Introduction to Oracle9i: SQL

Electronic Presentation
Introduction
Objectives

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

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

- Scalability
- Reliability
- Single development model
- Common skill sets
- One vendor
- One management interface

Copyright © Oracle Corporation, 2001. All rights reserved.
Oracle9i Application Server

<table>
<thead>
<tr>
<th>APACHE</th>
<th>Portals</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>![Portals Image]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>APACHE</th>
<th>Transactional Apps</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>![Transactional Apps Image]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>APACHE</th>
<th>Business Intelligence</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>![Business Intelligence Image]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>APACHE</th>
<th>Integration</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>![Integration Image]</td>
</tr>
</tbody>
</table>
Oracle9i Database

Object Relational Data

Documents

Multimedia

Messages
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 Internet Platform

Clients

Any browser

Any mail client

Any FTP client

Internet applications

Business logic and data

Presentation and business logic

Databases

Application servers

Network services

System management

Development tools

Java

SQL

PL/SQL
System Development Life Cycle

- Strategy and analysis
- Design
- Build and document
- Transition
- Production
Data Storage on Different Media

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>DEPARTMENT_NAME</th>
<th>MANAGER_ID</th>
<th>LOCATION_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Administration</td>
<td>200</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>Marketing</td>
<td>201</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>Shipping</td>
<td>124</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>IT</td>
<td>103</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>Sales</td>
<td>149</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>Executive</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>Accounting</td>
<td>203</td>
<td></td>
</tr>
<tr>
<td>190</td>
<td>Contracting</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GRA</th>
<th>LOWEST_SAL</th>
<th>HIGHEST_SAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1000</td>
<td>2999</td>
</tr>
<tr>
<td>B</td>
<td>3000</td>
<td>5999</td>
</tr>
<tr>
<td>C</td>
<td>6000</td>
<td>9999</td>
</tr>
<tr>
<td>D</td>
<td>10000</td>
<td>14999</td>
</tr>
<tr>
<td>E</td>
<td>15000</td>
<td>24999</td>
</tr>
<tr>
<td>F</td>
<td>25000</td>
<td>40000</td>
</tr>
</tbody>
</table>

Electronic spreadsheet

Filing cabinet

Database
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
Definition of a Relational Database

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

Table Name: EMPLOYEES

Table Name: DEPARTMENTS
Data Models

Model of system in client’s mind → Entity model of client’s model → Table model of entity model → Oracle server

Tables on disk
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 . . .”
Entity Relationship Modeling Conventions

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

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

---

**EMPLOYEE**
- #* number
- * name
- o job title

**DEPARTMENT**
- #* number
- * name
- o location

**Unique Identifier (UID)**
- Primary marked with “#”
- Secondary marked with “(#)”
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

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>174</td>
<td>Ellen</td>
<td>Abel</td>
<td>80</td>
</tr>
<tr>
<td>142</td>
<td>Curtis</td>
<td>Davies</td>
<td>50</td>
</tr>
<tr>
<td>102</td>
<td>Lex</td>
<td>De Hasan</td>
<td>90</td>
</tr>
<tr>
<td>104</td>
<td>Bruce</td>
<td>Ernst</td>
<td>60</td>
</tr>
<tr>
<td>202</td>
<td>Pat</td>
<td>Fay</td>
<td>20</td>
</tr>
<tr>
<td>206</td>
<td>William</td>
<td>Gietz</td>
<td>110</td>
</tr>
</tbody>
</table>

Table Name: DEPARTMENTS

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>DEPARTMENT_NAME</th>
<th>MANAGER_ID</th>
<th>LOCATION_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Administration</td>
<td>200</td>
<td>1700</td>
</tr>
<tr>
<td>20</td>
<td>Marketing</td>
<td>201</td>
<td>1800</td>
</tr>
<tr>
<td>50</td>
<td>Shipping</td>
<td>124</td>
<td>1500</td>
</tr>
<tr>
<td>60</td>
<td>IT</td>
<td>103</td>
<td>1400</td>
</tr>
<tr>
<td>80</td>
<td>Sales</td>
<td>149</td>
<td>2500</td>
</tr>
<tr>
<td>90</td>
<td>Executive</td>
<td>100</td>
<td>1700</td>
</tr>
<tr>
<td>110</td>
<td>Accounting</td>
<td>205</td>
<td>1700</td>
</tr>
<tr>
<td>190</td>
<td>Contracting</td>
<td></td>
<td>1700</td>
</tr>
</tbody>
</table>
### Relational Database Terminology

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

SQL statement is entered.

```sql
SELECT department_name
FROM departments;
```

Statement is sent to Oracle Server.

<table>
<thead>
<tr>
<th>DEPARTMENT_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Administration</td>
</tr>
<tr>
<td>Marketing</td>
</tr>
<tr>
<td>Shipping</td>
</tr>
<tr>
<td>IT</td>
</tr>
<tr>
<td>Sales</td>
</tr>
<tr>
<td>Executive</td>
</tr>
<tr>
<td>Accounting</td>
</tr>
<tr>
<td>Contracting</td>
</tr>
</tbody>
</table>
Relational Database Management System

User tables

Data dictionary
SQL Statements

<table>
<thead>
<tr>
<th>Statement</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT</td>
<td>Data retrieval</td>
</tr>
<tr>
<td>INSERT</td>
<td>Data manipulation language (DML)</td>
</tr>
<tr>
<td>UPDATE</td>
<td>Data manipulation language (DML)</td>
</tr>
<tr>
<td>DELETE</td>
<td>Data manipulation language (DML)</td>
</tr>
<tr>
<td>MERGE</td>
<td>Data manipulation language (DML)</td>
</tr>
<tr>
<td>CREATE</td>
<td>Data definition language (DDL)</td>
</tr>
<tr>
<td>ALTER</td>
<td>Data definition language (DDL)</td>
</tr>
<tr>
<td>DROP</td>
<td>Data definition language (DDL)</td>
</tr>
<tr>
<td>RENAME</td>
<td>Data definition language (DDL)</td>
</tr>
<tr>
<td>TRUNCATE</td>
<td>Data definition language (DDL)</td>
</tr>
<tr>
<td>COMMIT</td>
<td>Transaction control</td>
</tr>
<tr>
<td>ROLLBACK</td>
<td>Transaction control</td>
</tr>
<tr>
<td>SAVEPOINT</td>
<td>Transaction control</td>
</tr>
<tr>
<td>GRANT</td>
<td>Data control language (DCL)</td>
</tr>
<tr>
<td>REVOKE</td>
<td>Data control language (DCL)</td>
</tr>
</tbody>
</table>
# Tables Used in the Course

## EMPLOYEES

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>EMAIL</th>
<th>PHONE_NUMBER</th>
<th>HIRE_DATE</th>
<th>JOB_ID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Steven</td>
<td>King</td>
<td>SKING</td>
<td>515.123.4567</td>
<td>17-JUN-97</td>
<td>AD_PRES</td>
<td>24000</td>
</tr>
<tr>
<td>101</td>
<td>Neena</td>
<td>Kochhar</td>
<td>NKOCHHAR</td>
<td>515.123.4568</td>
<td>21-SEP-89</td>
<td>AD_VP</td>
<td>17000</td>
</tr>
<tr>
<td>102</td>
<td>Lex</td>
<td>De Haan</td>
<td>LDEHAAN</td>
<td>515.123.4569</td>
<td>13-JAN-93</td>
<td>AD_VP</td>
<td>17000</td>
</tr>
<tr>
<td>103</td>
<td>Alexander</td>
<td>Hunold</td>
<td>AHUNOLD</td>
<td>590.423.4567</td>
<td>03-JAN-90</td>
<td>IT_PROG</td>
<td>9000</td>
</tr>
<tr>
<td>104</td>
<td>Bruce</td>
<td>Einst</td>
<td>BERNST</td>
<td>590.423.4568</td>
<td>21-MAY-91</td>
<td>IT_PROG</td>
<td>6000</td>
</tr>
<tr>
<td>107</td>
<td>Diana</td>
<td>Lorentz</td>
<td>DLORENTZ</td>
<td>590.423.5567</td>
<td>07-FEB-99</td>
<td>IT_PROG</td>
<td>4200</td>
</tr>
<tr>
<td>124</td>
<td>Kevin</td>
<td>Mourgos</td>
<td>KMOURGOS</td>
<td>650.123.5234</td>
<td>16-NOV-99</td>
<td>ST_MAN</td>
<td>5800</td>
</tr>
<tr>
<td>141</td>
<td>Trenna</td>
<td>Rajs</td>
<td>TRAJS</td>
<td>650.121.3009</td>
<td>17-OCT-95</td>
<td>ST_CLERK</td>
<td>3600</td>
</tr>
<tr>
<td>142</td>
<td>Curtis</td>
<td>Davies</td>
<td>CDAVIES</td>
<td>650.121.2994</td>
<td>29-JAN-97</td>
<td>ST_CLERK</td>
<td>3100</td>
</tr>
<tr>
<td>143</td>
<td>Randall</td>
<td>Matos</td>
<td>RMATOS</td>
<td>650.121.2874</td>
<td>15-MAR-98</td>
<td>ST_CLERK</td>
<td>2900</td>
</tr>
</tbody>
</table>

## DEPARTMENTS

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>DEPARTMENT_NAME</th>
<th>MANAGER_ID</th>
<th>LOCATION_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Administration</td>
<td>200</td>
<td>1700</td>
</tr>
<tr>
<td>20</td>
<td>Marketing</td>
<td>201</td>
<td>1800</td>
</tr>
<tr>
<td>50</td>
<td>Shipping</td>
<td>124</td>
<td>1500</td>
</tr>
<tr>
<td>60</td>
<td>IT</td>
<td>103</td>
<td>1400</td>
</tr>
<tr>
<td>80</td>
<td>Sales</td>
<td>149</td>
<td>2500</td>
</tr>
<tr>
<td>90</td>
<td>Executive</td>
<td>100</td>
<td>1700</td>
</tr>
<tr>
<td>110</td>
<td>Accounting</td>
<td>205</td>
<td>1700</td>
</tr>
<tr>
<td>190</td>
<td>Contracting</td>
<td></td>
<td>1700</td>
</tr>
</tbody>
</table>

## JOB_GRADES

<table>
<thead>
<tr>
<th>GRA</th>
<th>LOWEST_SAL</th>
<th>HIGHEST_SAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1000</td>
<td>2999</td>
</tr>
<tr>
<td>B</td>
<td>3000</td>
<td>6999</td>
</tr>
<tr>
<td>C</td>
<td>6000</td>
<td>9999</td>
</tr>
<tr>
<td>D</td>
<td>10000</td>
<td>14999</td>
</tr>
<tr>
<td>E</td>
<td>15000</td>
<td>24999</td>
</tr>
<tr>
<td>F</td>
<td>25000</td>
<td>40000</td>
</tr>
</tbody>
</table>
Summary

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

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

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

Projection

Table 1

Selection

Table 1

Table 1

Join

Table 1

Table 2
Basic **SELECT** Statement

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

- **SELECT** identifies *what* columns
- **FROM** identifies *which* table
### Selecting All Columns

```sql
SELECT * FROM departments;
```

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>DEPARTMENT_NAME</th>
<th>MANAGER_ID</th>
<th>LOCATION_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Administration</td>
<td>200</td>
<td>1700</td>
</tr>
<tr>
<td>20</td>
<td>Marketing</td>
<td>201</td>
<td>1800</td>
</tr>
<tr>
<td>50</td>
<td>Shipping</td>
<td>124</td>
<td>1500</td>
</tr>
<tr>
<td>60</td>
<td>IT</td>
<td>103</td>
<td>1400</td>
</tr>
<tr>
<td>80</td>
<td>Sales</td>
<td>149</td>
<td>2500</td>
</tr>
<tr>
<td>90</td>
<td>Executive</td>
<td>100</td>
<td>1700</td>
</tr>
<tr>
<td>110</td>
<td>Accounting</td>
<td>205</td>
<td>1700</td>
</tr>
<tr>
<td>190</td>
<td>Contracting</td>
<td></td>
<td>1700</td>
</tr>
</tbody>
</table>

8 rows selected.
Selecting Specific Columns

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

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>LOCATION_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1700</td>
</tr>
<tr>
<td>20</td>
<td>1800</td>
</tr>
<tr>
<td>50</td>
<td>1500</td>
</tr>
<tr>
<td>60</td>
<td>1400</td>
</tr>
<tr>
<td>80</td>
<td>2500</td>
</tr>
<tr>
<td>90</td>
<td>1700</td>
</tr>
<tr>
<td>110</td>
<td>1700</td>
</tr>
<tr>
<td>190</td>
<td>1700</td>
</tr>
</tbody>
</table>

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.
Column Heading Defaults

• **iSQL*Plus:**
  – Default heading justification: Center
  – Default heading display: Uppercase

• **SQL*Plus:**
  – Character and Date column headings are left-justified
  – Number column headings are right-justified
  – Default heading display: Uppercase
Arithmetic Expressions

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Add</td>
</tr>
<tr>
<td>-</td>
<td>Subtract</td>
</tr>
<tr>
<td>*</td>
<td>Multiply</td>
</tr>
<tr>
<td>/</td>
<td>Divide</td>
</tr>
</tbody>
</table>
### Using Arithmetic Operators

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

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>SALARY</th>
<th>SALARY+300</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td>24000</td>
<td>24300</td>
</tr>
<tr>
<td>Kochhar</td>
<td>17000</td>
<td>17300</td>
</tr>
<tr>
<td>De Haan</td>
<td>17000</td>
<td>17300</td>
</tr>
<tr>
<td>Hunold</td>
<td>9000</td>
<td>9300</td>
</tr>
<tr>
<td>Ernst</td>
<td>6000</td>
<td>6300</td>
</tr>
<tr>
<td>Hartstein</td>
<td>13000</td>
<td>13300</td>
</tr>
<tr>
<td>Fay</td>
<td>5000</td>
<td>5300</td>
</tr>
<tr>
<td>Higgins</td>
<td>12000</td>
<td>12300</td>
</tr>
<tr>
<td>Gietz</td>
<td>8300</td>
<td>8600</td>
</tr>
</tbody>
</table>

20 rows selected.
Operator Precedence

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

SELECT last_name, salary, \textcolor{red}{12\times\text{salary} + 100}\ FROM employees;

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>SALARY</th>
<th>12\times\text{SALARY} + 100</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td>24000</td>
<td>288100</td>
</tr>
<tr>
<td>Kochhar</td>
<td>17000</td>
<td>204100</td>
</tr>
<tr>
<td>De Haan</td>
<td>17000</td>
<td>204100</td>
</tr>
<tr>
<td>Hunold</td>
<td>9000</td>
<td>108100</td>
</tr>
<tr>
<td>Ernst</td>
<td>6000</td>
<td>72100</td>
</tr>
<tr>
<td>Hartstein</td>
<td>13000</td>
<td>156100</td>
</tr>
<tr>
<td>Fay</td>
<td>6000</td>
<td>72100</td>
</tr>
<tr>
<td>Higgins</td>
<td>12000</td>
<td>144100</td>
</tr>
<tr>
<td>Gietz</td>
<td>8300</td>
<td>99700</td>
</tr>
</tbody>
</table>

20 rows selected.
Using Parentheses

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

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>SALARY</th>
<th>12*(SALARY+100)</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td>24000</td>
<td>285200</td>
</tr>
<tr>
<td>Kochhar</td>
<td>17000</td>
<td>205200</td>
</tr>
<tr>
<td>De Haan</td>
<td>17000</td>
<td>205200</td>
</tr>
<tr>
<td>Hunold</td>
<td>9000</td>
<td>109200</td>
</tr>
<tr>
<td>Ernst</td>
<td>6000</td>
<td>73200</td>
</tr>
</tbody>
</table>

...  

| Hartstein | 13000  | 157200          |
| Fay       | 6000   | 73200           |
| Higgins   | 12000  | 145200          |
| Gietz     | 8300   | 100800          |

20 rows selected.
Defining a Null Value

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

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

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>JOB_ID</th>
<th>SALARY</th>
<th>COMMISSION_PCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td>AD_PRES</td>
<td>24000</td>
<td></td>
</tr>
<tr>
<td>Kochhar</td>
<td>AD_VP</td>
<td>17000</td>
<td></td>
</tr>
<tr>
<td>Zlotkey</td>
<td>SA_MAN</td>
<td>10500</td>
<td>.2</td>
</tr>
<tr>
<td>Abel</td>
<td>SA_REP</td>
<td>11000</td>
<td>.3</td>
</tr>
<tr>
<td>Taylor</td>
<td>SA_REP</td>
<td>8600</td>
<td>.2</td>
</tr>
<tr>
<td>Gietz</td>
<td>AC_ACCOUNT</td>
<td>8300</td>
<td></td>
</tr>
</tbody>
</table>

20 rows selected.
Null Values in Arithmetic Expressions

Arithmetic expressions containing a null value evaluate to null.

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

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>12<em>salary</em>commission_pct</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td></td>
</tr>
<tr>
<td>Kochhar</td>
<td></td>
</tr>
<tr>
<td>Zlotkey</td>
<td>25200</td>
</tr>
<tr>
<td>Abel</td>
<td>39600</td>
</tr>
<tr>
<td>Taylor</td>
<td>20640</td>
</tr>
<tr>
<td>Gietz</td>
<td></td>
</tr>
</tbody>
</table>

20 rows selected.
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
Using Column Aliases

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

<table>
<thead>
<tr>
<th>NAME</th>
<th>COMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td></td>
</tr>
<tr>
<td>Kochhar</td>
<td></td>
</tr>
<tr>
<td>De Haan</td>
<td></td>
</tr>
</tbody>
</table>

... 20 rows selected.

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Annual Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td>288000</td>
</tr>
<tr>
<td>Kochhar</td>
<td>204000</td>
</tr>
<tr>
<td>De Haan</td>
<td>204000</td>
</tr>
</tbody>
</table>

... 20 rows selected.
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
Using the Concatenation Operator

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

<table>
<thead>
<tr>
<th>Employees</th>
</tr>
</thead>
<tbody>
<tr>
<td>KingAD_PRES</td>
</tr>
<tr>
<td>KochharAD_VP</td>
</tr>
<tr>
<td>De HaanAD_VP</td>
</tr>
<tr>
<td>HunoldIT_PROG</td>
</tr>
<tr>
<td>ErnstIT_PROG</td>
</tr>
<tr>
<td>LorentzIT_PROG</td>
</tr>
<tr>
<td>MourgosST_MAN</td>
</tr>
<tr>
<td>RajsST_CLERK</td>
</tr>
</tbody>
</table>

... 20 rows selected.
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.
Using Literal Character Strings

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

<table>
<thead>
<tr>
<th>Employee Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>King is a AD_PRES</td>
</tr>
<tr>
<td>Kochhar is a AD_VP</td>
</tr>
<tr>
<td>De Haan is a AD_VP</td>
</tr>
<tr>
<td>Hunold is a IT_PROG</td>
</tr>
<tr>
<td>Ernst is a IT_PROG</td>
</tr>
<tr>
<td>Lorentz is a IT_PROG</td>
</tr>
<tr>
<td>Mourgos is a ST_MAN</td>
</tr>
<tr>
<td>Rajs is a ST_CLERK</td>
</tr>
</tbody>
</table>

... 20 rows selected.
Duplicate Rows

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

```
SELECT department_id
FROM   employees;
```

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>90</td>
</tr>
<tr>
<td>90</td>
</tr>
<tr>
<td>90</td>
</tr>
<tr>
<td>90</td>
</tr>
<tr>
<td>60</td>
</tr>
<tr>
<td>60</td>
</tr>
<tr>
<td>60</td>
</tr>
<tr>
<td>60</td>
</tr>
<tr>
<td>60</td>
</tr>
<tr>
<td>50</td>
</tr>
<tr>
<td>50</td>
</tr>
<tr>
<td>50</td>
</tr>
</tbody>
</table>

20 rows selected.
Eliminating Duplicate Rows

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

```
SELECT DISTINCT department_id
FROM employees;
```
SQL and iSQL*Plus Interaction

- iSQL*Plus
- Internet Browser
- SQL statements
- Client
- iSQL*Plus commands
- Query results
- Formatted report
- Oracle server
- Internet Browser
- Internet Browser
- Internet Browser
- Internet Browser

Copyright © Oracle Corporation, 2001. All rights reserved.
### SQL Statements Versus iSQL*Plus Commands

<table>
<thead>
<tr>
<th>SQL</th>
<th>iSQL*Plus</th>
</tr>
</thead>
<tbody>
<tr>
<td>A language</td>
<td>An environment</td>
</tr>
<tr>
<td>ANSI standard</td>
<td>Oracle proprietary</td>
</tr>
<tr>
<td>Keyword cannot be abbreviated</td>
<td>Keywords can be abbreviated</td>
</tr>
<tr>
<td>Statements manipulate data and table definitions in the database</td>
<td>Commands do not allow manipulation of values in the database</td>
</tr>
<tr>
<td></td>
<td>Runs on a browser</td>
</tr>
<tr>
<td></td>
<td>Centrally loaded, does not have to be implemented on each machine</td>
</tr>
</tbody>
</table>

**SQL statements**

**iSQL*Plus commands**
Overview of iSQL*Plus

After you log into iSQL*Plus, you can:

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

From your Windows browser environment:

**Username:**

**Password:**

**Connection Identifier:**

**Privilege:**

[Log In] [Clear]
Displaying Table Structure

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

DESC[RIBE] tablename
### Displaying Table Structure

**DESCRIBE employees**

<table>
<thead>
<tr>
<th>Name</th>
<th>Null?</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPLOYEE_ID</td>
<td>NOT NULL</td>
<td>NUMBER(6)</td>
</tr>
<tr>
<td>FIRST_NAME</td>
<td></td>
<td>VARCHAR2(20)</td>
</tr>
<tr>
<td>LAST_NAME</td>
<td>NOT NULL</td>
<td>VARCHAR2(25)</td>
</tr>
<tr>
<td>EMAIL</td>
<td>NOT NULL</td>
<td>VARCHAR2(25)</td>
</tr>
<tr>
<td>PHONE_NUMBER</td>
<td></td>
<td>VARCHAR2(20)</td>
</tr>
<tr>
<td>HIRE_DATE</td>
<td>NOT NULL</td>
<td>DATE</td>
</tr>
<tr>
<td>JOB_ID</td>
<td>NOT NULL</td>
<td>VARCHAR2(10)</td>
</tr>
<tr>
<td>SALARY</td>
<td></td>
<td>NUMBER(8,2)</td>
</tr>
<tr>
<td>COMMISSION_PCT</td>
<td></td>
<td>NUMBER(2,2)</td>
</tr>
<tr>
<td>MANAGER_ID</td>
<td></td>
<td>NUMBER(6)</td>
</tr>
<tr>
<td>DEPARTMENT_ID</td>
<td></td>
<td>NUMBER(4)</td>
</tr>
</tbody>
</table>
Interacting with Script Files

SELECT last_name, hire_date, salary
FROM employees;

1. Enter statements:

2. Execute
Interacting with Script Files

1. Script Location: D:\temp\emp_sql.htm

2. Enter statements:
   
   ```sql
   SELECT last_name, hire_date, salary 
   FROM employees;
   ```

3. Execute
Interacting with Script Files

```sql
DESCRIBE employees;
SELECT first_name, last_name, job_id
FROM employees;
```
Summary

In this lesson, you should have learned how to:

• Write a `SELECT` statement that:
  – Returns all rows and columns from a table
  – Returns specified columns from a table
  – Uses column aliases to give descriptive column headings

• Use the `iSQL*Plus` environment to write, save, and execute SQL statements and `iSQL*Plus` commands.

```sql
SELECT  * | {[DISTINCT] column | expression [alias],...} 
FROM     table;
```
Practice 1 Overview

This practice covers the following topics:

• Selecting all data from different tables
• Describing the structure of tables
• Performing arithmetic calculations and specifying column names
• Using iSQL*Plus
Restricting and Sorting Data
Objectives

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

• Limit the rows retrieved by a query
• Sort the rows retrieved by a query
# Limiting Rows Using a Selection

The EMPLOYEES table contains information about employees in a company. To limit the rows retrieved, you can use a selection clause in the SQL query. For example, to retrieve all employees in department 90, you can use the following query:

```
SELECT * FROM EMPLOYEES WHERE DEPARTMENT_ID = 90;
```

This query will return all rows from the EMPLOYEES table where the DEPARTMENT_ID is 90. The result includes employees like King, Kochhar, De Haan, Hunold, Ernst, Lorentz, and Mourgos.

