT  update (department_name, location_id)
ON     departments
TO     scott, manager;
Grant succeeded.
```

ORACLE

Guidelines

- To grant privileges on an object, the object must be in your own schema, or you must have been granted the object privileges `WITH GRANT OPTION`.
- An object owner can grant any object privilege on the object to any other user or role of the database.
- The owner of an object automatically acquires all object privileges on that object.

The first example on the slide grants users Sue and Rich the privilege to query your `EMPLOYEES` table. The second example grants `UPDATE` privileges on specific columns in the `DEPARTMENTS` table to Scott and to the manager role.

If Sue or Rich now want to `SELECT` data from the employees table, the syntax they must use is:

```
SELECT  *
FROM    scott.employees;
```

Alternatively, they can create a synonym for the table and `SELECT` from the synonym:

```
CREATE SYNONYM emp FOR scott.employees;
SELECT * FROM emp;
```

Note: DBAs generally allocate system privileges; any user who owns an object can grant object privileges.

Using the WITH GRANT OPTION and PUBLIC Keywords

- **Give a user authority to pass along privileges.**

```
GRANT  select, insert
ON     departments
TO     scott
WITH   GRANT OPTION;
Grant succeeded.
```

- **Allow all users on the system to query data from Alice's DEPARTMENTS table.**

```
GRANT  select
ON     alice.departments
TO     PUBLIC;
Grant succeeded.
```

ORACLE

The WITH GRANT OPTION Keyword

A privilege that is granted with the WITH GRANT OPTION clause can be passed on to other users and roles by the grantee. Object privileges granted with the WITH GRANT OPTION clause are revoked when the grantor's privilege is revoked.

The example on the slide gives user Scott access to your DEPARTMENTS table with the privileges to query the table and add rows to the table. The example also allows Scott to give others these privileges.

The PUBLIC Keyword

An owner of a table can grant access to all users by using the PUBLIC keyword.

The second example allows all users on the system to query data from Alice's DEPARTMENTS table.

Confirming Privileges Granted

Data Dictionary View	Description
ROLE_SYS_PRIVS	System privileges granted to roles
ROLE_TAB_PRIVS	Table privileges granted to roles
USER_ROLE_PRIVS	Roles accessible by the user
USER_TAB_PRIVS_MADE	Object privileges granted on the user's objects
USER_TAB_PRIVS_RECD	Object privileges granted to the user
USER_COL_PRIVS_MADE	Object privileges granted on the columns of the user's objects
USER_COL_PRIVS_RECD	Object privileges granted to the user on specific columns
USER_SYS_PRIVS	Lists system privileges granted to the user

Confirming Granted Privileges

If you attempt to perform an unauthorized operation, such as deleting a row from a table for which you do not have the DELETE privilege, the Oracle server does not permit the operation to take place.

If you receive the Oracle server error message "table or view does not exist," you have done either of the following:

- Named a table or view that does not exist
- Attempted to perform an operation on a table or view for which you do not have the appropriate privilege

You can access the data dictionary to view the privileges that you have. The chart on the slide describes various data dictionary views.

How to Revoke Object Privileges

- **You use the `REVOKE` statement to revoke privileges granted to other users.**
- **Privileges granted to others through the `WITH GRANT OPTION` clause are also revoked.**

```
REVOKE {privilege [, privilege...]|ALL}
ON     object
FROM   {user[, user...]|role|PUBLIC}
[CASCADE CONSTRAINTS];
```

ORACLE

Revoking Object Privileges

You can remove privileges granted to other users by using the `REVOKE` statement. When you use the `REVOKE` statement, the privileges that you specify are revoked from the users you name and from any other users to whom those privileges were granted through the `WITH GRANT OPTION` clause.

In the syntax:

`CASCADE CONSTRAINTS`	is required to remove any referential integrity constraints made to the object by means of the `REFERENCES` privilege

Revoking Object Privileges

As user Alice, revoke the `SELECT` and `INSERT` privileges given to user Scott on the `DEPARTMENTS` table.