Here is the table with selected rows:

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>JOB_ID</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>King</td>
<td>AD_PRES</td>
<td>90</td>
</tr>
<tr>
<td>101</td>
<td>Kochhar</td>
<td>AD_VP</td>
<td>90</td>
</tr>
<tr>
<td>102</td>
<td>De Haan</td>
<td>AD_VP</td>
<td>90</td>
</tr>
<tr>
<td>103</td>
<td>Hunold</td>
<td>IT_PROG</td>
<td>60</td>
</tr>
<tr>
<td>104</td>
<td>Ernst</td>
<td>IT_PROG</td>
<td>60</td>
</tr>
<tr>
<td>107</td>
<td>Lorentz</td>
<td>IT_PROG</td>
<td>60</td>
</tr>
<tr>
<td>124</td>
<td>Mourgos</td>
<td>ST_MAN</td>
<td>50</td>
</tr>
</tbody>
</table>

20 rows selected.

In summary, by using a selection clause in the SQL query, you can limit the rows returned from the EMPLOYEES table to those in a specific department.
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.
Using the **WHERE** Clause

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

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>JOB_ID</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>King</td>
<td>AD_PRES</td>
<td>90</td>
</tr>
<tr>
<td>101</td>
<td>Kochhar</td>
<td>AD_VP</td>
<td>90</td>
</tr>
<tr>
<td>102</td>
<td>De Haan</td>
<td>AD_VP</td>
<td>90</td>
</tr>
</tbody>
</table>
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.

```sql
SELECT last_name, job_id, department_id
FROM employees
WHERE last_name = 'Whalen';
```
## Comparison Conditions

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Equal to</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Not equal to</td>
</tr>
</tbody>
</table>
Using Comparison Conditions

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

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matos</td>
<td>2600</td>
</tr>
<tr>
<td>Vargas</td>
<td>2500</td>
</tr>
</tbody>
</table>
Other Comparison Conditions

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>BETWEEN ...AND...</td>
<td>Between two values (inclusive),</td>
</tr>
<tr>
<td>IN(set)</td>
<td>Match any of a list of values</td>
</tr>
<tr>
<td>LIKE</td>
<td>Match a character pattern</td>
</tr>
<tr>
<td>IS NULL</td>
<td>Is a null value</td>
</tr>
</tbody>
</table>
Using the **BETWEEN** Condition

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

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

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rajs</td>
<td>3500</td>
</tr>
<tr>
<td>Davies</td>
<td>3100</td>
</tr>
<tr>
<td>Matos</td>
<td>2600</td>
</tr>
<tr>
<td>Vargas</td>
<td>2500</td>
</tr>
</tbody>
</table>
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);
```

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>SALARY</th>
<th>MANAGER_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>202</td>
<td>Fay</td>
<td>6000</td>
<td>201</td>
</tr>
<tr>
<td>200</td>
<td>Whalen</td>
<td>4400</td>
<td>101</td>
</tr>
<tr>
<td>205</td>
<td>Higgins</td>
<td>12000</td>
<td>101</td>
</tr>
<tr>
<td>101</td>
<td>Kochhar</td>
<td>17000</td>
<td>100</td>
</tr>
<tr>
<td>102</td>
<td>De Haan</td>
<td>17000</td>
<td>100</td>
</tr>
<tr>
<td>124</td>
<td>Murgos</td>
<td>5500</td>
<td>100</td>
</tr>
<tr>
<td>149</td>
<td>Zlotkey</td>
<td>10500</td>
<td>100</td>
</tr>
<tr>
<td>201</td>
<td>Hartstein</td>
<td>13000</td>
<td>100</td>
</tr>
</tbody>
</table>

8 rows selected.
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.

```sql
SELECT first_name
FROM employees
WHERE first_name LIKE 'S%';
```
Using the \texttt{LIKE} Condition

- You can combine pattern-matching characters.

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

- You can use the \texttt{ESCAPE} identifier to search for the actual \% and \_ symbols.

<table>
<thead>
<tr>
<th>LAST_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kochhar</td>
</tr>
<tr>
<td>Lorentz</td>
</tr>
<tr>
<td>Mourgos</td>
</tr>
</tbody>
</table>
Using the **NULL** Conditions

Test for nulls with the **IS NULL** operator.

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

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>MANAGER_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td></td>
</tr>
</tbody>
</table>
## Logical Conditions

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AND</strong></td>
<td>Returns <strong>TRUE</strong> if <em>both</em> component conditions are true</td>
</tr>
<tr>
<td><strong>OR</strong></td>
<td>Returns <strong>TRUE</strong> if <em>either</em> component condition is true</td>
</tr>
<tr>
<td><strong>NOT</strong></td>
<td>Returns <strong>TRUE</strong> if the following condition is false</td>
</tr>
</tbody>
</table>
AND requires both conditions to be true.

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

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>JOB_ID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>149</td>
<td>Zotkey</td>
<td>SA_MAN</td>
<td>10500</td>
</tr>
<tr>
<td>201</td>
<td>Hartstein</td>
<td>MK_MAN</td>
<td>13000</td>
</tr>
</tbody>
</table>
Using the **OR** Operator

**OR** requires either condition to be true.

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

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>JOB_ID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>King</td>
<td>AD_PRES</td>
<td>24000</td>
</tr>
<tr>
<td>101</td>
<td>Kochhar</td>
<td>AD_VP</td>
<td>17000</td>
</tr>
<tr>
<td>102</td>
<td>De Haan</td>
<td>AD_VP</td>
<td>17000</td>
</tr>
<tr>
<td>124</td>
<td>Mourgos</td>
<td>ST_MAN</td>
<td>5800</td>
</tr>
<tr>
<td>149</td>
<td>Zlotkey</td>
<td>SA_MAN</td>
<td>10500</td>
</tr>
<tr>
<td>174</td>
<td>Abel</td>
<td>SA_REP</td>
<td>11000</td>
</tr>
<tr>
<td>201</td>
<td>Hartstein</td>
<td>MK_MAN</td>
<td>13000</td>
</tr>
<tr>
<td>205</td>
<td>Higgins</td>
<td>AC_MGR</td>
<td>12000</td>
</tr>
</tbody>
</table>

8 rows selected.
Using the **NOT** Operator

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

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>JOB_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td>AD_PRES</td>
</tr>
<tr>
<td>Kochhar</td>
<td>AD_VP</td>
</tr>
<tr>
<td>De Haan</td>
<td>AD_VP</td>
</tr>
<tr>
<td>Mourgos</td>
<td>ST_MAN</td>
</tr>
<tr>
<td>Zlotkey</td>
<td>SA_MAN</td>
</tr>
<tr>
<td>Whalen</td>
<td>AD_ASST</td>
</tr>
<tr>
<td>Hartstein</td>
<td>MK_MAN</td>
</tr>
<tr>
<td>Fay</td>
<td>MK_REP</td>
</tr>
<tr>
<td>Higgins</td>
<td>AC_MGR</td>
</tr>
<tr>
<td>Gietz</td>
<td>AC_ACCOUNT</td>
</tr>
</tbody>
</table>

10 rows selected.
### Rules of Precedence

<table>
<thead>
<tr>
<th>Order Evaluated</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Arithmetic operators</td>
</tr>
<tr>
<td>2</td>
<td>Concatenation operator</td>
</tr>
<tr>
<td>3</td>
<td>Comparison conditions</td>
</tr>
<tr>
<td>4</td>
<td>IS [NOT] NULL, LIKE, [NOT] IN</td>
</tr>
<tr>
<td>5</td>
<td>[NOT] BETWEEN</td>
</tr>
<tr>
<td>6</td>
<td>NOT logical condition</td>
</tr>
<tr>
<td>7</td>
<td>AND logical condition</td>
</tr>
<tr>
<td>8</td>
<td>OR logical condition</td>
</tr>
</tbody>
</table>

Override rules of precedence by using parentheses.
Rules of Precedence

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

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>JOB_ID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td>AD_PRES</td>
<td>24000</td>
</tr>
<tr>
<td>Abel</td>
<td>SA_REP</td>
<td>11000</td>
</tr>
<tr>
<td>Taylor</td>
<td>SA_REP</td>
<td>8600</td>
</tr>
<tr>
<td>Grant</td>
<td>SA_REP</td>
<td>7000</td>
</tr>
</tbody>
</table>
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;
```

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>JOB_ID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td>AD_PRES</td>
<td>24000</td>
</tr>
</tbody>
</table>
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;
```

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>JOB_ID</th>
<th>DEPARTMENT_ID</th>
<th>HIRE_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td>AD_PRES</td>
<td></td>
<td>17-JUN-87</td>
</tr>
<tr>
<td>Whalen</td>
<td>AD_ASST</td>
<td></td>
<td>17-SEP-87</td>
</tr>
<tr>
<td>Kochhar</td>
<td>AD_VP</td>
<td>90</td>
<td>21-SEP-89</td>
</tr>
<tr>
<td>Hunold</td>
<td>IT_PROG</td>
<td>60</td>
<td>03-JAN-90</td>
</tr>
<tr>
<td>Ernst</td>
<td>IT_PROG</td>
<td></td>
<td>21-MAY-91</td>
</tr>
</tbody>
</table>

20 rows selected.
### Sorting in Descending Order

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

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>JOB_ID</th>
<th>DEPARTMENT_ID</th>
<th>HIRE_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zlotkey</td>
<td>SA_MAN</td>
<td>80</td>
<td>29-JAN-00</td>
</tr>
<tr>
<td>Mourgos</td>
<td>ST_MAN</td>
<td>50</td>
<td>16-NOV-99</td>
</tr>
<tr>
<td>Grant</td>
<td>SA_REP</td>
<td>20</td>
<td>24-MAY-99</td>
</tr>
<tr>
<td>Lorentz</td>
<td>IT_PROG</td>
<td>60</td>
<td>07-FEB-99</td>
</tr>
<tr>
<td>Vargas</td>
<td>ST_CLERK</td>
<td>50</td>
<td>09-JUL-98</td>
</tr>
<tr>
<td>Taylor</td>
<td>SA_REP</td>
<td>80</td>
<td>24-MAR-98</td>
</tr>
<tr>
<td>Matos</td>
<td>ST_CLERK</td>
<td>50</td>
<td>15-MAR-98</td>
</tr>
<tr>
<td>Fay</td>
<td>MKREP</td>
<td>20</td>
<td>17-AUG-97</td>
</tr>
<tr>
<td>Davies</td>
<td>ST_CLERK</td>
<td>50</td>
<td>29-JAN-97</td>
</tr>
</tbody>
</table>

20 rows selected.
Sorting by Column Alias

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

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>ANNSAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>144</td>
<td>Vargas</td>
<td>30000</td>
</tr>
<tr>
<td>143</td>
<td>Matos</td>
<td>31200</td>
</tr>
<tr>
<td>142</td>
<td>Davies</td>
<td>37200</td>
</tr>
<tr>
<td>141</td>
<td>Rajs</td>
<td>42000</td>
</tr>
<tr>
<td>107</td>
<td>Lorentz</td>
<td>50400</td>
</tr>
<tr>
<td>200</td>
<td>Whalen</td>
<td>52800</td>
</tr>
<tr>
<td>124</td>
<td>Mourgos</td>
<td>69600</td>
</tr>
<tr>
<td>104</td>
<td>Ernst</td>
<td>72000</td>
</tr>
<tr>
<td>202</td>
<td>Fay</td>
<td>72000</td>
</tr>
<tr>
<td>178</td>
<td>Grant</td>
<td>84000</td>
</tr>
</tbody>
</table>

20 rows selected.
Sorting by Multiple Columns

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

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

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>DEPARTMENT_ID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Whalen</td>
<td>10</td>
<td>4400</td>
</tr>
<tr>
<td>Hartstein</td>
<td>20</td>
<td>13000</td>
</tr>
<tr>
<td>Fay</td>
<td>20</td>
<td>6000</td>
</tr>
<tr>
<td>Murgos</td>
<td>50</td>
<td>5800</td>
</tr>
<tr>
<td>Rajs</td>
<td>50</td>
<td>3500</td>
</tr>
<tr>
<td>Davies</td>
<td>50</td>
<td>3100</td>
</tr>
<tr>
<td>Matos</td>
<td>50</td>
<td>2600</td>
</tr>
<tr>
<td>Vargas</td>
<td>50</td>
<td>2500</td>
</tr>
</tbody>
</table>

...  

20 rows selected.

- You can sort by a column that is not in the `SELECT` list.
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]]
```
Practice 2 Overview

This practice covers the following topics:

- Selecting data and changing the order of rows displayed
- Restricting rows by using the WHERE clause
- Sorting rows by using the ORDER BY clause
Single-Row Functions
Objectives

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

• Describe various types of functions available in SQL
• Use character, number, and date functions in SELECT statements
• Describe the use of conversion functions
SQL Functions

Function
performs action

Input:
- arg 1
- arg 2
- arg n

Output:
- Result value
Two Types of SQL Functions

Functions

Single-row functions

Multiple-row functions
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

\[ \text{function\_name } ([\text{arg1, arg2, ...}]) \]
Single-Row Functions

- Character
- General
- Conversion
- Date
- Number

Single-row functions
Character Functions

- **Case-manipulation functions**
  - LOWER
  - UPPER
  - INITCAP

- **Character-manipulation functions**
  - CONCAT
  - SUBSTR
  - LENGTH
  - INSTR
  - LPAD | RPAD
  - TRIM
  - REPLACE
Case Manipulation Functions

These functions convert case for character strings.

<table>
<thead>
<tr>
<th>Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOWER('SQL Course')</td>
<td>sql course</td>
</tr>
<tr>
<td>UPPER('SQL Course')</td>
<td>SQL COURSE</td>
</tr>
<tr>
<td>INITCAP('SQL Course')</td>
<td>Sql Course</td>
</tr>
</tbody>
</table>
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';
```

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>205</td>
<td>Higgins</td>
<td>110</td>
</tr>
</tbody>
</table>
Character-Manipulation Functions

These functions manipulate character strings:

<table>
<thead>
<tr>
<th>Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONCAT('Hello', 'World')</td>
<td>HelloWorld</td>
</tr>
<tr>
<td>SUBSTR('HelloWorld',1,5)</td>
<td>Hello</td>
</tr>
<tr>
<td>LENGTH('HelloWorld')</td>
<td>10</td>
</tr>
<tr>
<td>INSTR('HelloWorld', 'W')</td>
<td>6</td>
</tr>
<tr>
<td>LPAD(salary,10,'*')</td>
<td>*****24000</td>
</tr>
<tr>
<td>RPAD(salary, 10, '*')</td>
<td>24000*****</td>
</tr>
<tr>
<td>TRIM('H' FROM 'HelloWorld')</td>
<td>elloWorld</td>
</tr>
</tbody>
</table>
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';
```
Number Functions

- **ROUND**: Rounds value to specified decimal
  
  \[ \text{ROUND}(45.926, 2) \rightarrow 45.93 \]

- **TRUNC**: Truncates value to specified decimal
  
  \[ \text{TRUNC}(45.926, 2) \rightarrow 45.92 \]

- **MOD**: Returns remainder of division
  
  \[ \text{MOD}(1600, 300) \rightarrow 100 \]
Using the `ROUND` Function

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

**DUAL** is a dummy table you can use to view results from functions and calculations.
Using the TRUNC Function

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

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

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>SALARY</th>
<th>MOD(SALARY,5000)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abel</td>
<td>11000</td>
<td>1000</td>
</tr>
<tr>
<td>Taylor</td>
<td>8600</td>
<td>3600</td>
</tr>
<tr>
<td>Grant</td>
<td>7000</td>
<td>2000</td>
</tr>
</tbody>
</table>
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.

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

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>HIRE_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gietz</td>
<td>07-JUN-94</td>
</tr>
<tr>
<td>Grant</td>
<td>24-MAY-99</td>
</tr>
</tbody>
</table>
Working with Dates

SYSDATE is a function that returns:
• Date
• Time
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.
Using Arithmetic Operators with Dates

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

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>WEEKS</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td>744.245395</td>
</tr>
<tr>
<td>Kochhar</td>
<td>626.102538</td>
</tr>
<tr>
<td>De Haan</td>
<td>453.245395</td>
</tr>
</tbody>
</table>
# Date Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MONTHS_BETWEEN</td>
<td>Number of months between two dates</td>
</tr>
<tr>
<td>ADD_MONTHS</td>
<td>Add calendar months to date</td>
</tr>
<tr>
<td>NEXT_DAY</td>
<td>Next day of the date specified</td>
</tr>
<tr>
<td>LAST_DAY</td>
<td>Last day of the month</td>
</tr>
<tr>
<td>ROUND</td>
<td>Round date</td>
</tr>
<tr>
<td>TRUNC</td>
<td>Truncate date</td>
</tr>
</tbody>
</table>
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'
Using Date Functions

Assume \texttt{SYSDATE} = '25-JUL-95':

- \texttt{ROUND(SYSDATE, 'MONTH')} \rightarrow 01-AUG-95
- \texttt{ROUND(SYSDATE, 'YEAR')} \rightarrow 01-JAN-96
- \texttt{TRUNC(SYSDATE, 'MONTH')} \rightarrow 01-JUL-95
- \texttt{TRUNC(SYSDATE, 'YEAR')} \rightarrow 01-JAN-95
Practice 3, Part One: Overview

This practice covers the following topics:

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

Data type conversion

Implicit data type conversion
Explicit data type conversion
Implicit Data Type Conversion

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

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARCHAR2 or CHAR</td>
<td>NUMBER</td>
</tr>
<tr>
<td>VARCHAR2 or CHAR</td>
<td>DATE</td>
</tr>
<tr>
<td>NUMBER</td>
<td>VARCHAR2</td>
</tr>
<tr>
<td>DATE</td>
<td>VARCHAR2</td>
</tr>
</tbody>
</table>
Implicit Data Type Conversion

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

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARCHAR2 or CHAR</td>
<td>NUMBER</td>
</tr>
<tr>
<td>VARCHAR2 or CHAR</td>
<td>DATE</td>
</tr>
</tbody>
</table>
Explicit Data Type Conversion

- `TO_NUMBER`
- `TO_DATE`
- `TO_CHAR`

Types:
- NUMBER
- CHARACTER
- DATE

Copyright © Oracle Corporation, 2001. All rights reserved.
Explicit Data Type Conversion

NUMBER

TO_NUMBER

TO_CHAR

TO_DATE

DATE

CHARACTER
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
<table>
<thead>
<tr>
<th>Elements of the Date Format Model</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>YYYY</strong></td>
</tr>
<tr>
<td><strong>YEAR</strong></td>
</tr>
<tr>
<td><strong>MM</strong></td>
</tr>
<tr>
<td><strong>MONTH</strong></td>
</tr>
<tr>
<td><strong>MON</strong></td>
</tr>
<tr>
<td><strong>DY</strong></td>
</tr>
<tr>
<td><strong>DAY</strong></td>
</tr>
<tr>
<td><strong>DD</strong></td>
</tr>
</tbody>
</table>
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.
  
  ddsptth fourteenth
Using the `TO_CHAR` Function with Dates

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

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>HIREDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td>17 June 1987</td>
</tr>
<tr>
<td>Kochhar</td>
<td>21 September 1989</td>
</tr>
<tr>
<td>De Haan</td>
<td>13 January 1993</td>
</tr>
<tr>
<td>Hunold</td>
<td>3 January 1990</td>
</tr>
<tr>
<td>Ernst</td>
<td>21 May 1991</td>
</tr>
<tr>
<td>Lorentz</td>
<td>7 February 1999</td>
</tr>
<tr>
<td>Mourgos</td>
<td>16 November 1999</td>
</tr>
</tbody>
</table>

20 rows selected.
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:

<table>
<thead>
<tr>
<th>Format Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Represents a number</td>
</tr>
<tr>
<td>0</td>
<td>Forces a zero to be displayed</td>
</tr>
<tr>
<td>$</td>
<td>Places a floating dollar sign</td>
</tr>
<tr>
<td>L</td>
<td>Uses the floating local currency symbol</td>
</tr>
<tr>
<td>.</td>
<td>Prints a decimal point</td>
</tr>
<tr>
<td>,</td>
<td>Prints a thousand indicator</td>
</tr>
</tbody>
</table>
Using the `TO_CHAR` Function with Numbers

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

<table>
<thead>
<tr>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>$6,000.00</td>
</tr>
</tbody>
</table>
Using the TO_NUMBER and TO_DATE Functions

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

  \[
  \text{TO\_NUMBER}(\text{char}[, \ 'format\_model'\])
  \]

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

  \[
  \text{TO\_DATE}(\text{char}[, \ 'format\_model'\])
  \]

- These functions have an \texttt{fx} modifier. This modifier specifies the exact matching for the character argument and date format model of a TO_DATE function.
Using the `TO_NUMBER` and `TO_DATE` Functions

- Convert a character string to a number format using the `TO_NUMBER` function:
  
  ```sql
  TO_NUMBER(char[, 'format_model'])
  ```

- Convert a character string to a date format using the `TO_DATE` function:
  
  ```sql
  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.
## RR Date Format

<table>
<thead>
<tr>
<th>Current Year</th>
<th>Specified Date</th>
<th>RR Format</th>
<th>YY Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>1995</td>
<td>27-OCT-17</td>
<td>2017</td>
<td>1917</td>
</tr>
<tr>
<td>2001</td>
<td>27-OCT-17</td>
<td>2017</td>
<td>2017</td>
</tr>
<tr>
<td>2001</td>
<td>27-OCT-95</td>
<td>1995</td>
<td>2095</td>
</tr>
</tbody>
</table>

### If the specified two-digit year is:

<table>
<thead>
<tr>
<th>If two digits of the current year are:</th>
<th>0–49</th>
<th>50–99</th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tbody>
</table>
Example of \texttt{RR} Date Format

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

\begin{verbatim}
SELECT last_name, TO_CHAR(hire_date, 'DD-Mon-YYYY') 
FROM employees 
WHERE hire_date < TO_DATE('01-Jan-90', 'DD-Mon-RR');
\end{verbatim}

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>TO_CHAR(HIR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td>17-Jun-1987</td>
</tr>
<tr>
<td>Kochhar</td>
<td>21-Sep-1989</td>
</tr>
<tr>
<td>Whalen</td>
<td>17-Sep-1987</td>
</tr>
</tbody>
</table>
Nesting Functions

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

\[ F_3(F_2(F_1(col, arg_1), arg_2), arg_3) \]

- Step 1 = Result 1
- Step 2 = Result 2
- Step 3 = Result 3
Nesting Functions

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

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>NVL(TO_CHAR(MANAGER_ID),’NOMANAGER’)</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td>No Manager</td>
</tr>
</tbody>
</table>
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)
**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')`
Using the NVL Function

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

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>SALARY</th>
<th>NVL(COMMISSION_PCT,0)</th>
<th>AN_SAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td>24000</td>
<td>0</td>
<td>288000</td>
</tr>
<tr>
<td>Kochhar</td>
<td>17000</td>
<td>0</td>
<td>204000</td>
</tr>
<tr>
<td>De Haan</td>
<td>17000</td>
<td>0</td>
<td>204000</td>
</tr>
<tr>
<td>Hunold</td>
<td>9000</td>
<td>0</td>
<td>106000</td>
</tr>
<tr>
<td>Ernst</td>
<td>6000</td>
<td>0</td>
<td>72000</td>
</tr>
<tr>
<td>Lorentz</td>
<td>4200</td>
<td>0</td>
<td>50400</td>
</tr>
<tr>
<td>Mourgos</td>
<td>5800</td>
<td>0</td>
<td>69600</td>
</tr>
<tr>
<td>Rajs</td>
<td>3500</td>
<td>0</td>
<td>42000</td>
</tr>
</tbody>
</table>

20 rows selected.
Using the NVL2 Function

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

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>SALARY</th>
<th>COMMISSION_PCT</th>
<th>INCOME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zlotkey</td>
<td>10500</td>
<td>.2</td>
<td>SAL+COMM</td>
</tr>
<tr>
<td>Abel</td>
<td>11000</td>
<td>.3</td>
<td>SAL+COMM</td>
</tr>
<tr>
<td>Taylor</td>
<td>8600</td>
<td>.2</td>
<td>SAL+COMM</td>
</tr>
<tr>
<td>Mourgos</td>
<td>5800</td>
<td></td>
<td>SAL</td>
</tr>
<tr>
<td>Rajs</td>
<td>3500</td>
<td></td>
<td>SAL</td>
</tr>
<tr>
<td>Davies</td>
<td>3100</td>
<td></td>
<td>SAL</td>
</tr>
<tr>
<td>Matos</td>
<td>2600</td>
<td></td>
<td>SAL</td>
</tr>
<tr>
<td>Vargas</td>
<td>2500</td>
<td></td>
<td>SAL</td>
</tr>
</tbody>
</table>

8 rows selected.
Using the `NULLIF` Function

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

<table>
<thead>
<tr>
<th>FIRST_NAME</th>
<th>expr1</th>
<th>LAST_NAME</th>
<th>expr2</th>
<th>RESULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Steven</td>
<td>6</td>
<td>King</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>Neena</td>
<td>5</td>
<td>Kochhar</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>Lex</td>
<td>3</td>
<td>De Haan</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>Alexander</td>
<td>9</td>
<td>Hunold</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>Bruce</td>
<td>5</td>
<td>Ernst</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Diana</td>
<td>5</td>
<td>Lorentz</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>Kevin</td>
<td>5</td>
<td>Mourgos</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>Trenna</td>
<td>6</td>
<td>Rajs</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>Curtis</td>
<td>6</td>
<td>Davies</td>
<td></td>
<td>6</td>
</tr>
</tbody>
</table>

20 rows selected.
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.
Using the `COALESCE` Function

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

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>COMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grant</td>
<td>.15</td>
</tr>
<tr>
<td>Zlotkey</td>
<td>.2</td>
</tr>
<tr>
<td>Taylor</td>
<td>.2</td>
</tr>
<tr>
<td>Abel</td>
<td>.3</td>
</tr>
<tr>
<td>King</td>
<td></td>
</tr>
<tr>
<td>Kochhar</td>
<td>24000</td>
</tr>
<tr>
<td>De Haan</td>
<td>17000</td>
</tr>
<tr>
<td>Hunold</td>
<td>17000</td>
</tr>
<tr>
<td></td>
<td>9000</td>
</tr>
</tbody>
</table>

20 rows selected.
Conditional Expressions

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

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

```sql
CASE expr WHEN comparison_expr1 THEN return_expr1
  [WHEN comparison_expr2 THEN return_expr2
   WHEN comparison_exprn THEN return_exprn
   ELSE else_expr]
END
```
Using the **CASE** Expression

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

```sql
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;
```

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>JOB_ID</th>
<th>SALARY</th>
<th>REVISED_SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lorentz</td>
<td>IT_PROG</td>
<td>4200</td>
<td>4620</td>
</tr>
<tr>
<td>Mourgos</td>
<td>ST_MAN</td>
<td>5800</td>
<td>5800</td>
</tr>
<tr>
<td>Rajs</td>
<td>ST_CLERK</td>
<td>3500</td>
<td>4025</td>
</tr>
<tr>
<td>Gietz</td>
<td>AC_ACCOUNT</td>
<td>8300</td>
<td>8300</td>
</tr>
</tbody>
</table>

20 rows selected.
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])
```
Using the **DECODE** Function

```sql
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) AS REVISED_SALARY
FROM employees;
```

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>JOB_ID</th>
<th>SALARY</th>
<th>REVISED_SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lorentz</td>
<td>IT_PROG</td>
<td>4200</td>
<td>4620</td>
</tr>
<tr>
<td>Mourgos</td>
<td>ST_MAN</td>
<td>5800</td>
<td>6800</td>
</tr>
<tr>
<td>Rajs</td>
<td>ST_CLERK</td>
<td>3500</td>
<td>4025</td>
</tr>
<tr>
<td></td>
<td>AC_ACCOUNT</td>
<td>8300</td>
<td>8300</td>
</tr>
</tbody>
</table>

20 rows selected.
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;
```
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
Practice 3, Part Two: Overview

This practice covers the following topics:

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

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

• Write `SELECT` statements to access data from more than one table using equality and nonequality joins

• View data that generally does not meet a join condition by using outer joins

• Join a table to itself by using a self join
Obtaining Data from Multiple Tables

**EMPLOYEES**

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>King</td>
<td>90</td>
</tr>
<tr>
<td>101</td>
<td>Kochhar</td>
<td>90</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>202</td>
<td>Fay</td>
<td>20</td>
</tr>
<tr>
<td>205</td>
<td>Higgins</td>
<td>110</td>
</tr>
<tr>
<td>206</td>
<td>Gietz</td>
<td>110</td>
</tr>
</tbody>
</table>

**DEPARTMENTS**

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>DEPARTMENT_NAME</th>
<th>LOCATION_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Administration</td>
<td>1700</td>
</tr>
<tr>
<td>20</td>
<td>Marketing</td>
<td>1800</td>
</tr>
<tr>
<td>50</td>
<td>Shipping</td>
<td>1500</td>
</tr>
<tr>
<td>60</td>
<td>IT</td>
<td>1400</td>
</tr>
<tr>
<td>80</td>
<td>Sales</td>
<td>2500</td>
</tr>
<tr>
<td>90</td>
<td>Executive</td>
<td>1700</td>
</tr>
<tr>
<td>110</td>
<td>Accounting</td>
<td>1700</td>
</tr>
<tr>
<td>190</td>
<td>Contracting</td>
<td>1700</td>
</tr>
</tbody>
</table>

**EMPLOYEE_ID**

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>DEPARTMENT_ID</th>
<th>DEPARTMENT_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>10</td>
<td>Administration</td>
</tr>
<tr>
<td>201</td>
<td>20</td>
<td>Marketing</td>
</tr>
<tr>
<td>202</td>
<td>20</td>
<td>Marketing</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>102</td>
<td>90</td>
<td>Executive</td>
</tr>
<tr>
<td>205</td>
<td>110</td>
<td>Accounting</td>
</tr>
<tr>
<td>206</td>
<td>110</td>
<td>Accounting</td>
</tr>
</tbody>
</table>
Cartesian Products

• A Cartesian product is formed when:
  – A join condition is omitted
  – A join condition is invalid
  – All rows in the first table are joined to all rows in the second table

• To avoid a Cartesian product, always include a valid join condition in a WHERE clause.
Generating a Cartesian Product

**EMPLOYEES (20 rows)**

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>King</td>
<td>90</td>
</tr>
<tr>
<td>101</td>
<td>Kochhar</td>
<td>90</td>
</tr>
<tr>
<td>202</td>
<td>Fay</td>
<td>20</td>
</tr>
<tr>
<td>205</td>
<td>Higgins</td>
<td>110</td>
</tr>
<tr>
<td>206</td>
<td>Gietz</td>
<td>110</td>
</tr>
</tbody>
</table>

20 rows selected.

**DEPARTMENTS (8 rows)**

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>DEPARTMENT_NAME</th>
<th>LOCATION_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Administration</td>
<td>1700</td>
</tr>
<tr>
<td>20</td>
<td>Marketing</td>
<td>1800</td>
</tr>
<tr>
<td>50</td>
<td>Shipping</td>
<td>1500</td>
</tr>
<tr>
<td>60</td>
<td>IT</td>
<td>1400</td>
</tr>
<tr>
<td>80</td>
<td>Sales</td>
<td>2500</td>
</tr>
<tr>
<td>90</td>
<td>Executive</td>
<td>1700</td>
</tr>
<tr>
<td>110</td>
<td>Accounting</td>
<td>1700</td>
</tr>
<tr>
<td>190</td>
<td>Contracting</td>
<td>1700</td>
</tr>
</tbody>
</table>

8 rows selected.

**Cartesian product:**

\[20 \times 8 = 160 \text{ rows}\]
Types of Joins

Oracle Proprietary Joins (8i and prior):
• Equijoin
• Non-eqijoin
• 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
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.
What is an Equijoin?

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>10</td>
</tr>
<tr>
<td>201</td>
<td>20</td>
</tr>
<tr>
<td>202</td>
<td>20</td>
</tr>
<tr>
<td>124</td>
<td>50</td>
</tr>
<tr>
<td>141</td>
<td>50</td>
</tr>
<tr>
<td>142</td>
<td>50</td>
</tr>
<tr>
<td>143</td>
<td>50</td>
</tr>
<tr>
<td>144</td>
<td>50</td>
</tr>
<tr>
<td>103</td>
<td>60</td>
</tr>
<tr>
<td>104</td>
<td>60</td>
</tr>
<tr>
<td>107</td>
<td>60</td>
</tr>
<tr>
<td>149</td>
<td>80</td>
</tr>
<tr>
<td>174</td>
<td>80</td>
</tr>
<tr>
<td>176</td>
<td>80</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>DEPARTMENT_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Administration</td>
</tr>
<tr>
<td>20</td>
<td>Marketing</td>
</tr>
<tr>
<td>20</td>
<td>Marketing</td>
</tr>
<tr>
<td>50</td>
<td>Shipping</td>
</tr>
<tr>
<td>50</td>
<td>Shipping</td>
</tr>
<tr>
<td>50</td>
<td>Shipping</td>
</tr>
<tr>
<td>50</td>
<td>Shipping</td>
</tr>
<tr>
<td>60</td>
<td>IT</td>
</tr>
<tr>
<td>60</td>
<td>IT</td>
</tr>
<tr>
<td>60</td>
<td>IT</td>
</tr>
<tr>
<td>80</td>
<td>Sales</td>
</tr>
<tr>
<td>80</td>
<td>Sales</td>
</tr>
<tr>
<td>80</td>
<td>Sales</td>
</tr>
</tbody>
</table>
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;

Retrieving Records with Equijoins

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>DEPARTMENT_ID</th>
<th>DEPARTMENT_ID</th>
<th>LOCATION_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>Whalen</td>
<td>10</td>
<td>10</td>
<td>1700</td>
</tr>
<tr>
<td>201</td>
<td>Hartstein</td>
<td>20</td>
<td>20</td>
<td>1800</td>
</tr>
<tr>
<td>202</td>
<td>Fay</td>
<td>20</td>
<td>20</td>
<td>1800</td>
</tr>
<tr>
<td>124</td>
<td>Mourgos</td>
<td>50</td>
<td>50</td>
<td>1500</td>
</tr>
<tr>
<td>141</td>
<td>Rajs</td>
<td>50</td>
<td>50</td>
<td>1500</td>
</tr>
<tr>
<td>142</td>
<td>Davies</td>
<td>50</td>
<td>50</td>
<td>1500</td>
</tr>
<tr>
<td>143</td>
<td>Matos</td>
<td>50</td>
<td>50</td>
<td>1500</td>
</tr>
<tr>
<td>144</td>
<td>Vargas</td>
<td>50</td>
<td>50</td>
<td>1500</td>
</tr>
</tbody>
</table>

19 rows selected.
Additional Search Conditions Using the **AND** Operator

### EMPLOYEES

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Whalen</td>
<td>10</td>
</tr>
<tr>
<td>Hartslein</td>
<td>20</td>
</tr>
<tr>
<td>Fay</td>
<td>20</td>
</tr>
<tr>
<td>Mourgos</td>
<td>50</td>
</tr>
<tr>
<td>Rajs</td>
<td>50</td>
</tr>
<tr>
<td>Davies</td>
<td>50</td>
</tr>
<tr>
<td><strong>Matos</strong></td>
<td><strong>50</strong></td>
</tr>
<tr>
<td>Vargas</td>
<td>50</td>
</tr>
<tr>
<td>Hunold</td>
<td>60</td>
</tr>
<tr>
<td>Ernst</td>
<td>60</td>
</tr>
</tbody>
</table>

### DEPARTMENTS

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>DEPARTMENT_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Administration</td>
</tr>
<tr>
<td>20</td>
<td>Marketing</td>
</tr>
<tr>
<td>20</td>
<td>Marketing</td>
</tr>
<tr>
<td>50</td>
<td>Shipping</td>
</tr>
<tr>
<td>50</td>
<td>Shipping</td>
</tr>
<tr>
<td>50</td>
<td>Shipping</td>
</tr>
<tr>
<td><strong>50</strong></td>
<td><strong>Shipping</strong></td>
</tr>
<tr>
<td>50</td>
<td>Shipping</td>
</tr>
<tr>
<td>60</td>
<td>IT</td>
</tr>
<tr>
<td>60</td>
<td>IT</td>
</tr>
</tbody>
</table>
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.
Using Table Aliases

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

```sql
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;
```
Joining More than Two Tables

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.
Non-Equijoins

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

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td>24000</td>
</tr>
<tr>
<td>Kochhar</td>
<td>17000</td>
</tr>
<tr>
<td>De Haan</td>
<td>17000</td>
</tr>
<tr>
<td>Hunold</td>
<td>9000</td>
</tr>
<tr>
<td>Ernst</td>
<td>6000</td>
</tr>
<tr>
<td>Lorontz</td>
<td>4200</td>
</tr>
<tr>
<td>Murgos</td>
<td>5800</td>
</tr>
<tr>
<td>Rajs</td>
<td>3500</td>
</tr>
<tr>
<td>Davies</td>
<td>3100</td>
</tr>
<tr>
<td>Matos</td>
<td>2600</td>
</tr>
<tr>
<td>Vargas</td>
<td>2500</td>
</tr>
<tr>
<td>Zlotkey</td>
<td>10500</td>
</tr>
<tr>
<td>Abel</td>
<td>11000</td>
</tr>
<tr>
<td>Taylor</td>
<td>8600</td>
</tr>
</tbody>
</table>

20 rows selected

<table>
<thead>
<tr>
<th>GRA</th>
<th>LOWEST_SAL</th>
<th>HIGHEST_SAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1000</td>
<td>2999</td>
</tr>
<tr>
<td>B</td>
<td>3000</td>
<td>5999</td>
</tr>
<tr>
<td>C</td>
<td>6000</td>
<td>9999</td>
</tr>
<tr>
<td>D</td>
<td>10000</td>
<td>14999</td>
</tr>
<tr>
<td>E</td>
<td>15000</td>
<td>24999</td>
</tr>
<tr>
<td>F</td>
<td>25000</td>
<td>40000</td>
</tr>
</tbody>
</table>
Retrieving Records with Non-Equijoins

```sql
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;
```

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>SALARY</th>
<th>GRA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matos</td>
<td>2600</td>
<td>A</td>
</tr>
<tr>
<td>Vargas</td>
<td>2500</td>
<td>A</td>
</tr>
<tr>
<td>Lorentz</td>
<td>4200</td>
<td>B</td>
</tr>
<tr>
<td>Murgos</td>
<td>5800</td>
<td>B</td>
</tr>
<tr>
<td>Rajs</td>
<td>3500</td>
<td>B</td>
</tr>
<tr>
<td>Davies</td>
<td>3100</td>
<td>B</td>
</tr>
<tr>
<td>Whalen</td>
<td>4400</td>
<td>B</td>
</tr>
<tr>
<td>Hunold</td>
<td>9000</td>
<td>C</td>
</tr>
<tr>
<td>Ernst</td>
<td>6000</td>
<td>C</td>
</tr>
</tbody>
</table>

20 rows selected.
## Outer Joins

### DEPARTMENTS

<table>
<thead>
<tr>
<th>DEPARTMENT_NAME</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Administration</td>
<td>10</td>
</tr>
<tr>
<td>Marketing</td>
<td>20</td>
</tr>
<tr>
<td>Shipping</td>
<td>50</td>
</tr>
<tr>
<td>IT</td>
<td>60</td>
</tr>
<tr>
<td>Sales</td>
<td>80</td>
</tr>
<tr>
<td>Executive</td>
<td>90</td>
</tr>
<tr>
<td>Accounting</td>
<td>110</td>
</tr>
<tr>
<td>Contracting</td>
<td>190</td>
</tr>
</tbody>
</table>

8 rows selected.

### EMPLOYEES

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>LAST_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>90</td>
<td>King</td>
</tr>
<tr>
<td>90</td>
<td>Kochhar</td>
</tr>
<tr>
<td>90</td>
<td>De Haan</td>
</tr>
<tr>
<td>60</td>
<td>Hunold</td>
</tr>
<tr>
<td>60</td>
<td>Ernst</td>
</tr>
<tr>
<td>60</td>
<td>Lorentz</td>
</tr>
<tr>
<td>50</td>
<td>Mourgos</td>
</tr>
<tr>
<td>50</td>
<td>Rajs</td>
</tr>
<tr>
<td>50</td>
<td>Davies</td>
</tr>
<tr>
<td>50</td>
<td>Matos</td>
</tr>
<tr>
<td>50</td>
<td>Vargas</td>
</tr>
<tr>
<td>60</td>
<td>Zlotkey</td>
</tr>
</tbody>
</table>

20 rows selected.

There are no employees in department 190.
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 (+).

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

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

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>DEPARTMENT_ID</th>
<th>DEPARTMENT_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Whalen</td>
<td>10</td>
<td>Administration</td>
</tr>
<tr>
<td>Hartstein</td>
<td>20</td>
<td>Marketing</td>
</tr>
<tr>
<td>Fay</td>
<td>20</td>
<td>Marketing</td>
</tr>
<tr>
<td>Mourgos</td>
<td>50</td>
<td>Shipping</td>
</tr>
<tr>
<td>Rajs</td>
<td>50</td>
<td>Shipping</td>
</tr>
<tr>
<td>Davies</td>
<td>50</td>
<td>Shipping</td>
</tr>
<tr>
<td>Matos</td>
<td>50</td>
<td>Shipping</td>
</tr>
</tbody>
</table>

...  

<table>
<thead>
<tr>
<th>Gietz</th>
<th>110</th>
<th>Accountig</th>
</tr>
</thead>
</table>

20 rows selected.
Self Joins

**EMPLOYEES (WORKER)**

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>MANAGER_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>King</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>Kochhar</td>
<td>100</td>
</tr>
<tr>
<td>102</td>
<td>De Haan</td>
<td>100</td>
</tr>
<tr>
<td>103</td>
<td>Hunold</td>
<td>102</td>
</tr>
<tr>
<td>104</td>
<td>Ernst</td>
<td>103</td>
</tr>
<tr>
<td>107</td>
<td>Lorentz</td>
<td>103</td>
</tr>
<tr>
<td>124</td>
<td>Mourgos</td>
<td>100</td>
</tr>
</tbody>
</table>

**EMPLOYEES (MANAGER)**

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>King</td>
</tr>
<tr>
<td>101</td>
<td>Kochhar</td>
</tr>
<tr>
<td>102</td>
<td>De Haan</td>
</tr>
<tr>
<td>103</td>
<td>Hunold</td>
</tr>
<tr>
<td>104</td>
<td>Ernst</td>
</tr>
<tr>
<td>107</td>
<td>Lorentz</td>
</tr>
<tr>
<td>124</td>
<td>Mourgos</td>
</tr>
</tbody>
</table>

MANAGER_ID in the WORKER table is equal to EMPLOYEE_ID in the MANAGER table.
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.|        |                                               |
Practice 4, Part One: Overview

This practice covers writing queries to join tables together using Oracle syntax.
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)];
```
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.

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

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>DEPARTMENT_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td>Administration</td>
</tr>
<tr>
<td>Kochhar</td>
<td>Administration</td>
</tr>
<tr>
<td>De Haan</td>
<td>Administration</td>
</tr>
<tr>
<td>Hunold</td>
<td>Administration</td>
</tr>
</tbody>
</table>

...  
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.
Retrieving Records with Natural Joins

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

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>DEPARTMENT_NAME</th>
<th>LOCATION_ID</th>
<th>CITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>IT</td>
<td>1400</td>
<td>Southlake</td>
</tr>
<tr>
<td>50</td>
<td>Shipping</td>
<td>1500</td>
<td>South San Francisco</td>
</tr>
<tr>
<td>10</td>
<td>Administration</td>
<td>1700</td>
<td>Seattle</td>
</tr>
<tr>
<td>90</td>
<td>Executive</td>
<td>1700</td>
<td>Seattle</td>
</tr>
<tr>
<td>110</td>
<td>Accounting</td>
<td>1700</td>
<td>Seattle</td>
</tr>
<tr>
<td>190</td>
<td>Contracting</td>
<td>1700</td>
<td>Seattle</td>
</tr>
<tr>
<td>20</td>
<td>Marketing</td>
<td>1800</td>
<td>Toronto</td>
</tr>
<tr>
<td>80</td>
<td>Sales</td>
<td>2500</td>
<td>Oxford</td>
</tr>
</tbody>
</table>

8 rows selected.
Creating Joins with the \texttt{USING} Clause

- If several columns have the same names but the data types do not match, the \texttt{NATURAL JOIN} clause can be modified with the \texttt{USING} clause to specify the columns that should be used for an equijoin.