```
REVOKE  select, insert
ON      departments
FROM    scott;
Revoke succeeded.
```

Revoking Object Privileges (continued)

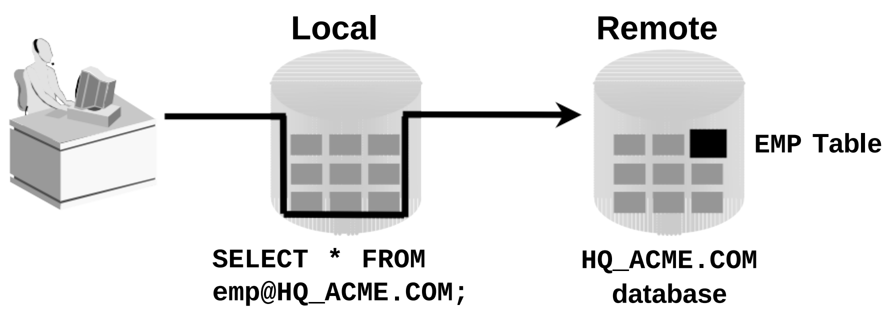
The example on the slide revokes SELECT and INSERT privileges given to user Scott on the DEPARTMENTS table.

Note: If a user is granted a privilege with the WITH GRANT OPTION clause, that user can also grant the privilege with the WITH GRANT OPTION clause, so that a long chain of grantees is possible, but no circular grants are permitted. If the owner revokes a privilege from a user who granted the privilege to other users, the revoking cascades to all privileges granted.

For example, if user A grants SELECT privilege on a table to user B including the WITH GRANT OPTION clause, user B can grant to user C the SELECT privilege with the WITH GRANT OPTION clause as well, and user C can then grant to user D the SELECT privilege. If user A revokes privilege from user B, then the privileges granted to users C and D are also revoked.

Database Links

A database link connection allows local users to access data on a remote database.



ORACLE

Database Links

A database link is a pointer that defines a one-way communication path from an Oracle database server to another database server. The link pointer is actually defined as an entry in a data dictionary table. To access the link, you must be connected to the local database that contains the data dictionary entry.

A database link connection is one-way in the sense that a client connected to local database A can use a link stored in database A to access information in remote database B, but users connected to database B cannot use the same link to access data in database A. If local users on database B want to access data on database A, they must define a link that is stored in the data dictionary of database B.

A database link connection gives local users access to data on a remote database. For this connection to occur, each database in the distributed system must have a unique global database name. The global database name uniquely identifies a database server in a distributed system.

The great advantage of database links is that they allow users to access another user's objects in a remote database so that they are bounded by the privilege set of the object's owner. In other words, a local user can access a remote database without having to be a user on the remote database.

The example shows a user `SCOTT` accessing the `EMP` table on the remote database with the global name `HQ.ACME.COM`.

Note: Typically, the DBA is responsible for creating the database link. The dictionary view `USER_DB_LINKS` contains information on links to which a user has access.

Database Links

- **Create the database link.**

```
CREATE PUBLIC DATABASE LINK hq.acme.com
USING ’sales’;
Database link created.
```

- **Write SQL statements that use the database link.**

```
SELECT *
FROM emp@HQ.ACME.COM;
```

ORACLE

Using Database Links

The example shown creates a database link. The `USING` clause identifies the service name of a remote database.

Once the database link is created, you can write SQL statements against the data in the remote site. If a synonym is set up, you can write SQL statements using the synonym.

For example:

```
CREATE PUBLIC SYNONYM HQ_EMP FOR emp@HQ.ACME.COM;
```

Then write a SQL statement that uses the synonym:

```
SELECT * FROM HQ_EMP;
```

You cannot grant privileges on remote objects.

Summary

In this lesson, you should have learned about DCL statements that control access to the database and database objects:

Statement	Action
`CREATE USER`	Creates a user (usually performed by a DBA)
`GRANT`	Gives other users privileges to access the your objects
`CREATE ROLE`	Creates a collection of privileges (usually performed by a DBA)
`ALTER USER`	Changes a user's password
`REVOKE`	Removes privileges on an object from users

ORACLE

Summary

DBAs establish initial database security for users by assigning privileges to the users.

- The DBA creates users who must have a password. The DBA is also responsible for establishing the initial system privileges for a user.
- Once the user has created an object, the user can pass along any of the available object privileges to other users or to all users by using the `GRANT` statement.
- A DBA can create roles by using the `CREATE ROLE` statement to pass along a collection of system or object privileges to multiple users. Roles make granting and revoking privileges easier to maintain.
- Users can change their password by using the `ALTER USER` statement.
- You can remove privileges from users by using the `REVOKE` statement.
- With data dictionary views, users can view the privileges granted to them and those that are granted on their objects.
- With database links, you can access data on remote databases. Privileges cannot be granted on remote objects.

Practice 13

1. What privilege should a user be given to log on to the Oracle Server? Is this a system or an object privilege?

   ____

2. What privilege should a user be given to create tables?

   ____

3. If you create a table, who can pass along privileges to other users on your table?

   ____

4. You are the DBA. You are creating many users who require the same system privileges. What should you use to make your job easier?

   ____

5. What command do you use to change your password?

   ____

6. Grant another user access to your `DEPARTMENTS` table. Have the user grant you query access to his or her `DEPARTMENTS` table.
7. Query all the rows in your `DEPARTMENTS` table.

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

8 rows selected.

8. Add a new row to your `DEPARTMENTS` table. Team 1 should add Education as department number 500. Team 2 should add Human Resources department number 510. Query the other team's table.
9. Create a synonym for the other team's `DEPARTMENTS` table.

10. Query all the rows in the other team's `DEPARTMENTS` table by using your synonym.

Team 1 SELECT statement results:

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

9 rows selected.

Team 2 SELECT statement results:

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

9 rows selected.

11. Query the `USER_TABLES` data dictionary to see information about the tables that you own.

TABLE_NAME
COUNTRIES
DEPARTMENTS
DEPT
EMP
EMPLOYEES
JOBS
JOB_GRADES
JOB_HISTORY
LOCATIONS
REGIONS

10 rows selected.

12. Query the `ALL_TABLES` data dictionary view to see information about all the tables that you can access. Exclude tables that you own.

    Note: Your list may not exactly match the list shown below.

TABLE_NAME	OWNER
DEPARTMENTS	***owner***

13. Revoke the `SELECT` privilege on your table from the other team.

14. Remove the row you inserted into the `DEPARTMENTS` table in step 8 and save the changes.