- Use the \texttt{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 \texttt{NATURAL JOIN} and \texttt{USING} clauses are mutually exclusive.
Retrieving Records with the **USING** Clause

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

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>LOCATION_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>Whalen</td>
<td>1700</td>
</tr>
<tr>
<td>201</td>
<td>Hartstein</td>
<td>1800</td>
</tr>
<tr>
<td>202</td>
<td>Fay</td>
<td>1800</td>
</tr>
<tr>
<td>124</td>
<td>Mourgos</td>
<td>1500</td>
</tr>
<tr>
<td>141</td>
<td>Rajs</td>
<td>1500</td>
</tr>
<tr>
<td>142</td>
<td>Davies</td>
<td>1500</td>
</tr>
<tr>
<td>143</td>
<td>Matos</td>
<td>1500</td>
</tr>
<tr>
<td>144</td>
<td>Vargas</td>
<td>1500</td>
</tr>
<tr>
<td>103</td>
<td>Hunold</td>
<td>1400</td>
</tr>
</tbody>
</table>

19 rows selected.
Creating Joins with the `ON` Clause

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

```sql
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);
```

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>DEPARTMENT_ID</th>
<th>DEPARTMENT_ID</th>
<th>LOCATION_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>Whalen</td>
<td>10</td>
<td>10</td>
<td>1700</td>
</tr>
<tr>
<td>201</td>
<td>Hartstein</td>
<td>20</td>
<td>20</td>
<td>1800</td>
</tr>
<tr>
<td>202</td>
<td>Fay</td>
<td>20</td>
<td>20</td>
<td>1800</td>
</tr>
<tr>
<td>124</td>
<td>Mourgos</td>
<td>50</td>
<td>50</td>
<td>1500</td>
</tr>
<tr>
<td>141</td>
<td>Rajs</td>
<td>50</td>
<td>50</td>
<td>1500</td>
</tr>
<tr>
<td>142</td>
<td>Davies</td>
<td>50</td>
<td>50</td>
<td>1500</td>
</tr>
<tr>
<td>143</td>
<td>Matios</td>
<td>50</td>
<td>50</td>
<td>1500</td>
</tr>
</tbody>
</table>

19 rows selected.
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;
```

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>CITY</th>
<th>DEPARTMENT_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>103</td>
<td>Southlake</td>
<td>IT</td>
</tr>
<tr>
<td>104</td>
<td>Southlake</td>
<td>IT</td>
</tr>
<tr>
<td>107</td>
<td>Southlake</td>
<td>IT</td>
</tr>
<tr>
<td>124</td>
<td>South San Francisco</td>
<td>Shipping</td>
</tr>
<tr>
<td>141</td>
<td>South San Francisco</td>
<td>Shipping</td>
</tr>
<tr>
<td>142</td>
<td>South San Francisco</td>
<td>Shipping</td>
</tr>
<tr>
<td>143</td>
<td>South San Francisco</td>
<td>Shipping</td>
</tr>
<tr>
<td>144</td>
<td>South San Francisco</td>
<td>Shipping</td>
</tr>
</tbody>
</table>

19 rows selected.
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.
LEFT OUTER JOIN

```sql
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);
```

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>DEPARTMENT_ID</th>
<th>DEPARTMENT_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Whalen</td>
<td>10</td>
<td>Administration</td>
</tr>
<tr>
<td>Fay</td>
<td>20</td>
<td>Marketing</td>
</tr>
<tr>
<td>Hartstein</td>
<td>20</td>
<td>Marketing</td>
</tr>
<tr>
<td>De Haan</td>
<td>90</td>
<td>Executive</td>
</tr>
<tr>
<td>Kochhar</td>
<td>90</td>
<td>Executive</td>
</tr>
<tr>
<td>King</td>
<td>90</td>
<td>Executive</td>
</tr>
<tr>
<td>Gietz</td>
<td>110</td>
<td>Accounting</td>
</tr>
<tr>
<td>Higgins</td>
<td>110</td>
<td>Accounting</td>
</tr>
<tr>
<td>Grant</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

20 rows selected.
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);

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>DEPARTMENT_ID</th>
<th>DEPARTMENT_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td></td>
<td>90 Executive</td>
</tr>
<tr>
<td>Kochhar</td>
<td></td>
<td>90 Executive</td>
</tr>
<tr>
<td>Whalen</td>
<td>10</td>
<td>Administration</td>
</tr>
<tr>
<td>Hartstein</td>
<td>20</td>
<td>Marketing</td>
</tr>
<tr>
<td>Fay</td>
<td>20</td>
<td>Marketing</td>
</tr>
<tr>
<td>Higgins</td>
<td>110</td>
<td>Accounting</td>
</tr>
<tr>
<td>Gietz</td>
<td>110</td>
<td>Accounting</td>
</tr>
</tbody>
</table>

20 rows selected.
### FULL OUTER JOIN

```sql
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) ;
```

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>DEPARTMENT_ID</th>
<th>DEPARTMENT_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Whalen</td>
<td>10</td>
<td>Administration</td>
</tr>
<tr>
<td>Fay</td>
<td>20</td>
<td>Marketing</td>
</tr>
<tr>
<td>De Haan</td>
<td>90</td>
<td>Executive</td>
</tr>
<tr>
<td>Kochhar</td>
<td>90</td>
<td>Executive</td>
</tr>
<tr>
<td>King</td>
<td>90</td>
<td>Executive</td>
</tr>
<tr>
<td>Gietz</td>
<td>110</td>
<td>Accounting</td>
</tr>
<tr>
<td>Higgins</td>
<td>110</td>
<td>Accounting</td>
</tr>
<tr>
<td>Grant</td>
<td></td>
<td>Contracting</td>
</tr>
</tbody>
</table>

21 rows selected.
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 ;
```

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>DEPARTMENT_ID</th>
<th>DEPARTMENT_ID</th>
<th>LOCATION_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>174</td>
<td>Abel</td>
<td></td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>176</td>
<td>Taylor</td>
<td></td>
<td>80</td>
<td>80</td>
</tr>
</tbody>
</table>
Summary

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

- Oracle proprietary syntax for versions 8i and earlier
- SQL: 1999 compliant syntax for version 9i
Practice 4, Part Two: Overview

This practice covers the following topics:

• Joining tables using an equijoin
• Performing outer and self joins
• Adding conditions
Aggregating Data Using Group Functions
Objectives

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

• Identify the available group functions
• Describe the use of group functions
• Group data using the GROUP BY clause
• Include or exclude grouped rows by using the HAVING clause
What Are Group Functions?

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

**EMPLOYEES**

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>90</td>
<td>24000</td>
</tr>
<tr>
<td>90</td>
<td>17000</td>
</tr>
<tr>
<td>90</td>
<td>17000</td>
</tr>
<tr>
<td>60</td>
<td>6000</td>
</tr>
<tr>
<td>60</td>
<td>4200</td>
</tr>
<tr>
<td>50</td>
<td>5800</td>
</tr>
<tr>
<td>50</td>
<td>3500</td>
</tr>
<tr>
<td>50</td>
<td>3100</td>
</tr>
<tr>
<td>50</td>
<td>2600</td>
</tr>
<tr>
<td>50</td>
<td>2500</td>
</tr>
<tr>
<td>80</td>
<td>10500</td>
</tr>
<tr>
<td>80</td>
<td>11000</td>
</tr>
<tr>
<td>80</td>
<td>8600</td>
</tr>
<tr>
<td>10</td>
<td>4400</td>
</tr>
</tbody>
</table>

The maximum salary in the EMPLOYEES table.

20 rows selected.
Types of Group Functions

- AVG
- COUNT
- MAX
- MIN
- STDDEV
- SUM
- VARIANCE
Group Functions Syntax

```
SELECT [column,] group_function(column), ...
FROM table
[WHERE condition]
[GROUP BY column]
[ORDER BY column];
```
Using the \texttt{AVG} and \texttt{SUM} Functions

You can use \texttt{AVG} and \texttt{SUM} for numeric data.

\begin{verbatim}
SELECT AVG(salary), MAX(salary),
      MIN(salary), SUM(salary)
FROM employees
WHERE job_id LIKE '%REP%';
\end{verbatim}

<table>
<thead>
<tr>
<th>AVG(SALARY)</th>
<th>MAX(SALARY)</th>
<th>MIN(SALARY)</th>
<th>SUM(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8150</td>
<td>11000</td>
<td>6000</td>
<td>32600</td>
</tr>
</tbody>
</table>
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;
```

<table>
<thead>
<tr>
<th>MIN(hire_date)</th>
<th>MAX(hire_date)</th>
</tr>
</thead>
<tbody>
<tr>
<td>17-JUN-87</td>
<td>29-JAN-00</td>
</tr>
</tbody>
</table>
COUNT (*) returns the number of rows in a table.

```
SELECT COUNT(*)
FROM employees
WHERE department_id = 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.

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

<table>
<thead>
<tr>
<th>COUNT(COMMISSION_PCT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
</tr>
</tbody>
</table>
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;
```

<table>
<thead>
<tr>
<th>COUNT(DISTINCT DEPARTMENT_ID)</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
</tr>
</tbody>
</table>
Group Functions and Null Values

Group functions ignore null values in the column.

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

<table>
<thead>
<tr>
<th>AVG(COMMISSION_PCT)</th>
<th>0.2125</th>
</tr>
</thead>
</table>
Using the `NVL` Function
with Group Functions

The `NVL` function forces group functions to include null values.

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

| `AVG(NVL(COMMISSION_PCT,0))` | .0425 |
Creating Groups of Data

EMPLOYEES

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>4400</td>
</tr>
<tr>
<td>20</td>
<td>13000</td>
</tr>
<tr>
<td>20</td>
<td>6000</td>
</tr>
<tr>
<td>50</td>
<td>5800</td>
</tr>
<tr>
<td>50</td>
<td>3500</td>
</tr>
<tr>
<td>50</td>
<td>3100</td>
</tr>
<tr>
<td>50</td>
<td>2500</td>
</tr>
<tr>
<td>50</td>
<td>2600</td>
</tr>
<tr>
<td>60</td>
<td>9000</td>
</tr>
<tr>
<td>60</td>
<td>6000</td>
</tr>
<tr>
<td>60</td>
<td>4200</td>
</tr>
<tr>
<td>80</td>
<td>10500</td>
</tr>
<tr>
<td>80</td>
<td>8600</td>
</tr>
<tr>
<td>80</td>
<td>11000</td>
</tr>
<tr>
<td>90</td>
<td>24000</td>
</tr>
<tr>
<td>90</td>
<td>17000</td>
</tr>
</tbody>
</table>

... 20 rows selected.

The average salary in the EMPLOYEES table for each department.

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>AVG(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>4400</td>
</tr>
<tr>
<td>20</td>
<td>9500</td>
</tr>
<tr>
<td>50</td>
<td>3500</td>
</tr>
<tr>
<td>60</td>
<td>6400</td>
</tr>
<tr>
<td>80</td>
<td>10033.3333</td>
</tr>
<tr>
<td>90</td>
<td>19333.3333</td>
</tr>
<tr>
<td>110</td>
<td>10150</td>
</tr>
</tbody>
</table>

Oracle
Creating Groups of Data: The GROUP BY Clause Syntax

```sql
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.
**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 ;
```

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>AVG(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>4400</td>
</tr>
<tr>
<td>20</td>
<td>9500</td>
</tr>
<tr>
<td>50</td>
<td>3500</td>
</tr>
<tr>
<td>60</td>
<td>6400</td>
</tr>
<tr>
<td>80</td>
<td>10033.3333</td>
</tr>
<tr>
<td>90</td>
<td>19333.3333</td>
</tr>
<tr>
<td>110</td>
<td>10150</td>
</tr>
</tbody>
</table>

8 rows selected.
Using the **GROUP BY** Clause

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

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

<table>
<thead>
<tr>
<th>AVG(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4400</td>
</tr>
<tr>
<td>5500</td>
</tr>
<tr>
<td>3500</td>
</tr>
<tr>
<td>6400</td>
</tr>
<tr>
<td>10333.3333</td>
</tr>
<tr>
<td>19333.3333</td>
</tr>
<tr>
<td>10150</td>
</tr>
<tr>
<td>7000</td>
</tr>
</tbody>
</table>
Grouping by More Than One Column

**EMPLOYEES**

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>JOB_ID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>90</td>
<td>AD_PRES</td>
<td>24000</td>
</tr>
<tr>
<td>90</td>
<td>AD_VP</td>
<td>17000</td>
</tr>
<tr>
<td>80</td>
<td>IT_PROG</td>
<td>9000</td>
</tr>
<tr>
<td>80</td>
<td>IT_PROG</td>
<td>6000</td>
</tr>
<tr>
<td>80</td>
<td>IT_PROG</td>
<td>4200</td>
</tr>
<tr>
<td>50</td>
<td>ST_MAN</td>
<td>5800</td>
</tr>
<tr>
<td>50</td>
<td>ST_CLERK</td>
<td>3500</td>
</tr>
<tr>
<td>50</td>
<td>ST_CLERK</td>
<td>3100</td>
</tr>
<tr>
<td>50</td>
<td>ST_CLERK</td>
<td>2600</td>
</tr>
<tr>
<td>50</td>
<td>ST_CLERK</td>
<td>2500</td>
</tr>
<tr>
<td>80</td>
<td>SA_MAN</td>
<td>10500</td>
</tr>
<tr>
<td>80</td>
<td>SA_REP</td>
<td>11000</td>
</tr>
<tr>
<td>80</td>
<td>SA_REP</td>
<td>8600</td>
</tr>
<tr>
<td>20</td>
<td>MK_REP</td>
<td>6000</td>
</tr>
<tr>
<td>110</td>
<td>AC_MGR</td>
<td>12000</td>
</tr>
<tr>
<td>110</td>
<td>AC_ACCOUNT</td>
<td>8300</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>JOB_ID</th>
<th>SUM(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>AD_ASST</td>
<td>4400</td>
</tr>
<tr>
<td>20</td>
<td>MK_MAN</td>
<td>13000</td>
</tr>
<tr>
<td>20</td>
<td>MK_REP</td>
<td>6000</td>
</tr>
<tr>
<td>50</td>
<td>ST_CLERK</td>
<td>11700</td>
</tr>
<tr>
<td>50</td>
<td>ST_MAN</td>
<td>5800</td>
</tr>
<tr>
<td>60</td>
<td>IT_PROG</td>
<td>13200</td>
</tr>
<tr>
<td>80</td>
<td>SA_MAN</td>
<td>10500</td>
</tr>
<tr>
<td>80</td>
<td>SA_REP</td>
<td>19600</td>
</tr>
<tr>
<td>90</td>
<td>AD_FRES</td>
<td>24000</td>
</tr>
<tr>
<td>90</td>
<td>AD_VP</td>
<td>34000</td>
</tr>
<tr>
<td>110</td>
<td>AC_ACCOUNT</td>
<td>8300</td>
</tr>
<tr>
<td>110</td>
<td>AC_MGR</td>
<td>12000</td>
</tr>
<tr>
<td></td>
<td>SA_REP</td>
<td>7000</td>
</tr>
</tbody>
</table>

13 rows selected.

20 rows selected.
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;
```

<table>
<thead>
<tr>
<th>DEPT_ID</th>
<th>JOB_ID</th>
<th>SUM(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>AD_ASST</td>
<td>4400</td>
</tr>
<tr>
<td>20</td>
<td>MK_MAN</td>
<td>13000</td>
</tr>
<tr>
<td>20</td>
<td>MK_REP</td>
<td>6000</td>
</tr>
<tr>
<td>50</td>
<td>ST_CLERK</td>
<td>11700</td>
</tr>
<tr>
<td>50</td>
<td>ST_MAN</td>
<td>5800</td>
</tr>
<tr>
<td>60</td>
<td>IT_PROG</td>
<td>19200</td>
</tr>
<tr>
<td>60</td>
<td>SA_MAN</td>
<td>10500</td>
</tr>
<tr>
<td>60</td>
<td>SA_REP</td>
<td>19600</td>
</tr>
<tr>
<td>90</td>
<td>AD_PRES</td>
<td>24000</td>
</tr>
<tr>
<td>90</td>
<td>AD_VP</td>
<td>34000</td>
</tr>
<tr>
<td>110</td>
<td>AC_ACCOUNT</td>
<td>8300</td>
</tr>
<tr>
<td>110</td>
<td>AC_MGR</td>
<td>12000</td>
</tr>
<tr>
<td></td>
<td>SA_REP</td>
<td>7000</td>
</tr>
</tbody>
</table>

13 rows selected.
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**
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.

```sql
SELECT department_id, AVG(salary)
FROM employees
WHERE AVG(salary) > 8000
GROUP BY department_id;
```

```sql
WHERE AVG(salary) > 8000
*
ERROR at line 3:
ORA-00934: group function is not allowed here
```

Cannot use the `WHERE` clause to restrict groups
Excluding Group Results

**EMPLOYEES**

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>90</td>
<td>24000</td>
</tr>
<tr>
<td>90</td>
<td>17000</td>
</tr>
<tr>
<td>90</td>
<td>17000</td>
</tr>
<tr>
<td>60</td>
<td>9000</td>
</tr>
<tr>
<td>60</td>
<td>6000</td>
</tr>
<tr>
<td>60</td>
<td>4200</td>
</tr>
<tr>
<td>50</td>
<td>5800</td>
</tr>
<tr>
<td>50</td>
<td>3500</td>
</tr>
<tr>
<td>50</td>
<td>3100</td>
</tr>
<tr>
<td>50</td>
<td>2600</td>
</tr>
<tr>
<td>50</td>
<td>2500</td>
</tr>
<tr>
<td>80</td>
<td>10500</td>
</tr>
<tr>
<td>80</td>
<td>11000</td>
</tr>
<tr>
<td>80</td>
<td>8600</td>
</tr>
</tbody>
</table>

...  

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>MAX(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>13000</td>
</tr>
<tr>
<td>80</td>
<td>11000</td>
</tr>
<tr>
<td>90</td>
<td>24000</td>
</tr>
<tr>
<td>110</td>
<td>12000</td>
</tr>
<tr>
<td>110</td>
<td>8300</td>
</tr>
</tbody>
</table>

20 rows selected.
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];
```
Using the **HAVING** Clause

```sql
SELECT department_id, MAX(salary)
FROM employees
GROUP BY department_id
HAVING MAX(salary) > 10000;
```

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>MAX(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>13000</td>
</tr>
<tr>
<td>80</td>
<td>11000</td>
</tr>
<tr>
<td>90</td>
<td>24000</td>
</tr>
<tr>
<td>110</td>
<td>12000</td>
</tr>
</tbody>
</table>
Using the `HAVING` Clause

```sql
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);
```

<table>
<thead>
<tr>
<th>JOB_ID</th>
<th>PAYROLL</th>
</tr>
</thead>
<tbody>
<tr>
<td>IT_FROG</td>
<td></td>
</tr>
<tr>
<td>AD_PRES</td>
<td></td>
</tr>
<tr>
<td>AD_VP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>19200</td>
</tr>
<tr>
<td></td>
<td>24000</td>
</tr>
<tr>
<td></td>
<td>34000</td>
</tr>
</tbody>
</table>
Nesting Group Functions

Display the maximum average salary.

```
SELECT MAX(AVG(salary))
FROM employees
GROUP BY department_id;
```
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];
```
Practice 5 Overview

This practice covers the following topics:

• Writing queries that use the group functions
• Grouping by rows to achieve more than one result
• Excluding groups by using the `HAVING` clause
Subqueries
Objectives

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

• Describe the types of problem that subqueries can solve
• Define subqueries
• List the types of subqueries
• Write single-row and multiple-row subqueries
Using a Subquery to Solve a Problem

Who has a salary greater than Abel’s?

Main Query:

Which employees have salaries greater than Abel’s salary?

Subquery

What is Abel’s salary?
Subquery Syntax

- The subquery (inner query) executes once before the main query.
- The result of the subquery is used by the main query (outer query).

```
SELECT select_list
FROM table
WHERE expr operator
(SELECT select_list
FROM table);
```
Using a Subquery

```
SELECT last_name
FROM employees
WHERE salary >
    (SELECT salary
     FROM employees
     WHERE last_name = 'Abel');
```

<table>
<thead>
<tr>
<th>LAST_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
</tr>
<tr>
<td>Kochhar</td>
</tr>
<tr>
<td>De Haan</td>
</tr>
<tr>
<td>Hartstein</td>
</tr>
<tr>
<td>Higgins</td>
</tr>
</tbody>
</table>
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.
Types of Subqueries

• Single-row subquery

Main query
Subquery
returns
ST_CLERK

• Multiple-row subquery

Main query
Subquery
returns
ST_CLERK
SA_MAN
Single-Row Subqueries

- Return only one row
- Use single-row comparison operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Equal to</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Not equal to</td>
</tr>
</tbody>
</table>
Executing Single-Row Subqueries

```
SELECT last_name, job_id, salary
FROM employees
WHERE job_id = (SELECT job_id
                FROM employees
                WHERE employee_id = 141)
AND salary > (SELECT salary
               FROM employees
               WHERE employee_id = 143);
```

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>JOB_ID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rajs</td>
<td>ST_CLERK</td>
<td>3500</td>
</tr>
<tr>
<td>Davias</td>
<td>ST_CLERK</td>
<td>3100</td>
</tr>
</tbody>
</table>
Using Group Functions in a Subquery

```sql
SELECT last_name, job_id, salary
FROM employees
WHERE salary = (SELECT MIN(salary)
                FROM employees);
```

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>JOB_ID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Margas</td>
<td>ST_CLERK</td>
<td>2500</td>
</tr>
</tbody>
</table>
The **HAVING** Clause with Subqueries

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

```sql
SELECT department_id, MIN(salary)
FROM employees
GROUP BY department_id
HAVING MIN(salary) > (SELECT MIN(salary)
                         FROM employees
                         WHERE department_id = 50);
```
What is Wrong with this Statement?

SELECT employee_id, last_name
FROM employees
WHERE salary =
    (SELECT MIN(salary)
     FROM employees
     GROUP BY department_id);

ERROR at line 4:
ORA-01427: single-row subquery returns more than one row

Single-row operator with multiple-row subquery
Will this Statement Return Rows?

```sql
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
Multiple-Row Subqueries

• Return more than one row
• Use multiple-row comparison operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IN</strong></td>
<td>Equal to any member in the list</td>
</tr>
<tr>
<td><strong>ANY</strong></td>
<td>Compare value to each value returned by the subquery</td>
</tr>
<tr>
<td><strong>ALL</strong></td>
<td>Compare value to every value returned by the subquery</td>
</tr>
</tbody>
</table>
Using the **ANY** Operator in Multiple-Row Subqueries

```sql
SELECT employee_id, last_name, job_id, salary
FROM employees
WHERE salary < ANY
    (SELECT salary
     FROM employees
     WHERE job_id = 'IT_PROG')
AND job_id <> 'IT_PROG';
```

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>JOB_ID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>124</td>
<td>Mourgès</td>
<td>ST_MAN</td>
<td>6800</td>
</tr>
<tr>
<td>141</td>
<td>Rajs</td>
<td>ST_CLERK</td>
<td>3500</td>
</tr>
<tr>
<td>142</td>
<td>Davies</td>
<td>ST_CLERK</td>
<td>3100</td>
</tr>
<tr>
<td>143</td>
<td>Matos</td>
<td>ST_CLERK</td>
<td>2600</td>
</tr>
<tr>
<td>144</td>
<td>Vargas</td>
<td>ST_CLERK</td>
<td>2500</td>
</tr>
</tbody>
</table>

10 rows selected.
Using the **ALL** Operator in Multiple-Row Subqueries

```sql
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';
```
Null Values in a Subquery

```sql
SELECT emp.last_name
FROM   employees emp
WHERE  emp.employee_id NOT IN
       (SELECT mgr.manager_id
        FROM   employees mgr);

no rows selected
```
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);
```
Practice 6 Overview

This practice covers the following topics:

- Creating subqueries to query values based on unknown criteria
- Using subqueries to find out which values exist in one set of data and not in another
Producing Readable Output with iSQL*Plus
Objectives

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

- Produce queries that require a substitution variable
- Customize the iSQL*Plus environment
- Produce more readable output
- Create and execute script files
Substitution Variables

User

I want to query different values.

... salary = ? ...
... department_id = ? ...
... last_name = ? ...
Substitution Variables

Use iSQL*Plus substitution variables to:

• Temporarily store values
  – Single ampersand (&)
  – Double ampersand (&&)
  – DEFINE command
• Pass variable values between SQL statements
• Dynamically alter headers and footers
Using the & Substitution Variable

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

```
SELECT employee_id, last_name, salary, department_id
FROM employees
WHERE employee_id = &employee_num;
```
Using the & Substitution Variable

old 3: WHERE employee_id = &employee_num
new 3: WHERE employee_id = 101

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>SALARY</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Kochhar</td>
<td>17000</td>
<td>90</td>
</tr>
</tbody>
</table>
Character and Date Values with Substitution Variables

Use single quotation marks for date and character values.

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

Define Substitution Variables

```
"job_title" = "IT_PROG"
```

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>DEPARTMENT_ID</th>
<th>SALARY*12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hunold</td>
<td>60</td>
<td>108000</td>
</tr>
<tr>
<td>Ernst</td>
<td>60</td>
<td>72000</td>
</tr>
<tr>
<td>Lorentz</td>
<td>60</td>
<td>50400</td>
</tr>
</tbody>
</table>
Specifying Column Names, Expressions, and Text

Use substitution variables to supplement the following:

• \texttt{WHERE} conditions
• \texttt{ORDER BY} clauses
• Column expressions
• Table names
• Entire \texttt{SELECT} statements
SELECT employee_id, last_name, job_id, &column_name
FROM employees
WHERE &condition
ORDER BY &order_column ;

Define Substitution Variables

| "column_name" | salary |
| "condition"  | salary > 15000 |
| "order_column" | last_name |

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>JOB_ID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>102</td>
<td>De Haan</td>
<td>AD_VP</td>
<td>17000</td>
</tr>
<tr>
<td>100</td>
<td>King</td>
<td>AD_PRES</td>
<td>24000</td>
</tr>
<tr>
<td>101</td>
<td>Kochhar</td>
<td>AD_VP</td>
<td>17000</td>
</tr>
</tbody>
</table>
Defining Substitution Variables

- You can predefine variables using the iSQL*Plus DEFINE command.
  \[ \text{DEFINE } \text{variable} = \text{value} \] creates a user variable with the CHAR data type.
- If you need to predefine a variable that includes spaces, you must enclose the value within single quotation marks when using the DEFINE command.
- A defined variable is available for the session
DEFINE and UNDEFINE Commands

- A variable remains defined until you either:
  - Use the UNDEFINE command to clear it
  - Exit iSQL*Plus
- You can verify your changes with the DEFINE command.

```
DEFINE job_title = IT_PROG
DEFINE job_title
DEFINE JOB_TITLE       = "IT_PROG" (CHAR)

UNDEFINE job_title
DEFINE job_title
UNDEFINE job_title
DEFINIE job_title
SP2-0135: symbol job_title is UNDEFINED
```
Using the **DEFINE** Command with & Substitution Variable

- Create the substitution variable using the **DEFINE** command.

```sql
DEFINE employee_num = 200
```

- Use a variable prefixed with an ampersand (&) to substitute the value in the SQL statement.

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

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>SALARY</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>Whalen</td>
<td>4400</td>
<td>10</td>
</tr>
</tbody>
</table>
Using the && Substitution Variable

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

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

Define Substitution Variables

```
"column_name" department_id
```

Submit for Execution  Cancel

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>JOB_ID</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>Whalen</td>
<td>AD_ASST</td>
<td>10</td>
</tr>
<tr>
<td>201</td>
<td>Hartstein</td>
<td>MK_MAN</td>
<td>20</td>
</tr>
</tbody>
</table>

... 20 rows selected.
Using the `VERIFY` Command

Use the `VERIFY` command to toggle the display of the substitution variable, before and after `iSQL*Plus` replaces substitution variables with values.

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

"employee_num" [200]

```
old  3: WHERE  employee_id = &employee_num
new  3: WHERE  employee_id = 200
```
Customizing the SQL*Plus Environment

- Use `SET` commands to control current session.

  ```
  SET system_variable value
  ```

- Verify what you have set by using the `SHOW` command.

  ```
  SET ECHO ON
  SHOW ECHO
  echo ON
  ```
SET Command Variables

- ARRAYSIZE {20 | n}
- FEEDBACK {6 | n | OFF | ON}
- HEADING {OFF | ON}
- LONG {80 | n | ON | text}

SET HEADING OFF

SHOW HEADING

HEADING OFF
iSQL*Plus Format Commands

- COLUMN [column option]
- TTITLE [text | OFF | ON]
- BTITLE [text | OFF | ON]
- BREAK [ON report_element]
The COLUMN Command

Controls display of a column:

```
COL[UMN] [{column|alias} [option]]
```

- **CLE[AR]**: Clears any column formats
- **HEA[ADING] text**: Sets the column heading
- **FOR[MAT] format**: Changes the display of the column using a format model
- **NOPRINT | PRINT**
- **NULL**
Using the COLUMN Command

• **Create column headings.**

```
COLUMN last_name HEADING 'Employee|Name'
COLUMN salary JUSTIFY LEFT FORMAT $99,990.00
COLUMN manager FORMAT 999999999 NULL 'No manager'
```

• **Display the current setting for the LAST_NAME column.**

```
COLUMN last_name
```

• **Clear settings for the LAST_NAME column.**

```
COLUMN last_name CLEAR
```
## COLUMN Format Models

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Single zero-suppression digit</td>
<td>999999</td>
<td>1234</td>
</tr>
<tr>
<td>0</td>
<td>Enforces leading zero</td>
<td>099999</td>
<td>001234</td>
</tr>
<tr>
<td>$</td>
<td>Floating dollar sign</td>
<td>$9999</td>
<td>$1234</td>
</tr>
<tr>
<td>L</td>
<td>Local currency</td>
<td>L9999</td>
<td>L1234</td>
</tr>
<tr>
<td>.</td>
<td>Position of decimal point</td>
<td>9999.99</td>
<td>1234.00</td>
</tr>
<tr>
<td>,</td>
<td>Thousand separator</td>
<td>9,999</td>
<td>1,234</td>
</tr>
</tbody>
</table>
Using the **BREAK** Command

Use the **BREAK** command to suppress duplicates.

```
BREAK ON job_id
```
Using the **TTITLE** and **BTITLE** Commands

- Display headers and footers.
  
  \[ t \text{at} | \text{OFF}|\text{ON} \]

- Set the report header.

  **TTITLE** 'Salary|Report'

- Set the report footer.

  **BTITLE** 'Confidential'
Using the **TTITLE** and **BTITLE** Commands

- **Display headers and footers.**

  \[
  \text{TTL[TLE]} \ [\text{text} | \text{OFF} | \text{ON}]
  \]

- **Set the report header.**

  \[
  \text{TTITLE} \ 'Salary|Report'
  \]

- **Set the report footer.**

  \[
  \text{BTITLE} \ 'Confidential'
  \]
Creating a Script File to Run a Report

1. Create and test the SQL `SELECT` statement.
2. Save the `SELECT` statement into a script file.
3. Load the script file into an editor.
4. Add formatting commands before the `SELECT` statement.
5. Verify that the termination character follows the `SELECT` statement.
Creating a Script File to Run a Report

6. Clear formatting commands after the `SELECT` statement.
7. Save the script file.
8. Load the script file into the `iSQL*Plus` text window, and click the Execute button.
## Sample Report

### Fri Sep 28

<table>
<thead>
<tr>
<th>Job Category</th>
<th>Employee</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC_ACCOUNT</td>
<td>Gietz</td>
<td>$8,300.00</td>
</tr>
<tr>
<td>AC_MGR</td>
<td>Higgins</td>
<td>$12,000.00</td>
</tr>
<tr>
<td>AD_ASST</td>
<td>Whalen</td>
<td>$4,400.00</td>
</tr>
<tr>
<td>IT_PROG</td>
<td>Ernst</td>
<td>$6,000.00</td>
</tr>
<tr>
<td></td>
<td>Hunold</td>
<td>$9,000.00</td>
</tr>
<tr>
<td></td>
<td>Lorentz</td>
<td>$4,200.00</td>
</tr>
<tr>
<td>MK_MAN</td>
<td>Hartstein</td>
<td>$13,000.00</td>
</tr>
<tr>
<td>MK_REP</td>
<td>Fay</td>
<td>$6,000.00</td>
</tr>
<tr>
<td>SA_MAN</td>
<td>Zlotkey</td>
<td>$10,500.00</td>
</tr>
<tr>
<td>SA_REP</td>
<td>Abel</td>
<td>$11,000.00</td>
</tr>
<tr>
<td></td>
<td>Grant</td>
<td>$7,000.00</td>
</tr>
<tr>
<td></td>
<td>Taylor</td>
<td>$8,600.00</td>
</tr>
</tbody>
</table>
Sample Report

<table>
<thead>
<tr>
<th>Job Category</th>
<th>Employee</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC_ACCOUNT</td>
<td>Gietz</td>
<td>$8,300.00</td>
</tr>
<tr>
<td>AC_MGR</td>
<td>Higgins</td>
<td>$12,000.00</td>
</tr>
<tr>
<td>AD_ASST</td>
<td>Whalen</td>
<td>$4,400.00</td>
</tr>
<tr>
<td>IT_PROG</td>
<td>Ernst</td>
<td>$6,000.00</td>
</tr>
<tr>
<td></td>
<td>Hunold</td>
<td>$9,000.00</td>
</tr>
<tr>
<td></td>
<td>Lorentz</td>
<td>$4,200.00</td>
</tr>
<tr>
<td>MK_MAN</td>
<td>Hartstein</td>
<td>$13,000.00</td>
</tr>
<tr>
<td>MK_REP</td>
<td>Fay</td>
<td>$6,000.00</td>
</tr>
<tr>
<td>SA_MAN</td>
<td>Zlotkey</td>
<td>$10,500.00</td>
</tr>
<tr>
<td>SA_REP</td>
<td>Abel</td>
<td>$11,000.00</td>
</tr>
<tr>
<td></td>
<td>Grant</td>
<td>$7,000.00</td>
</tr>
<tr>
<td></td>
<td>Taylor</td>
<td>$8,600.00</td>
</tr>
</tbody>
</table>

Confidential
Summary

In this lesson, you should have learned how to:

• Use $SQL*Plus substitution variables to store values temporarily
• Use SET commands to control the current COLUMN command to control the display of

  COLUMN command to control the display of

• Use the BREAK command to suppress duplicates and divide rows into sections
• Use the TTITLE and BTITLE commands to display headers and footers
Practice 7 Overview

This practice covers the following topics:

- Creating a query to display values using substitution variables
- Starting a command file containing variables
Manipulating Data
Objectives

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

• Describe each DML statement
• Insert rows into a table
• Update rows in a table
• Delete rows from a table
• Merge rows in a table
• Control transactions
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.
Adding a New Row to a Table

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>DEPARTMENT_NAME</th>
<th>MANAGER_ID</th>
<th>LOCATION_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Administration</td>
<td>200</td>
<td>1700</td>
</tr>
<tr>
<td>20</td>
<td>Marketing</td>
<td>201</td>
<td>1800</td>
</tr>
<tr>
<td>50</td>
<td>Shipping</td>
<td>124</td>
<td>1500</td>
</tr>
<tr>
<td>60</td>
<td>IT</td>
<td>103</td>
<td>1400</td>
</tr>
<tr>
<td>80</td>
<td>Sales</td>
<td>149</td>
<td>2500</td>
</tr>
<tr>
<td>90</td>
<td>Executive</td>
<td>100</td>
<td>1700</td>
</tr>
<tr>
<td>110</td>
<td>Accounting</td>
<td>205</td>
<td>1700</td>
</tr>
<tr>
<td>190</td>
<td>Contracting</td>
<td></td>
<td>1700</td>
</tr>
</tbody>
</table>

...insert a new row into the DEPARTMENTS table...

New row

70 Public Relations 100 1700
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.
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.
- Enclose character and date values within single quotation marks.

```sql
INSERT INTO departments(department_id, department_name, manager_id, location_id)
VALUES (70, 'Public Relations', 100, 1700);
1 row created.
```
Inserting Rows with Null Values

• Implicit method: Omit the column from the column list.

```sql
INSERT INTO departments (department_id, department_name)
VALUES (30, 'Purchasing');
1 row created.
```

• Explicit method: Specify the NULL keyword in the VALUES clause.

```sql
INSERT INTO departments
VALUES (100, 'Finance', NULL, NULL);
1 row created.
```
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.
```
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.

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>EMAIL</th>
<th>PHONE_NUMBER</th>
<th>HIRE_DATE</th>
<th>JOB_ID</th>
<th>SALARY</th>
<th>COMMISSION_P</th>
</tr>
</thead>
<tbody>
<tr>
<td>114</td>
<td>Den</td>
<td>Raphealy</td>
<td>DRAFHEAL</td>
<td>515.127.4561</td>
<td>03-FEB-99</td>
<td>AC_ACCOUNT</td>
<td>11000</td>
<td></td>
</tr>
</tbody>
</table>
Creating a Script

- Use & substitution in a SQL statement to prompt for values.
- & is a placeholder for the variable value.

```
INSERT INTO departments
(department_id, department_name, location_id)
VALUES
(&department_id, '&department_name', &location);
```

Define Substitution Variables

```
"department_id"  40
"department_name"  Human Resources
"location"  2500
```

1 row created.
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.
# Changing Data in a Table

### EMPLOYEES

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>EMAIL</th>
<th>HIRE_DATE</th>
<th>JOB_ID</th>
<th>SALARY</th>
<th>DEPARTMENT_ID</th>
<th>COMMISSION_P</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Steven</td>
<td>King</td>
<td>SKING</td>
<td>17-JUN-87</td>
<td>AD_PRES</td>
<td>24000</td>
<td>90</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>Neena</td>
<td>Kochhar</td>
<td>NKOCHHAR</td>
<td>21-SEP-89</td>
<td>AD_VP</td>
<td>17000</td>
<td>90</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td>Lex</td>
<td>De Haan</td>
<td>LDEHAAN</td>
<td>13-JAN-93</td>
<td>AD_VP</td>
<td>17000</td>
<td>90</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td>Alexander</td>
<td>Hunold</td>
<td>AHUNOLD</td>
<td>03-JAN-90</td>
<td>IT_PROG</td>
<td>9000</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td>Bruce</td>
<td>Ernst</td>
<td>BERNST</td>
<td>21-MAY-91</td>
<td>IT_PROG</td>
<td>6000</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>107</td>
<td>Diana</td>
<td>Lorentz</td>
<td>DLORENTZ</td>
<td>07-FEB-99</td>
<td>IT_PROG</td>
<td>4200</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>124</td>
<td>Kevin</td>
<td>Mourgos</td>
<td>KMOURGOS</td>
<td>16-NOV-99</td>
<td>ST_MAN</td>
<td>5800</td>
<td>50</td>
<td></td>
</tr>
</tbody>
</table>

Update rows in the **EMPLOYEES** table.
The `UPDATE` Statement Syntax

- **Modify existing rows with the `UPDATE` statement.**

  ```sql
  UPDATE table
  SET column = value [, column = value, ...]
  [WHERE condition];
  ```

- **Update more than one row at a time, if required.**
Updating Rows in a Table

• Specific row or rows are modified if you specify the WHERE clause.

```sql
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.

```sql
UPDATE copy_emp
SET    department_id = 110;
22 rows updated.
```
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.
```
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: Integrity Constraint Error

```sql
UPDATE employees
SET    department_id = 55
WHERE  department_id = 110;

ERROR at line 1:
ORA-02291: integrity constraint (HR.EMP_DEPT_FK) violated - parent key not found
```

Department number 55 does not exist
Removing a Row from a Table

DEPARTMENTS

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>DEPARTMENT_NAME</th>
<th>MANAGER_ID</th>
<th>LOCATION_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Administration</td>
<td>200</td>
<td>1700</td>
</tr>
<tr>
<td>20</td>
<td>Marketing</td>
<td>201</td>
<td>1800</td>
</tr>
<tr>
<td>30</td>
<td>Purchasing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>Finance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>Shipping</td>
<td>124</td>
<td>1500</td>
</tr>
<tr>
<td>60</td>
<td>IT</td>
<td>103</td>
<td>1400</td>
</tr>
</tbody>
</table>

Delete a row from the DEPARTMENTS table.

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>DEPARTMENT_NAME</th>
<th>MANAGER_ID</th>
<th>LOCATION_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Administration</td>
<td>200</td>
<td>1700</td>
</tr>
<tr>
<td>20</td>
<td>Marketing</td>
<td>201</td>
<td>1800</td>
</tr>
<tr>
<td>30</td>
<td>Purchasing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>Shipping</td>
<td>124</td>
<td>1500</td>
</tr>
<tr>
<td>60</td>
<td>IT</td>
<td>103</td>
<td>1400</td>
</tr>
</tbody>
</table>
The **DELETE** Statement

You can remove existing rows from a table by using the **DELETE** statement.

```sql
DELETE [FROM] table
[WHERE] condition;
```
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.
```
Deleting Rows Based on Another Table

Use subqueries in `DELETE` statements to remove rows from a table based on values from another table.

```sql
DELETE FROM employees
WHERE department_id =
(SELECT department_id
FROM departments
WHERE department_name LIKE 'Public%');
1 row deleted.
```
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.
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.
Using a Subquery in an INSERT Statement

```sql
SELECT employee_id, last_name, email, hire_date, job_id, salary, department_id
FROM employees
WHERE department_id = 50;
```

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>EMAIL</th>
<th>HIRE_DATE</th>
<th>JOB_ID</th>
<th>SALARY</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>124</td>
<td>Bourgos</td>
<td>KMOURGOS</td>
<td>16-NOV-95</td>
<td>ST_MAN</td>
<td>5300</td>
<td>50</td>
</tr>
<tr>
<td>141</td>
<td>Rajs</td>
<td>TRAJS</td>
<td>17-OCT-95</td>
<td>ST_CLERK</td>
<td>3500</td>
<td>50</td>
</tr>
<tr>
<td>142</td>
<td>Davies</td>
<td>CDAVIES</td>
<td>29-JAN-97</td>
<td>ST_CLERK</td>
<td>3100</td>
<td>50</td>
</tr>
<tr>
<td>143</td>
<td>Matos</td>
<td>RMATOS</td>
<td>15-MAR-98</td>
<td>ST_CLERK</td>
<td>2300</td>
<td>50</td>
</tr>
<tr>
<td>144</td>
<td>Vargas</td>
<td>PYARGAS</td>
<td>09-JUL-98</td>
<td>ST_CLERK</td>
<td>2500</td>
<td>50</td>
</tr>
<tr>
<td>99999</td>
<td>Taylor</td>
<td>DTAYLOR</td>
<td>07-JUN-99</td>
<td>ST_CLERK</td>
<td>5000</td>
<td>50</td>
</tr>
</tbody>
</table>

6 rows selected.
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.

```sql
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);
```

ERROR at line 1:
ORA-01402: view WITH CHECK OPTION where-clause violation
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.
Using Explicit Default Values

- **DEFAULT with INSERT:**

```sql
INSERT INTO departments
(department_id, department_name, manager_id)
VALUES (300, 'Engineering', DEFAULT);
```

- **DEFAULT with UPDATE:**

```sql
UPDATE departments
SET manager_id = DEFAULT WHERE department_id = 10;
```
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
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);
```
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);
```
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.
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
Database Transactions

- Begin when the first DML SQL statement is executed
- End with one of the following events:
  - A COMMIT or ROLLBACK statement is issued
  - A DDL or DCL statement executes (automatic commit)
  - The user exits iSQL*Plus
  - The system crashes
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
Controlling Transactions

Transaction

- **DELETE**
- **SAVEPOINT A**
- **INSERT**
- **UPDATE**
- **SAVEPOINT B**
- **INSERT**

- **ROLLBACK** to **SAVEPOINT B**
- **ROLLBACK** to **SAVEPOINT A**
- **ROLLBACK**

**COMMIT**
to a Marker

- Create a marker in a current transaction by using the SAVEPOINT statement.
- Rollback to that marker by using the ROLLBACK TO SAVEPOINT statement.

```
UPDATE...
SAVEPOINT update_done;
Savepoint created.
INSERT...
ROLLBACK TO SAVEPOINT update_done;
Rollback complete.
```
Implicit Transaction Processing

- An automatic commit occurs under the following circumstances:
  - DDL statement is issued
  - DCL statement is issued
  - Normal exit from iSQL*Plus, without explicitly issuing COMMIT or ROLLBACK statements
- An automatic rollback occurs under an abnormal termination of iSQL*Plus or a system failure.
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.
State of the Data after COMMIT

- Locks on the affected rows are released; those rows are available for other users to manipulate.
- All savepoints are erased.
Comitting 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.
State of the Data After ROLLBACK

Discard all pending changes by using the ROLLBACK

- Data changes are undone.
- Previous state of the data is restored.
  Locks on the affected rows are released.

22 rows deleted.
ROLLBACK;
Rollback 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.
Read Consistency

• Read consistency guarantees a consistent view of the data at all times.
• Changes made by one user do not conflict with changes made by another user.
• Read consistency ensures that on the same data:
  – Readers do not wait for writers.
  – Writers do not wait for readers.
Implementation of Read Consistency

```
SELECT * FROM userA.employees;
```

Read consistent image

Data

Rollback

changed and unchanged data before change “old” data
Locking

- Prevent destructive interaction between concurrent transactions
- Require no user action
- Automatically use the lowest level of restrictiveness
- Are held for the duration of the transaction
- Are of two types: explicit locking and implicit locking
Implicit Locking

• Two lock modes:
  – Exclusive: Locks out other users
  – Share: Allows other users to access

• High level of data concurrency:
  – DML: Table share, row exclusive
  – Queries: No locks required
  – DDL: Protects object definitions

• Locks held until commit or rollback
Summary

In this lesson, you should have learned how to use DML statements and control transactions.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSERT</td>
<td>Adds a new row to the table</td>
</tr>
<tr>
<td>UPDATE</td>
<td>Modifies existing rows in the table</td>
</tr>
<tr>
<td>DELETE</td>
<td></td>
</tr>
<tr>
<td>MERGE</td>
<td>Conditionally inserts or updates data in a table</td>
</tr>
<tr>
<td>COMMIT</td>
<td>Makes all pending changes permanent</td>
</tr>
<tr>
<td>SAVEPOINT</td>
<td>Is used to rollback to the savepoint marker</td>
</tr>
<tr>
<td>ROLLBACK</td>
<td>Discards all pending data changes</td>
</tr>
</tbody>
</table>
Practice 8 Overview

- Inserting rows into the tables
- Updating and deleting rows in the table
- Controlling transactions
Creating and Managing Tables
Objectives

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

• Describe the main database objects
• Create tables
• Describe the data types that can be used when specifying column definition
• Alter table definitions
• Drop, rename, and truncate tables
Database Objects

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table</td>
<td>Basic unit of storage; composed of rows and columns</td>
</tr>
<tr>
<td>View</td>
<td>Logically represents subsets of data from one or more tables</td>
</tr>
<tr>
<td>Sequence</td>
<td>Numeric value generator</td>
</tr>
<tr>
<td>Index</td>
<td>Improves the performance of some queries</td>
</tr>
<tr>
<td>Synonym</td>
<td>Gives alternative names to objects</td>
</tr>
</tbody>
</table>
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
The **CREATE TABLE** Statement

- **You must have:**
  - `CREATE TABLE` privilege
  - A storage area

```sql
CREATE TABLE [schema.]table
  (column datatype [DEFAULT expr][, ...]);
```

- **You specify:**
  - Table name
  - Column name, column data type, and column size
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.
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.
Creating Tables

• Create the table.

```
CREATE TABLE dept
    (deptno NUMBER(2),
     dname VARCHAR2(14),
     loc VARCHAR2(13));
```

Table created.

• Confirm table creation.

```
DESCRIBE dept
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Null?</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEPTNO</td>
<td></td>
<td>NUMBER(2)</td>
</tr>
<tr>
<td>DNAME</td>
<td></td>
<td>VARCHAR2(14)</td>
</tr>
<tr>
<td>LOC</td>
<td></td>
<td>VARCHAR2(13)</td>
</tr>
</tbody>
</table>
Tables in the Oracle Database

• User Tables:
  – Are a collection of tables created and maintained by the user
  – Contain user information

• Data Dictionary:
  – Is a collection of tables created and maintained by the Oracle Server
  – Contain database information
Querying the Data Dictionary

- See the names of tables owned by the user.
  
  ```sql
  SELECT table_name
  FROM user_tables;
  ```

- View distinct object types owned by the user.
  
  ```sql
  SELECT DISTINCT object_type
  FROM user_objects;
  ```

- View tables, views, synonyms, and sequences owned by the user.
  
  ```sql
  SELECT *
  FROM user_catalog;
  ```
## Data Types

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARCHAR2(size)</td>
<td>Variable-length character data</td>
</tr>
<tr>
<td>CHAR(size)</td>
<td>Fixed-length character data</td>
</tr>
<tr>
<td>NUMBER(p,s)</td>
<td>Variable-length numeric data</td>
</tr>
<tr>
<td>DATE</td>
<td>Date and time values</td>
</tr>
<tr>
<td>LONG</td>
<td>Variable-length character data up to 2 gigabytes</td>
</tr>
<tr>
<td>CLOB</td>
<td>Character data up to 4 gigabytes</td>
</tr>
<tr>
<td>RAW and LONG RAW</td>
<td>Raw binary data</td>
</tr>
<tr>
<td>BLOB</td>
<td>Binary data up to 4 gigabytes</td>
</tr>
<tr>
<td>BFILE</td>
<td>Binary data stored in an external file; up to 4 gigabytes</td>
</tr>
<tr>
<td>ROWID</td>
<td>A 64 base number system representing the unique address of a row in its table.</td>
</tr>
</tbody>
</table>
DateTime Data Types

Datetime enhancements with Oracle9i:
• New Datetime data types have been introduced.
• New data type storage is available.
• Enhancements have been made to time zones and local time zone.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMESTAMP</td>
<td>Date with fractional seconds</td>
</tr>
<tr>
<td>INTERVAL YEAR TO MONTH</td>
<td>Stored as an interval of years and months</td>
</tr>
<tr>
<td>INTERVAL DAY TO SECOND</td>
<td>Stored as an interval of days to hours minutes and seconds</td>
</tr>
</tbody>
</table>
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)]`
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.

```sql
TIMESTAMP[(fractional_seconds_precision)]
WITH TIME ZONE
```
**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
  ```
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.
```
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.
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.
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.

```sql
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.
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

<table>
<thead>
<tr>
<th>Name</th>
<th>Null?</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPLOYEE_ID</td>
<td>NULL</td>
<td>NUMEER(6)</td>
</tr>
<tr>
<td>LAST_NAME</td>
<td>NOT NULL</td>
<td>VARCHAR2(25)</td>
</tr>
<tr>
<td>ANNSAL</td>
<td>NOT NULL</td>
<td>NUMEER</td>
</tr>
<tr>
<td>HIRE_DATE</td>
<td>NOT NULL</td>
<td>DATE</td>
</tr>
</tbody>
</table>
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
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);
```
Adding a Column

**DEPT80**

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>ANNSAL</th>
<th>HIRE_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>149</td>
<td>Zlotkey</td>
<td>126000</td>
<td>29-JAN-00</td>
</tr>
<tr>
<td>174</td>
<td>Abel</td>
<td>132000</td>
<td>11-MAY-96</td>
</tr>
<tr>
<td>176</td>
<td>Taylor</td>
<td>103200</td>
<td>24-MAR-98</td>
</tr>
</tbody>
</table>

New column

"Add a new column to the DEPT80 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.

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>SALARY</th>
<th>HIRE_DATE</th>
<th>JOB_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>149</td>
<td>Zlotkey</td>
<td>126000</td>
<td>29-JAN-00</td>
<td></td>
</tr>
<tr>
<td>174</td>
<td>Abel</td>
<td>132000</td>
<td>11-MAY-96</td>
<td></td>
</tr>
<tr>
<td>176</td>
<td>Taylor</td>
<td>103200</td>
<td>24-MAR-98</td>
<td></td>
</tr>
</tbody>
</table>
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.
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.
```
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;
```
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.
```
Changing the Name of an Object

- To change the name of a table, view, sequence, or synonym, you execute the `RENAME` statement.

  ```sql
  RENAME dept TO detail_dept;
  Table renamed.
  ```

- You must be the owner of the object.
Truncating a Table

• The `TRUNCATE TABLE` statement:
  - Removes all rows from a table
  - Releases the storage space used by that table

  ```sql
  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.
Adding Comments to a Table

• You can add comments to a table or column by using the `COMMENT` statement.

```sql
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`
## Summary

In this lesson, you should have learned how to use DDL statements to create, alter, drop, and rename tables.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE TABLE</td>
<td>Creates a table</td>
</tr>
<tr>
<td>ALTER TABLE</td>
<td>Modifies table structures</td>
</tr>
<tr>
<td>DROP TABLE</td>
<td>Removes the rows and table structure</td>
</tr>
<tr>
<td>RENAME</td>
<td>Changes the name of a table, view, sequence, or synonym</td>
</tr>
<tr>
<td>TRUNCATE</td>
<td>Removes all rows from a table and releases the storage space</td>
</tr>
<tr>
<td>COMMENT</td>
<td>Adds comments to a table or view</td>
</tr>
</tbody>
</table>
Practice 9 Overview

This practice covers the following topics:

• Creating new tables
• Creating a new table by using the `CREATE TABLE AS` syntax
• Modifying column definitions
• Verifying that the tables exist
• Adding comments to tables
• Dropping tables
• Altering tables
Including Constraints
Objectives

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

• Describe constraints
• Create and maintain constraints
What are Constraints?

- Constraints enforce rules at the table level.
- Constraints prevent the deletion of a table if there are dependencies.
- The following constraint types are valid:
  - NOT NULL
  - UNIQUE
  - PRIMARY KEY
  - FOREIGN KEY
  - CHECK
Constraint Guidelines

• Name a constraint or the Oracle server generates a name by using the `SYS_Cn` format.
• Create a constraint either:
  – At the same time as the table is created, or
  – After the table has been created
• Define a constraint at the column or table level.
• View a constraint in the data dictionary.
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

- Column constraint level
  
  \textit{column} [CONSTRAINT \textit{constraint\_name}] \textit{constraint\_type},

- Table constraint level
  
  \textit{column}, ...
  
  [CONSTRAINT \textit{constraint\_name}] \textit{constraint\_type} (\textit{column}, ...),
The **NOT NULL** Constraint

Ensures that null values are not permitted for the column:

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>EMAIL</th>
<th>PHONE_NUMBER</th>
<th>HIRE_DATE</th>
<th>JOB_ID</th>
<th>SALARY</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>King</td>
<td>SKING</td>
<td>515.123.4567</td>
<td>17-JUN-87</td>
<td>AD_PRES</td>
<td>24000</td>
<td>90</td>
</tr>
<tr>
<td>101</td>
<td>Kochhar</td>
<td>NKOCHHAR</td>
<td>515.123.4568</td>
<td>21-SEP-89</td>
<td>AD_VP</td>
<td>17000</td>
<td>90</td>
</tr>
<tr>
<td>102</td>
<td>De Haan</td>
<td>DEHAAN</td>
<td>515.123.4569</td>
<td>13-JAN-93</td>
<td>AD_VP</td>
<td>17000</td>
<td>90</td>
</tr>
<tr>
<td>103</td>
<td>Hunold</td>
<td>AHUNOLD</td>
<td>590.423.4567</td>
<td>03-JAN-90</td>
<td>IT_PROG</td>
<td>9000</td>
<td>60</td>
</tr>
<tr>
<td>104</td>
<td>Ernst</td>
<td>BEIRSTE</td>
<td>590.423.4568</td>
<td>21-MAY-91</td>
<td>IT_PROG</td>
<td>6000</td>
<td>60</td>
</tr>
<tr>
<td>178</td>
<td>Grant</td>
<td>KRANT</td>
<td>011.44.1644.429263</td>
<td>24-MAY-99</td>
<td>SA_REP</td>
<td>7000</td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>Whalen</td>
<td>JWHALEN</td>
<td>515.123.4444</td>
<td>17-SEP-87</td>
<td>AD_ASST</td>
<td>4400</td>
<td>10</td>
</tr>
</tbody>
</table>

20 rows selected.
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: `emp_hire_date_nn`  
User named: `emp_hire_date_nn`
**The UNIQUE Constraint**

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>EMAIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>King</td>
<td>SKING</td>
</tr>
<tr>
<td>101</td>
<td>Kochhar</td>
<td>NKOCCHAR</td>
</tr>
<tr>
<td>102</td>
<td>De Haan</td>
<td>LDEHAAN</td>
</tr>
<tr>
<td>103</td>
<td>Hunold</td>
<td>AHUNOLD</td>
</tr>
<tr>
<td>104</td>
<td>Ernst</td>
<td>BERNST</td>
</tr>
</tbody>
</table>

... 

Allowed already exists

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>EMAIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>208</td>
<td>Smith</td>
<td>JSMITH</td>
</tr>
<tr>
<td>209</td>
<td>Smith</td>
<td>JSMITH</td>
</tr>
</tbody>
</table>
The **UNIQUE** Constraint

Defined at either the table level or the column level:

```sql
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));
```
## The PRIMARY KEY Constraint

### DEPARTMENTS

**PRIMARY KEY**

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>DEPARTMENT_NAME</th>
<th>MANAGER_ID</th>
<th>LOCATION_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Administration</td>
<td>200</td>
<td>1700</td>
</tr>
<tr>
<td>20</td>
<td>Marketing</td>
<td>201</td>
<td>1800</td>
</tr>
<tr>
<td>50</td>
<td>Shipping</td>
<td>124</td>
<td>1500</td>
</tr>
<tr>
<td>60</td>
<td>IT</td>
<td>103</td>
<td>1400</td>
</tr>
<tr>
<td>80</td>
<td>Sales</td>
<td>149</td>
<td>2500</td>
</tr>
</tbody>
</table>

...
The **PRIMARY KEY** Constraint

Defined at either the table level or the column level:

```sql
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));
```
The FOREIGN KEY Constraint

**DEPARTMENTS**

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>DEPARTMENT_NAME</th>
<th>MANAGER_ID</th>
<th>LOCATION_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Administration</td>
<td>200</td>
<td>1700</td>
</tr>
<tr>
<td>20</td>
<td>Marketing</td>
<td>201</td>
<td>1800</td>
</tr>
<tr>
<td>50</td>
<td>Shipping</td>
<td>124</td>
<td>1500</td>
</tr>
<tr>
<td>60</td>
<td>IT</td>
<td>103</td>
<td>1400</td>
</tr>
<tr>
<td>80</td>
<td>Sales</td>
<td>149</td>
<td>2500</td>
</tr>
</tbody>
</table>

**EMPLOYEES**

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>King</td>
<td>90</td>
</tr>
<tr>
<td>101</td>
<td>Kochhar</td>
<td>90</td>
</tr>
<tr>
<td>102</td>
<td>De Haan</td>
<td>90</td>
</tr>
<tr>
<td>103</td>
<td>Hunold</td>
<td>60</td>
</tr>
<tr>
<td>104</td>
<td>Einst</td>
<td>60</td>
</tr>
<tr>
<td>107</td>
<td>Lorentz</td>
<td>60</td>
</tr>
</tbody>
</table>

**INSERT INTO**

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>200</td>
<td>Ford</td>
<td>9</td>
</tr>
<tr>
<td>201</td>
<td>201</td>
<td>Ford</td>
<td>60</td>
</tr>
</tbody>
</table>

Not allowed (9 does not exist)

Allowed
The **FOREIGN KEY** Constraint

Defined at either the table level or the column level:

```sql
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));
```
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 **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

```sql
..., salary NUMBER(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);```

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.
```
Dropping a Constraint

• Remove the manager constraint from the EMPLOYEES table.

```sql
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.

```sql
ALTER TABLE departments
DROP PRIMARY KEY CASCADE;
Table altered.
```
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.
```
Enabling Constraints

- Activate an integrity constraint currently disabled in the table definition by using the `ENABLE` clause.

```sql
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.
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.
Cascading Constraints

Example:

```
ALTER TABLE test1
DROP (pk) CASCADE CONSTRAINTS;
Table altered.
```

```
ALTER TABLE test1
DROP (pk, fk, col1) CASCADE CONSTRAINTS;
Table altered.
```
Viewing Constraints

Query the `USER_CONSTRAINTS` table to view all constraint definitions and names.

```sql
SELECT constraint_name, constraint_type, search_condition
FROM user_constraints
WHERE table_name = 'EMPLOYEES';
```

<table>
<thead>
<tr>
<th>CONSTRAINT_NAME</th>
<th>C</th>
<th>SEARCH_CONDITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMP_LAST_NAME_NN</td>
<td>C</td>
<td>&quot;LAST_NAME&quot; IS NOT NULL</td>
</tr>
<tr>
<td>EMP_EMAIL_NN</td>
<td>C</td>
<td>&quot;EMAIL&quot; IS NOT NULL</td>
</tr>
<tr>
<td>EMP_HIRE_DATE_NN</td>
<td>C</td>
<td>&quot;HIRE_DATE&quot; IS NOT NULL</td>
</tr>
<tr>
<td>EMP_JOB_NN</td>
<td>C</td>
<td>&quot;JOB_ID&quot; IS NOT NULL</td>
</tr>
<tr>
<td>EMP_SALARY_MIN</td>
<td>C</td>
<td>salary &gt; 0</td>
</tr>
<tr>
<td>EMP_EMAIL_UK</td>
<td>U</td>
<td></td>
</tr>
</tbody>
</table>

...
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';
```

<table>
<thead>
<tr>
<th>CONSTRAINT_NAME</th>
<th>COLUMN_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMP_DEPT_FK</td>
<td>DEPARTMENT_ID</td>
</tr>
<tr>
<td>EMP_EMAIL_NN</td>
<td>EMAIL</td>
</tr>
<tr>
<td>EMP_EMAIL_UK</td>
<td>EMAIL</td>
</tr>
<tr>
<td>EMP_EMP_ID_PK</td>
<td>EMPLOYEE_ID</td>
</tr>
<tr>
<td>EMP_HIRE_DATE_NN</td>
<td>HIRE_DATE</td>
</tr>
<tr>
<td>EMP_JOE_FK</td>
<td>JOB_ID</td>
</tr>
<tr>
<td>EMP_JOE_NN</td>
<td>JOB_ID</td>
</tr>
</tbody>
</table>

...
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.
Practice 10 Overview

This practice covers the following topics:

- Adding constraints to existing tables
- Adding more columns to a table
- Displaying information in data dictionary views
Creating Views
Objectives

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

• Describe a view
• Create, alter the definition of, and drop a view
• Retrieve data through a view
• Insert, update, and delete data through a view
• Create and use an inline view
• Perform “Top-N” analysis
## Database Objects

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table</td>
<td>Basic unit of storage; composed of rows and columns</td>
</tr>
<tr>
<td>View</td>
<td>Logically represents subsets of data from one or more tables</td>
</tr>
<tr>
<td>Sequence</td>
<td>Generates primary key values</td>
</tr>
<tr>
<td>Index</td>
<td>Improves the performance of some queries</td>
</tr>
<tr>
<td>Synonym</td>
<td>Alternative name for an object</td>
</tr>
</tbody>
</table>
What is a View?

**EMPLOYEES Table:**

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>EMAIL</th>
<th>PHONE_NUMBER</th>
<th>HIRE_DATE</th>
<th>JOB_ID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Steven</td>
<td>King</td>
<td>SKING</td>
<td>515.123.4567</td>
<td>17-JUN-87</td>
<td>AD_PRES</td>
<td>240</td>
</tr>
<tr>
<td>101</td>
<td>Neena</td>
<td>Kochhar</td>
<td>NKOCHMAR</td>
<td>515.123.4568</td>
<td>21-SEP-89</td>
<td>AD_VP</td>
<td>170</td>
</tr>
<tr>
<td>102</td>
<td>Lex</td>
<td>De Haan</td>
<td>LDEHAAN</td>
<td>515.123.4569</td>
<td>13-JAN-93</td>
<td>AD_VP</td>
<td>170</td>
</tr>
<tr>
<td>103</td>
<td>Alexander</td>
<td>Hunold</td>
<td>AHUNOLD</td>
<td>590.423.4567</td>
<td>03-JAN-90</td>
<td>IT_PROG</td>
<td>90</td>
</tr>
<tr>
<td>104</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>105</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>106</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>107</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>108</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>109</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>112</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>113</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>114</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>115</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>116</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>117</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>118</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>119</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>120</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>121</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>122</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>123</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>124</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>125</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>126</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>127</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>129</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>130</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>131</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>132</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>133</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>134</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>135</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>136</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>137</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>138</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>139</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>140</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>141</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>142</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>143</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>144</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>145</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>146</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>147</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>148</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>149</td>
<td>Zlotkey</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>150</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>151</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>152</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>153</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>154</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>155</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>156</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>157</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>158</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>159</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>160</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>161</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>162</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>163</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>164</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>165</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>166</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>167</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>168</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>169</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>170</td>
<td>Kimberly</td>
<td>Grant</td>
<td>KORANT</td>
<td>011.44.1044.423205</td>
<td>29-MAY-99</td>
<td>SA_MAN</td>
<td>105</td>
</tr>
<tr>
<td>171</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>172</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>173</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>174</td>
<td>Abel</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11000</td>
</tr>
<tr>
<td>175</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>176</td>
<td>Taylor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6600</td>
</tr>
</tbody>
</table>

20 rows selected.
Why Use Views?

- To restrict data access
- To make complex queries easy
- To provide data independence
- To present different views of the same data
# Simple Views and Complex Views

<table>
<thead>
<tr>
<th>Feature</th>
<th>Simple Views</th>
<th>Complex Views</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of tables</td>
<td>One</td>
<td>One or more</td>
</tr>
<tr>
<td>Contain functions</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Contain groups of data</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>DML operations through a view</td>
<td>Yes</td>
<td>Not always</td>
</tr>
</tbody>
</table>
Creating a View

• You embed a subquery within the `CREATE VIEW` statement.

```sql
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.
Creating a View

• Create a view, EMPVU80, that contains details of employees in department 80.

CREATE VIEW empvu80
AS SELECT employee_id, last_name, salary
FROM employees
WHERE department_id = 80;
View created.

• Describe the structure of the view by using the iSQL*Plus DESCRIBE command.

DESCRIBE empvu80
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.
## Retrieving Data from a View

```
SELECT *
FROM salvu50;
```

<table>
<thead>
<tr>
<th>ID_NUMBER</th>
<th>NAME</th>
<th>ANN_SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>124</td>
<td>Mourgos</td>
<td></td>
</tr>
<tr>
<td>141</td>
<td>Rajs</td>
<td>42000</td>
</tr>
<tr>
<td>142</td>
<td>Davies</td>
<td>37200</td>
</tr>
<tr>
<td>143</td>
<td>Matos</td>
<td>31200</td>
</tr>
<tr>
<td>144</td>
<td>Vargas</td>
<td>30000</td>
</tr>
</tbody>
</table>
Querying a View

**Oracle Server**

```
SELECT employee_id, last_name, salary 
FROM employees 
WHERE department_id=80;
```

**iSQL*Plus**

```
SELECT * 
FROM empvu80;
```

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>149</td>
<td>Zlotkey</td>
<td>10500</td>
</tr>
<tr>
<td>174</td>
<td>Abel</td>
<td>11000</td>
</tr>
<tr>
<td>176</td>
<td>Taylor</td>
<td>6600</td>
</tr>
</tbody>
</table>
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.
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.
```
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
Rules for Performing DML Operations on a View

You cannot modify data in a view if it contains:

- Group functions
- A GROUP BY clause
- The DISTINCT keyword
- The pseudocolumn ROWNUM keyword
- Columns defined by expressions
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
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.

```sql
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.
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.
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.
Removing a View

You can remove a view without losing data because a view is based on underlying tables in the database.

```sql
DROP VIEW view;

DROP VIEW empvu80;
View dropped.
```
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.
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.
Performing Top-N Analysis

The high-level structure of a Top-N analysis query is:

```sql
SELECT [column_list], ROWNUM
FROM   (SELECT [column_list]
        FROM table
        ORDER BY Top-N_column)
WHERE  ROWNUM <= N;
```
Example of Top-N Analysis

To display the top three earner names and salaries from the EMPLOYEES table:

```sql
SELECT ROWNUM as RANK, last_name, salary
FROM (SELECT last_name, salary FROM employees
      ORDER BY salary DESC)
WHERE ROWNUM <= 3;
```

<table>
<thead>
<tr>
<th>RANK</th>
<th>LAST_NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>King</td>
<td>24000</td>
</tr>
<tr>
<td>2</td>
<td>Kochhar</td>
<td>17000</td>
</tr>
<tr>
<td>3</td>
<td>De Haan</td>
<td>17000</td>
</tr>
</tbody>
</table>
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.
Practice 11 Overview

This practice covers the following topics:

- Creating a simple view
- Creating a complex view
- Creating a view with a check constraint
- Attempting to modify data in the view
- Displaying view definitions
- Removing views
Other Database Objects
Objectives

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

• Create, maintain, and use sequences
• Create and maintain indexes
• Create private and public synonyms
# Database Objects

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table</td>
<td>Basic unit of storage; composed of rows and columns</td>
</tr>
<tr>
<td>View</td>
<td>Logically represents subsets of data from one or more tables</td>
</tr>
<tr>
<td>Sequence</td>
<td>Generates primary key values</td>
</tr>
<tr>
<td>Index</td>
<td>Improves the performance of some queries</td>
</tr>
<tr>
<td>Synonym</td>
<td>Alternative name for an object</td>
</tr>
</tbody>
</table>
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
The **CREATE SEQUENCE** Statement Syntax

Define a sequence to generate sequential numbers automatically:

```sql
CREATE SEQUENCE sequence
  [INCREMENT BY n]
  [START WITH n]
  [{MAXVALUE n | NOMAXVALUE}]
  [{MINVALUE n | NOMINVALUE}]
  [{CYCLE | NOCYCLE}]
  [{CACHE n | NOCACHE}];
```
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.

```sql
CREATE SEQUENCE dept_deptid_seq
    INCREMENT BY 10
    START WITH 120
    MAXVALUE 9999
    NOCACHE
    NOCYCLE;
```

Sequence created.
Confirming Sequences

- Verify your sequence values in the USER_SEQUENCES data dictionary table.

```sql
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.
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

- Insert a new department named “Support” in location ID 2500.

```sql
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.

```sql
SELECT dept_deptid_seq.CURRVAL
FROM dual;
```
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.
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.
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.
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.
```
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
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.
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.
```
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
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';
```
Function-Based Indexes

- A function-based index is an index based on expressions.
- The index expression is built from table columns, constants, SQL functions, and user-defined functions.

```sql
CREATE INDEX upper_dept_name_idx
ON departments(UPPER(department_name));

Index created.

SELECT *
FROM departments
WHERE UPPER(department_name) = 'SALES';
```
Removing an Index

- Remove an index from the data dictionary by using the `DROP INDEX` command.

```sql
DROP INDEX index;
```

- Remove the `UPPER_LAST_NAME_IDX` index from the data dictionary.

```sql
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.
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;
```
Creating and Removing Synonyms

- Create a shortened name for the DEPT_SUM_VU view.

  ```sql
  CREATE SYNONYM d_sum
  FOR dept_sum_vu;
  Synonym Created.
  ```

- Drop a synonym.

  ```sql
  DROP SYNONYM d_sum;
  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 USERSEQUENCES 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
Practice 12 Overview

This practice covers the following topics:

- Creating sequences
- Using sequences
- Creating nonunique indexes
- Displaying data dictionary information about sequences and indexes
- Dropping indexes
Controlling User Access
Objectives

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

• Create users
• Create roles to ease setup and maintenance of the security model
• Use the `GRANT` and `REVOKE` statements to grant and revoke object privileges
• Create and access database links
Controlling User Access

Database administrator

Username and password
Privileges

Users
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
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
Creating Users

The DBA creates users by using the `CREATE USER` statement.

```sql
CREATE USER user
IDENTIFIED BY password;
```

```sql
CREATE USER scott
IDENTIFIED BY tiger;
User created.
```
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
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.
What is a Role?

Allocating privileges without a role

Allocating privileges with a role

Users

Manager

Privileges

What is a Role?

Allocating privileges without a role

Allocating privileges with a role

Users

Manager

Privileges

What is a Role?
Creating and Granting Privileges to a Role

• Create a role

```sql
CREATE ROLE manager;
Role created.
```

• Grant privileges to a role

```sql
GRANT create table, create view
TO manager;
Grant succeeded.
```

• Grant a role to users

```sql
GRANT manager TO DEHAAN, KOCHHAR;
Grant succeeded.
```
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.
```
## Object Privileges

<table>
<thead>
<tr>
<th>Object Privilege</th>
<th>Table</th>
<th>View</th>
<th>Sequence</th>
<th>Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALTER</td>
<td>√</td>
<td></td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>DELETE</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXECUTE</td>
<td></td>
<td></td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>INDEX</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INSERT</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REFERENCES</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SELECT</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>UPDATE</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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];
```
Granting Object Privileges

- Grant query privileges on the `EMPLOYEES` table.

```sql
GRANT select
ON employees
TO sue, rich;
Grant succeeded.
```

- Grant privileges to update specific columns to users and roles.

```sql
GRANT update (department_name, location_id)
ON departments
TO scott, manager;
Grant succeeded.
```
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.
```
## Confirming Privileges Granted

<table>
<thead>
<tr>
<th>Data Dictionary View</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROLE_SYS_PRIVS</td>
<td>System privileges granted to roles</td>
</tr>
<tr>
<td>ROLE_TAB_PRIVS</td>
<td>Table privileges granted to roles</td>
</tr>
<tr>
<td>USER_ROLE_PRIVS</td>
<td>Roles accessible by the user</td>
</tr>
<tr>
<td>USER_TAB_PRIVS_MADE</td>
<td>Object privileges granted on the user’s objects</td>
</tr>
<tr>
<td>USER_TAB_PRIVS_RECD</td>
<td>Object privileges granted to the user</td>
</tr>
<tr>
<td>USER_COL_PRIVS_MADE</td>
<td>Object privileges granted on the columns of the user’s objects</td>
</tr>
<tr>
<td>USER_COL_PRIVS_RECD</td>
<td>Object privileges granted to the user on specific columns</td>
</tr>
<tr>
<td>USER_SYS_PRIVS</td>
<td>Lists system privileges granted to the user</td>
</tr>
</tbody>
</table>
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];
```
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.
```
A database link connection allows local users to access data on a remote database.

```
SELECT * FROM emp@HQ_ACME.COM;
```

**Local**

**Remote**

`EMP Table`

`HQ_ACME.COM database`
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;
Summary

In this lesson, you should have learned about DCL statements that control access to the database and database objects:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE USER</td>
<td>Creates a user (usually performed by a DBA)</td>
</tr>
<tr>
<td>GRANT</td>
<td>Gives other users privileges to access the your objects</td>
</tr>
<tr>
<td>CREATE ROLE</td>
<td>Creates a collection of privileges (usually performed by a DBA)</td>
</tr>
<tr>
<td>ALTER USER</td>
<td>Changes a user’s password</td>
</tr>
<tr>
<td>REVOKE</td>
<td>Removes privileges on an object from users</td>
</tr>
</tbody>
</table>
Practice 13 Overview

This practice covers the following topics:

- Granting other users privileges to your table
- Modifying another user’s table through the privileges granted to you
- Creating a synonym
- Querying the data dictionary views related to privileges
SQL Workshop
Workshop Overview

This workshop covers:

• Creating tables and sequences
• Modifying data in the tables
• Modifying table definitions
• Creating views
• Writing scripts containing SQL and iSQL*Plus commands
• Generating a simple report
Using SET Operators
Objectives

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

• Describe SET operators
• Use a SET operator to combine multiple queries into a single query
• Control the order of rows returned
The **SET** Operators

- **UNION/UNION ALL**
- **INTERSECT**
- **MINUS**
Tables Used in This Lesson

The tables used in this lesson are:

- **EMPLOYEES**: Provides details regarding all current employees
- **JOB_HISTORY**: Records the details of the start date and end date of the former job, and the job identification number and department when an employee switches jobs
The **UNION** Operator

A  B

The **UNION** operator returns results from both queries after eliminating duplications.
Using the \texttt{UNION} Operator

Display the current and previous job details of all employees. Display each employee only once.

\begin{verbatim}
SELECT employee_id, job_id 
FROM   employees
UNION
SELECT employee_id, job_id 
FROM   job_history;
\end{verbatim}

\begin{tabular}{|c|c|}
\hline
\textbf{EMPLOYEE\_ID} & \textbf{JOB\_ID} \\
\hline
100 & AD\_PRES \\
101 & AC\_ACCOUNT \\
\hline
200 & AC\_ACCOUNT \\
200 & AD\_ASST \\
\hline
205 & AC\_MGR \\
206 & AC\_ACCOUNT \\
\hline
\end{tabular}
The `UNION ALL` operator returns results from both queries, including all duplications.
Using the `UNION ALL` Operator

Display the current and previous departments of all employees.

```sql
SELECT employee_id, job_id, department_id
FROM employees
UNION ALL
SELECT employee_id, job_id, department_id
FROM job_history
ORDER BY employee_id;
```

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>JOB_ID</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>AD_PRES</td>
<td>90</td>
</tr>
<tr>
<td>101</td>
<td>AD_VP</td>
<td>90</td>
</tr>
<tr>
<td>200</td>
<td>AD_ASST</td>
<td>10</td>
</tr>
<tr>
<td>200</td>
<td>AD_ASST</td>
<td>90</td>
</tr>
<tr>
<td>200</td>
<td>AC_ACCOUNT</td>
<td>90</td>
</tr>
<tr>
<td>205</td>
<td>AC_MGR</td>
<td>110</td>
</tr>
<tr>
<td>206</td>
<td>AC_ACCOUNT</td>
<td>110</td>
</tr>
</tbody>
</table>

30 rows selected.
The INTERSECT Operator
Using the **INTERSECT** Operator

Display the employee IDs and job IDs of employees who currently have a job title that they held before beginning their tenure with the company.

```
SELECT employee_id, job_id
FROM   employees
INTERSECT
SELECT employee_id, job_id
FROM   job_history;
```
The **MINUS** Operator

A

B
The MINUS Operator

Display the employee IDs of those employees who have not changed their jobs even once.

```
SELECT employee_id, job_id
FROM employees
MINUS
SELECT employee_id, job_id
FROM job_history;
```
SET Operator Guidelines

• The expressions in the `SELECT` lists must match in number and data type.
• Parentheses can be used to alter the sequence of execution.
• The `ORDER BY` clause:
  – Can appear only at the very end of the statement
  – Will accept the column name, aliases from the first `SELECT` statement, or the positional notation
The Oracle Server and SET Operators

- Duplicate rows are automatically eliminated except in `UNION ALL`.
- Column names from the first query appear in the result.
- The output is sorted in ascending order by default except in `UNION ALL`.
Matching the `SELECT` Statements

Using the `UNION` operator, display the department ID, location, and hire date for all employees.

```sql
SELECT department_id, TO_NUMBER(null) location, hire_date
FROM employees
UNION
SELECT department_id, location_id, TO_DATE(null) hire_date
FROM departments;
```

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>LOCATION</th>
<th>HIRE_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1700</td>
<td>17-SEP-97</td>
</tr>
<tr>
<td>10</td>
<td>1800</td>
<td>17-FEB-96</td>
</tr>
<tr>
<td>20</td>
<td>1700</td>
<td>07-JUN-94</td>
</tr>
<tr>
<td>190</td>
<td>1700</td>
<td>24-MAY-99</td>
</tr>
</tbody>
</table>

27 rows selected.
Matching the **SELECT** Statement

- Using the **UNION** operator, display the employee ID, job ID, and salary of all employees.

```sql
SELECT employee_id, job_id, salary
FROM employees
UNION
SELECT employee_id, job_id, 0
FROM job_history;
```

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>JOB_ID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>AD_PRES</td>
<td>24000</td>
</tr>
<tr>
<td>101</td>
<td>AC_ACCOUNT</td>
<td>0</td>
</tr>
<tr>
<td>101</td>
<td>AC_MGR</td>
<td>0</td>
</tr>
<tr>
<td>205</td>
<td>AC_MGR</td>
<td>12000</td>
</tr>
<tr>
<td>206</td>
<td>AC_ACCOUNT</td>
<td>8300</td>
</tr>
</tbody>
</table>

30 rows selected.
Controlling the Order of Rows

Produce an English sentence using two \texttt{UNION} operators.

<table>
<thead>
<tr>
<th>My dream</th>
</tr>
</thead>
<tbody>
<tr>
<td>'sing'</td>
</tr>
<tr>
<td>'I'd like to teach'</td>
</tr>
<tr>
<td>'the world to'</td>
</tr>
</tbody>
</table>

\begin{verbatim}
COLUMN a_dummy NOPRINT
SELECT 'sing' AS "My dream", 3 a_dummy
FROM dual
UNION
SELECT 'I\'d like to teach', 1
FROM dual
UNION
SELECT 'the world to', 2
FROM dual
ORDER BY 2;
\end{verbatim}
In this lesson, you should have learned how to:

- Use `UNION` to return all distinct rows
- Use `UNION ALL` to return all rows, including duplicates
- Use `INTERSECT` to return all rows shared by both queries
- Use `MINUS` to return all distinct rows selected by the first query but not by the second
- Use `ORDER BY` only at the very end of the statement
Practice 15 Overview

This practice covers using the Oracle9i datetime functions.
Oracle9i Datetime Functions
Objectives

After completing this lesson, you should be able to use the following datetime functions:

- **TZ_OFFSET**
- **CURRENT_DATE**
- **CURRENT_TIMESTAMP**
- **LOCALTIMESTAMP**
- **DBTIMEZONE**
- **SESSIONTIMEZONE**
- **EXTRACT**
- **FROM_TZ**
- **TO_TIMESTAMP**
- **TO_TIMESTAMP_TZ**
- **TO_YMINTERVAL**
The image represents the time for each time zone when Greenwich time is 12:00.
Oracle9i Datetime Support

• In Oracle9i, you can include the time zone in your date and time data, and provide support for fractional seconds.

• Three new data types are added to DATE:
  - TIMESTAMP
  - TIMESTAMP WITH TIME ZONE (TSTZ)
  - TIMESTAMP WITH LOCAL TIME ZONE (TSLTZ)

• Oracle9i provides daylight savings support for datetime data types in the server.
TZ_OFFSET

- Display the time zone offset for the time zone 'US/Eastern'

```
SELECT TZ_OFFSET('US/Eastern') FROM DUAL;
```

- Display the time zone offset for the time zone 'Canada/Yukon'

```
SELECT TZ_OFFSET('Canada/Yukon') FROM DUAL;
```

- Display the time zone offset for the time zone 'Europe/London'

```
SELECT TZ_OFFSET('Europe/London') FROM DUAL;
```
CURRENT_DATE

- Display the current date and time in the session’s time zone.

```
ALTER SESSION
SET NLS_DATE_FORMAT = 'DD-MON-YYYY HH24:MI:SS';
```

```
ALTER SESSION SET TIME_ZONE = '-5:0';
SELECT SESSIONTIMEZONE, CURRENT_DATE FROM DUAL;
```

<table>
<thead>
<tr>
<th>SESSIONTIMEZONE</th>
<th>CURRENT_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>-05:00</td>
<td>03-OCT-2001 09:37:06</td>
</tr>
</tbody>
</table>

```
ALTER SESSION SET TIME_ZONE = '-8:0';
SELECT SESSIONTIMEZONE, CURRENT_DATE FROM DUAL;
```

<table>
<thead>
<tr>
<th>SESSIONTIMEZONE</th>
<th>CURRENT_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>-08:00</td>
<td>03-OCT-2001 06:38:07</td>
</tr>
</tbody>
</table>

- CURRENT_DATE is sensitive to the session time zone.
- The return value is a date in the Gregorian calendar.
CURRENT_TIMESTAMP

- Display the current date and fractional time in the session's time zone.

```
ALTER SESSION SET TIME_ZONE = '-5:0';
SELECT SESSIONTIMEZONE, CURRENT_TIMESTAMP
FROM DUAL;
```

<table>
<thead>
<tr>
<th>SESSIONTIMEZONE</th>
<th>CURRENT_TIMESTAMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>-05:00</td>
<td>03-OCT-01 09.40.59.000000 AM -05:00</td>
</tr>
</tbody>
</table>

```
ALTER SESSION SET TIME_ZONE = '-8:0';
SELECT SESSIONTIMEZONE, CURRENT_TIMESTAMP
FROM DUAL;
```

<table>
<thead>
<tr>
<th>SESSIONTIMEZONE</th>
<th>CURRENT_TIMESTAMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>-08:00</td>
<td>03-OCT-01 06.41.38.000000 AM -08:00</td>
</tr>
</tbody>
</table>

- CURRENT_TIMESTAMP is sensitive to the session time zone.
- The return value is of the TIMESTAMP WITH TIME ZONE datatype.
LOCALTIMESTAMP

• Display the current date and time in the session time zone in a value of TIMESTAMP data type.

```
ALTER SESSION SET TIME_ZONE = '-5:0';
SELECT CURRENT_TIMESTAMP, LOCALTIMESTAMP
FROM DUAL;
```

<table>
<thead>
<tr>
<th>CURRENT_TIMESTAMP</th>
<th>LOCALTIMESTAMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>03-OCT-01 09.44.21.000000 AM -05:00</td>
<td>03-OCT-01 09.44.21.000000 AM</td>
</tr>
</tbody>
</table>

```
ALTER SESSION SET TIME_ZONE = '-8:0';
SELECT CURRENT_TIMESTAMP, LOCALTIMESTAMP
FROM DUAL;
```

<table>
<thead>
<tr>
<th>CURRENT_TIMESTAMP</th>
<th>LOCALTIMESTAMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>03-OCT-01 06.45.21.000000 AM -08:00</td>
<td>03-OCT-01 06.45.21.000001 AM</td>
</tr>
</tbody>
</table>

• LOCALTIMESTAMP returns a TIMESTAMP value, whereas CURRENT_TIMESTAMP returns a TIMESTAMP WITH TIME ZONE value.
DBTIMEZONE and SESSIONTIMEZONE

• Display the value of the database time zone.

```sql
SELECT DBTIMEZONE FROM DUAL;
```

<table>
<thead>
<tr>
<th>DBTIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>-05:00</td>
</tr>
</tbody>
</table>

• Display the value of the session's time zone.

```sql
SELECT SESSIONTIMEZONE FROM DUAL;
```

<table>
<thead>
<tr>
<th>SESSIONTIMEZONE</th>
</tr>
</thead>
<tbody>
<tr>
<td>-08:00</td>
</tr>
</tbody>
</table>
EXTRACT

- Display the **YEAR** component from the **SYSDATE**.

```sql
SELECT EXTRACT (YEAR FROM SYSDATE) FROM DUAL;
```

<table>
<thead>
<tr>
<th>EXTRACT(YEARFROMSYSDATE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2001</td>
</tr>
</tbody>
</table>

- Display the **MONTH** component from the **HIRE_DATE** for those employees whose **MANAGER_ID** is 100.

```sql
SELECT last_name, hire_date,
       EXTRACT (MONTH FROM HIRE_DATE)
FROM employees
WHERE manager_id = 100;
```

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>HIRE_DATE</th>
<th>EXTRACT(MONTHFROMHIRE_DATE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kochhar</td>
<td>21-SEP-69</td>
<td></td>
</tr>
<tr>
<td>De Haan</td>
<td>13-JAN-93</td>
<td></td>
</tr>
<tr>
<td>Mourgos</td>
<td>16-NOV-99</td>
<td></td>
</tr>
<tr>
<td>Zlotkey</td>
<td>29-JAN-00</td>
<td></td>
</tr>
<tr>
<td>Hartstein</td>
<td>17-FEB-96</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>
TIMESTAMP Conversion Using FROM_TZ

- Display the TIMESTAMP value '2000-03-28 08:00:00' as a TIMESTAMP WITH TIME ZONE value.

```sql
SELECT FROM_TZ(TIMESTAMP '2000-03-28 08:00:00', '3:00')
FROM DUAL;
```

- Display the TIMESTAMP value '2000-03-28 08:00:00' as a TIMESTAMP WITH TIME ZONE value for the time zone region 'Australia/North'.

```sql
SELECT FROM_TZ(TIMESTAMP '2000-03-28 08:00:00', 'Australia/North')
FROM DUAL;
```
STRING To TIMESTAMP Conversion Using TO_TIMESTAMP and TO_TIMESTAMP_TZ

- Display the character string '2000-12-01 11:00:00' as a TIMESTAMP value.

```
SELECT TO_TIMESTAMP ('2000-12-01 11:00:00', 'YYYY-MM-DD HH:MI:SS')
FROM DUAL;
```

```
TO_TIMESTAMP('2000-12-0111:00:00','YYYY-MM-DD HH:MI:SS')
01-DEC-00 11:00:00.000000000 AM
```

- Display the character string '1999-12-01 11:00:00 -8:00' as a TIMESTAMP WITH TIME ZONE value.

```
SELECT TO_TIMESTAMP_TZ ('1999-12-01 11:00:00 -8:00', 'YYYY-MM-DD HH:MI:SS TZH:TZM')
FROM DUAL;
```

```
TO_TIMESTAMP_TZ('1999-12-0111:00:00-8:00','YYYY-MM-DD HH:MI:SS TZH:TZM')
01-DEC-99 11:00:00.000000000 AM -06:00
```
Time Interval Conversion with TO_YMINTERVAL

Display a date that is one year two months after the hire date for the employees working in the department with the DEPARTMENT_ID 20

```
SELECT hire_date,
     hire_date + TO_YMINTERVAL('01-02') AS HIRE_DATE_YMININTERVAL
FROM EMPLOYEES
WHERE department_id = 20;
```

<table>
<thead>
<tr>
<th>HIRE_DATE</th>
<th>HIRE_DATE_YMININTERVAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>17-FEB-1996 00:00:00</td>
<td>17-APR-1997 00:00:00</td>
</tr>
<tr>
<td>17-AUG-1997 00:00:00</td>
<td>17-OCT-1998 00:00:00</td>
</tr>
</tbody>
</table>
Summary

In this lesson, you should have learned how to use the following functions:

- **TZ_OFFSET**
- **FROM_TZ**
- **TO_TIMESTAMP**
- **TO_TIMESTAMP_TZ**
- **TO_YMINTERVAL**
- **CURRENT_DATE**
- **CURRENT_TIMESTAMP**
- **LOCALTIMESTAMP**
- **DBTIMEZONE**
- **SESSIONTIMEZONE**
- **EXTRACT**
Practice 16 Overview

This practice covers using the Oracle9i datetime functions.
Enhancements to the
GROUP BY Clause
Objectives

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

• Use the **ROLLUP** operation to produce subtotal values
• Use the **CUBE** operation to produce cross-tabulation values
• Use the **GROUPING** function to identify the row values created by **ROLLUP** or **CUBE**
• Use **GROUPING SETS** to produce a single result set
Review of Group Functions

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

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

Example:

```
SELECT AVG(salary), STDDEV(salary), COUNT(commission_pct), MAX(hire_date)
FROM employees
WHERE job_id LIKE 'SA%';
```
Review of the \texttt{GROUP BY} Clause

Syntax:

\begin{verbatim}
SELECT [column,] group_function(column). . .
FROM table
[WHERE condition]
[GROUP BY group_by_expression]
[ORDER BY column];
\end{verbatim}

Example:

\begin{verbatim}
SELECT department_id, job_id, SUM(salary),
       COUNT(employee_id)
FROM employees
GROUP BY department_id, job_id;
\end{verbatim}
Review of the **HAVING** Clause

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

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

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

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

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

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

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>JOB_ID</th>
<th>SUM(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>AD_ASST</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>4400</td>
</tr>
<tr>
<td>20</td>
<td>MK_MAN</td>
<td>13000</td>
</tr>
<tr>
<td>20</td>
<td>MK_REP</td>
<td>6000</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>19000</td>
</tr>
<tr>
<td>50</td>
<td>ST_CLERK</td>
<td>11700</td>
</tr>
<tr>
<td>50</td>
<td>ST_MAN</td>
<td>5800</td>
</tr>
<tr>
<td>50</td>
<td></td>
<td>17500</td>
</tr>
<tr>
<td></td>
<td></td>
<td>40900</td>
</tr>
</tbody>
</table>

9 rows selected.
CUBE Operator

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

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

SELECT department_id, job_id, SUM(salary)
FROM employees
WHERE department_id < 60
GROUP BY CUBE (department_id, job_id) ;
GROUPING Function

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

- The **GROUPING** function can be used with either the **CUBE** or **ROLLUP** operator.
- Using the **GROUPING** function, you can find the groups forming the subtotal in a row.
- Using the **GROUPING** function, you can differentiate stored **NULL** values from **NULL** values created by **ROLLUP** or **CUBE**.
- The **GROUPING** function returns 0 or 1.
GROUPING Function: Example

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

<table>
<thead>
<tr>
<th>DEPTID</th>
<th>JOB</th>
<th>SUM(SALARY)</th>
<th>GRP_DEPT</th>
<th>GRP_JOB</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>AD_ASST</td>
<td>4400</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>4400</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>MK_MAN</td>
<td>13000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>MK_REP</td>
<td>6000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>19000</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>23400</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

6 rows selected.
GROUPING SETS

• GROUPING SETS are a further extension of the GROUP BY clause.
• You can use GROUPING SETS to define multiple groupings in the same query.
• The Oracle Server computes all groupings specified in the GROUPING SETS clause and combines the results of individual groupings with a UNION ALL operation.
• Grouping set efficiency:
  – Only one pass over the base table is required.
  – There is no need to write complex UNION statements.
  – The more elements the GROUPING SETS have, the greater the performance benefit.
Hidden Slide
GROUPING SETS: Example

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

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>JOB_ID</th>
<th>MANAGER_ID</th>
<th>AVG(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>AD_ASST</td>
<td></td>
<td>4400</td>
</tr>
<tr>
<td>20</td>
<td>MK_MAN</td>
<td></td>
<td>13000</td>
</tr>
<tr>
<td>20</td>
<td>MK_REP</td>
<td></td>
<td>6000</td>
</tr>
<tr>
<td>50</td>
<td>ST_CLERK</td>
<td></td>
<td>2925</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SA_MAN</td>
<td>100</td>
<td>10500</td>
</tr>
<tr>
<td></td>
<td>SA_REP</td>
<td>149</td>
<td>8966.66667</td>
</tr>
<tr>
<td></td>
<td>ST_CLERK</td>
<td>124</td>
<td>2925</td>
</tr>
<tr>
<td></td>
<td>ST_MAN</td>
<td>100</td>
<td>5800</td>
</tr>
</tbody>
</table>

26 rows selected.
Hidden Slide
Composite Columns

• A composite column is a collection of columns that are treated as a unit.

  ROLLUP (a, (b, c), d)

• To specify composite columns, use the GROUP BY clause to group columns within parentheses so that the Oracle server treats them as a unit while computing ROLLUP or CUBE operations.

• When used with ROLLUP or CUBE, composite columns would mean skipping aggregation across certain levels.
Hidden Slide
SELECT department_id, job_id, manager_id,
       SUM(salary)
FROM employees
GROUP BY ROLLUP ( department_id, (job_id, manager_id));
Hidden Slide
Concatenated Groupings

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

GROUP BY GROUPING SETS(a, b), GROUPING SETS(c, d)
### Concatenated Groupings Example

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

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>JOB_ID</th>
<th>MANAGER_ID</th>
<th>SUM(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>AD_ASST</td>
<td>101</td>
<td>4400</td>
</tr>
<tr>
<td>20</td>
<td>MK_MAN</td>
<td>100</td>
<td>13000</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>101</td>
<td>4400</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>100</td>
<td>13000</td>
</tr>
<tr>
<td>10</td>
<td>AD_ASST</td>
<td></td>
<td>4400</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td>4400</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4400</td>
</tr>
<tr>
<td></td>
<td>SA_REP</td>
<td></td>
<td>7000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>7000</td>
</tr>
</tbody>
</table>

49 rows selected.
Summary

In this lesson, you should have learned how to:

• Use the `ROLLUP` operation to produce subtotal values
• Use the `CUBE` operation to produce cross-tabulation values
• Use the `GROUPING` function to identify the row values created by `ROLLUP` or `CUBE`
• Use the `GROUPING SETS` syntax to define multiple groupings in the same query
• Use the `GROUP BY` clause, to combine expressions in various ways:
  – Composite columns
  – Concatenated grouping sets
Practice 17 Overview

This practice covers the following topics:

• Using the ROLLUP operator
• Using the CUBE operator
• Using the GROUPING function
• Using GROUPING SETS
Hidden Slide
Hidden Slide
Hidden Slide
Hidden Slide
Hidden Slide
Advanced Subqueries
After completing this lesson, you should be able to do the following:

Write a multiple-column subquery

• Describe and explain the behavior of subqueries when null values are retrieved
• Write a subquery in a FROM clause
• Use scalar subqueries in SQL
• Describe the types of problems that can be solved with correlated subqueries
• Write correlated subqueries
• Update and delete rows using correlated subqueries
• Use the EXISTS and NOT EXISTS operators
• Use the WITH clause
What Is a Subquery?

A subquery is a `SELECT` statement embedded in a clause of another SQL statement.

```
SELECT ... FROM   ... WHERE  ...
(SELECT ... FROM   ... WHERE  ...) ...
```
Subqueries

- The subquery (inner query) executes once before the main query.
- The result of the subquery is used by the main query (outer query).

```
SELECT select_list
FROM   table
WHERE  expr operator (SELECT select_list
                       FROM   table);
```
Using a Subquery

```
SELECT last_name
FROM employees
WHERE salary > (SELECT salary
                FROM employees
                WHERE employee_id = 149) ;
```

<table>
<thead>
<tr>
<th>LAST_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
</tr>
<tr>
<td>Kochhar</td>
</tr>
<tr>
<td>De Haan</td>
</tr>
<tr>
<td>Abel</td>
</tr>
<tr>
<td>Hartstein</td>
</tr>
<tr>
<td>Higgins</td>
</tr>
</tbody>
</table>

6 rows selected.
Multiple-Column Subqueries

WHERE (MANAGER_ID, DEPARTMENT_ID) IN

Subquery

100   90
102   60
124   50

Each row of the main query is compared to values from a multiple-row and multiple-column subquery.
Column Comparisons

Column comparisons in a multiple-column subquery can be:

• Pairwise comparisons
• Nonpairwise comparisons
Display the details of the employees who are managed by the same manager and work in the same department as the employees with EMPLOYEE_ID 178 or 174.

```
SELECT employee_id, manager_id, department_id
FROM employees
WHERE (manager_id, department_id) IN
    (SELECT manager_id, department_id
     FROM employees
     WHERE employee_id IN (178,174))
AND employee_id NOT IN (178,174);
```
Nonpairwise Comparison Subquery

Display the details of the employees who are managed by the same manager as the employees with EMPLOYEE_ID 174 or 141 and work in the same department as the employees with EMPLOYEE_ID 174 or 141.

```sql
SELECT employee_id, manager_id, department_id
FROM employees
WHERE manager_id IN
  (SELECT manager_id
   FROM employees
   WHERE employee_id IN (174,141))
AND department_id IN
  (SELECT department_id
   FROM employees
   WHERE employee_id IN (174,141))
AND employee_id NOT IN(174,141);
```
Using a Subquery in the FROM Clause

```
SELECT  a.last_name, a.salary, 
        a.department_id, b.salavg
FROM    employees a, (SELECT   department_id, 
                      AVG(salary) salavg
                      FROM     employees
                      GROUP BY department_id) b
WHERE   a.department_id = b.department_id
AND     a.salary > b.salavg;
```

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>SALARY</th>
<th>DEPARTMENT_ID</th>
<th>SALAVG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hartstein</td>
<td>13000</td>
<td>20</td>
<td>9500</td>
</tr>
<tr>
<td>Mourgos</td>
<td>5800</td>
<td>50</td>
<td>3500</td>
</tr>
<tr>
<td>Hunold</td>
<td>9000</td>
<td>60</td>
<td>6400</td>
</tr>
<tr>
<td>Zlotkey</td>
<td>10500</td>
<td>80</td>
<td>10033.333</td>
</tr>
<tr>
<td>Abel</td>
<td>11000</td>
<td>80</td>
<td>10033.333</td>
</tr>
<tr>
<td>King</td>
<td>24000</td>
<td>90</td>
<td>19333.333</td>
</tr>
<tr>
<td>Higgins</td>
<td>12000</td>
<td>110</td>
<td>10150</td>
</tr>
</tbody>
</table>

7 rows selected.
Scalar Subquery Expressions

• A scalar subquery expression is a subquery that returns exactly one column value from one row.

• Scalar subqueries were supported in Oracle8i only in a limited set of cases. For example:
  – SELECT statement (FROM and WHERE clauses)
  – VALUES list of an INSERT statement

• In Oracle9i, scalar subqueries can be used in:
  – Condition and expression part of DECODE and CASE
  – All clauses of SELECT except GROUP BY
Scalar Subqueries: Examples

Scalar Subqueries in `CASE` Expressions

SELECT employee_id, last_name,
    (CASE
        WHEN department_id = (SELECT department_id FROM departments WHERE location_id = 1800)
        THEN 'Canada' ELSE 'USA' END) location
FROM employees;

Scalar Subqueries in `ORDER BY` Clause

SELECT employee_id, last_name
FROM employees e
ORDER BY (SELECT department_name
            FROM departments d
            WHERE e.department_id = d.department_id);
Correlated Subqueries

Correlated subqueries are used for row-by-row processing. Each subquery is executed once for every row of the outer query.

- **GET**
  - candidate row from outer query

- **EXECUTE**
  - inner query using candidate row value

- **USE**
  - values from inner query to qualify or disqualify candidate row
Correlated Subqueries

```
SELECT column1, column2, ...
FROM table1
WHERE column1 operator
    (SELECT column1, column2
     FROM table2
     WHERE expr1 = outer.expr2);
```

The subquery references a column from a table in the parent query.
Using Correlated Subqueries

Find all employees who earn more than the average salary in their department.

```sql
SELECT last_name, salary, department_id
FROM employees outer
WHERE salary >
    (SELECT AVG(salary)
     FROM employees
     WHERE department_id =
         outer.department_id);
```

Each time a row from the outer query is processed, the inner query is evaluated.
Using Correlated Subqueries

Display details of those employees who have switched jobs at least twice.

```
SELECT e.employee_id, last_name, e.job_id
FROM employees e
WHERE 2 <= (SELECT COUNT(*)
            FROM job_history
            WHERE employee_id = e.employee_id);
```

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>JOB_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Kochhar</td>
<td>AD_VP</td>
</tr>
<tr>
<td>176</td>
<td>Taylor</td>
<td>SA REP</td>
</tr>
<tr>
<td>200</td>
<td>Whalen</td>
<td>AD ASST</td>
</tr>
</tbody>
</table>
Using the EXISTS Operator

- The **EXISTS** operator tests for existence of rows in the results set of the subquery.
- If a subquery row value is found:
  - The search does not continue in the inner query
  - The condition is flagged **TRUE**
- If a subquery row value is not found:
  - The condition is flagged **FALSE**
  - The search continues in the inner query
Using the **EXISTS** Operator

Find employees who have at least one person reporting to them.

```
SELECT employee_id, last_name, job_id, department_id
FROM employees outer
WHERE EXISTS ( SELECT 'X'
    FROM employees
    WHERE manager_id = outer.employee_id);
```

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>JOB_ID</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>King</td>
<td>AD_FRES</td>
<td>90</td>
</tr>
<tr>
<td>101</td>
<td>Kochhar</td>
<td>AD_VP</td>
<td>90</td>
</tr>
<tr>
<td>102</td>
<td>De Haan</td>
<td>AD_VP</td>
<td>90</td>
</tr>
<tr>
<td>103</td>
<td>Hunold</td>
<td>IT_PROG</td>
<td>60</td>
</tr>
<tr>
<td>124</td>
<td>Mourgos</td>
<td>ST_MAN</td>
<td>50</td>
</tr>
<tr>
<td>149</td>
<td>Zlotkey</td>
<td>SA_MAN</td>
<td>80</td>
</tr>
<tr>
<td>201</td>
<td>Hartstein</td>
<td>MK_MAN</td>
<td>20</td>
</tr>
<tr>
<td>205</td>
<td>Higgins</td>
<td>AC_MGR</td>
<td>110</td>
</tr>
</tbody>
</table>

8 rows selected.
Using the **NOT EXISTS** Operator

Find all departments that do not have any employees.

```sql
SELECT department_id, department_name
FROM departments d
WHERE NOT EXISTS (SELECT 'X'
    FROM employees
    WHERE department_id = d.department_id);
```

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>DEPARTMENT_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>190</td>
<td>Contracting</td>
</tr>
</tbody>
</table>
Correlated UPDATE

UPDATE table1 alias1
SET column = (SELECT expression
             FROM table2 alias2
             WHERE alias1.column = alias2.column);

Use a correlated subquery to update rows in one table based on rows from another table.
Correlated **UPDATE**

- Denormalize the **EMPLOYEES** table by adding a column to store the department name.
- Populate the table by using a correlated update.

```sql
ALTER TABLE employees
ADD(department_name VARCHAR2(14));

UPDATE employees e
SET department_name =
    (SELECT department_name
     FROM departments d
     WHERE e.department_id = d.department_id);
```
Correlated DELETE

```sql
DELETE FROM table1 alias1
WHERE column operator (SELECT expression
FROM table2 alias2
WHERE alias1.column = alias2.column);
```

Use a correlated subquery to delete rows in one table based on rows from another table.
Correlated DELETE

Use a correlated subquery to delete only those rows from the EMPLOYEES table that also exist in the EMP_HISTORY table.

```
DELETE FROM employees E
WHERE employee_id =
    (SELECT employee_id
     FROM emp_history
     WHERE employee_id = E.employee_id);
```
The `WITH` Clause

- Using the `WITH` clause, you can use the same query block in a `SELECT` statement when it occurs more than once within a complex query.
- The `WITH` clause retrieves the results of a query block and stores it in the user's temporary tablespace.
- The `WITH` clause improves performance
**WITH Clause: Example**

Using the `WITH` clause, write a query to display the department name and total salaries for those departments whose total salary is greater than the average salary across departments.
WITH Clause: Example

WITH

department_costs AS (  
SELECT d.department_name, SUM(e.salary) AS dept_total  
FROM employees e, departments d  
WHERE e.department_id = d.department_id  
GROUP BY d.department_name),

avg_cost AS (  
SELECT SUM(dept_total)/COUNT(*) AS dept_avg  
FROM department_costs)

SELECT *  
FROM department_costs  
WHERE dept_total >  
(SELECT dept_avg  
FROM avg_cost)  
ORDER BY department_name;
Summary

In this lesson, you should have learned the following:

• A multiple-column subquery returns more than one column.
• Multiple-column comparisons can be pairwise or nonpairwise.
• A multiple-column subquery can also be used in the FROM clause of a SELECT statement.
• Scalar subqueries have been enhanced in Oracle9i.
Summary

- Correlated subqueries are useful whenever a subquery must return a different result for each candidate row.
- The `EXISTS` operator is a Boolean operator that tests the presence of a value.
- Correlated subqueries can be used with `SELECT`, `UPDATE`, and `DELETE` statements.
- You can use the `WITH` clause to use the same query block in a `SELECT` statement when it occurs more than once.
Practice 18 Overview

This practice covers the following topics:

- Creating multiple-column subqueries
- Writing correlated subqueries
- Using the `EXISTS` operator
- Using scalar subqueries
- Using the `WITH` clause
Objectives

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

• Interpret the concept of a hierarchical query
• Create a tree-structured report
• Format hierarchical data
• Exclude branches from the tree structure
Sample Data from the EMPLOYEES Table

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>JOB_ID</th>
<th>MANAGER_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>King</td>
<td>AD_PRES</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>Kochhar</td>
<td>AD_VP</td>
<td>100</td>
</tr>
<tr>
<td>102</td>
<td>De Haan</td>
<td>AD_VP</td>
<td>100</td>
</tr>
<tr>
<td>103</td>
<td>Hunold</td>
<td>IT_PROG</td>
<td>102</td>
</tr>
<tr>
<td>104</td>
<td>Einst</td>
<td>IT_PROG</td>
<td>103</td>
</tr>
<tr>
<td>107</td>
<td>Lorentz</td>
<td>IT_PROG</td>
<td>103</td>
</tr>
<tr>
<td>124</td>
<td>Morgos</td>
<td>ST_MAN</td>
<td>100</td>
</tr>
<tr>
<td>141</td>
<td>Rajs</td>
<td>ST_CLERK</td>
<td>124</td>
</tr>
<tr>
<td>142</td>
<td>Davies</td>
<td>ST_CLERK</td>
<td>124</td>
</tr>
<tr>
<td>143</td>
<td>Matos</td>
<td>ST_CLERK</td>
<td>124</td>
</tr>
<tr>
<td>144</td>
<td>Vargas</td>
<td>ST_CLERK</td>
<td>124</td>
</tr>
<tr>
<td>149</td>
<td>Zlotkey</td>
<td>SA_MAN</td>
<td>100</td>
</tr>
<tr>
<td>174</td>
<td>Abel</td>
<td>SA_REP</td>
<td>149</td>
</tr>
<tr>
<td>176</td>
<td>Taylor</td>
<td>SA_REP</td>
<td>149</td>
</tr>
<tr>
<td>178</td>
<td>Grant</td>
<td>SA_REP</td>
<td>149</td>
</tr>
<tr>
<td>200</td>
<td>Whalen</td>
<td>AD_ASST</td>
<td>101</td>
</tr>
<tr>
<td>201</td>
<td>Hartstein</td>
<td>MK_MAN</td>
<td>100</td>
</tr>
<tr>
<td>202</td>
<td>Fay</td>
<td>MK_REP</td>
<td>201</td>
</tr>
<tr>
<td>205</td>
<td>Higgins</td>
<td>AC_MGR</td>
<td>101</td>
</tr>
<tr>
<td>206</td>
<td>Gietz</td>
<td>AC_ACCOUNT</td>
<td>205</td>
</tr>
</tbody>
</table>

20 rows selected.
Natural Tree Structure

**EMPLOYEE_ID = 100** (Parent)

**MANAGER_ID = 100** (Child)

- King
  - Mourgos
    - Zlotkey
    - Hartstein
  - Rajs
    - Abel
    - Taylor
    - Grant
  - Davies
  - Matos
  - Vargas

- Kochhar
  - Whalen
  - Higgins
  - Hunold
    - Gietz
    - Ernst
    - Lorentz

ORACLE
Hierarchical Queries

WHERE condition:

expr comparison_operator expr
Starting Point

• Specifies the condition that must be met
• Accepts any valid condition

START WITH column1 = value

Using the EMPLOYEES table, start with the employee whose last name is Kochhar.

...START WITH last_name = 'Kochhar'
Walking the Tree

Direction

Top down

Column1 = Parent Key
Column2 = Child Key

Bottom up

Column1 = Child Key
Column2 = Parent Key

CONNECT BY PRIOR column1 = column2

... CONNECT BY PRIOR employee_id = manager_id

Walk from the top down, using the EMPLOYEES table.
SELECT employee_id, last_name, job_id, manager_id
FROM   employees
START  WITH  employee_id = 101
CONNECT BY PRIOR manager_id = employee_id ;

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>JOB_ID</th>
<th>MANAGER_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Kochhar</td>
<td>AD_VP</td>
<td>100</td>
</tr>
<tr>
<td>100</td>
<td>King</td>
<td>AD_PRES</td>
<td></td>
</tr>
</tbody>
</table>
SELECT last_name||' reports to '||PRIOR last_name "Walk Top Down"
FROM employees
START WITH last_name = 'King'
CONNECT BY PRIOR employee_id = manager_id ;

<table>
<thead>
<tr>
<th>Walk Top Down</th>
</tr>
</thead>
<tbody>
<tr>
<td>King reports to</td>
</tr>
<tr>
<td>Kochhar reports to King</td>
</tr>
<tr>
<td>Whalen reports to Kochhar</td>
</tr>
<tr>
<td>Higgins reports to Kochhar</td>
</tr>
<tr>
<td>Zlotkey reports to King</td>
</tr>
<tr>
<td>Abel reports to Zlotkey</td>
</tr>
<tr>
<td>Taylor reports to Zlotkey</td>
</tr>
<tr>
<td>Grant reports to Zlotkey</td>
</tr>
<tr>
<td>Hartstein reports to King</td>
</tr>
<tr>
<td>Fay reports to Hartstein</td>
</tr>
</tbody>
</table>

20 rows selected.
Ranking Rows with the LEVEL Pseudocolumn

Level 1
root/parent

King

Level 2
parent/child

Mourgos

Zlotkey

Hartstein

Level 3
parent/child/leaf

Rajs
Davies
Matos
Vargas

Abel
Taylor
Grant

Level 4
leaf

Whalen
Higgins
Hunold

Gietz
Ernst
Lorentz

Kochhar
De Hann
Formatting Hierarchical Reports Using \texttt{LEVEL} and \texttt{LPAD}

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

\begin{verbatim}
COLUMN org_chart FORMAT A12
SELECT LPAD(last_name, LENGTH(last_name)+(LEVEL*2)-2,'_') AS org_chart
FROM employees
START WITH last_name='King'
CONNECT BY PRIOR employee_id=manager_id
\end{verbatim}
Pruning Branches

Use the **WHERE** clause to eliminate a node.

WHERE last_name != 'Higgins'

Use the **CONNECT BY** clause to eliminate a branch.

CONNECT BY PRIOR
employee_id = manager_id
AND last_name != 'Higgins'
Summary

In this lesson, you should have learned the following:

• You can use hierarchical queries to view a hierarchical relationship between rows in a table.
• You specify the direction and starting point of the query.
• You can eliminate nodes or branches by pruning.
Practice 19 Overview

This practice covers the following topics:

- Distinguishing hierarchical queries from nonhierarchical queries
- Walking through a tree
- Producing an indented report by using the `LEVEL` pseudocolumn
- Pruning the tree structure
- Sorting the output
Oracle9i Extensions to DML and DDL Statements
Objectives

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

• Describe the features of multitable inserts
• Use the following types of multitable inserts
  – Unconditional \texttt{INSERT}
  – Pivoting \texttt{INSERT}
  – Conditional \texttt{ALL INSERT}
  – Conditional \texttt{FIRST INSERT}
• Create and use external tables
• Name the index at the time of creating a primary key constraint
Review of the INSERT Statement

• Add new rows to a table by using the INSERT statement.

```sql
INSERT INTO table [(column [, column...])]
VALUES (value [, value...]);
```

• Only one row is inserted at a time with this syntax.

```sql
INSERT INTO departments(department_id, department_name, manager_id, location_id)
VALUES (70, 'Public Relations', 100, 1700);
1 row created.
```
Review of the **UPDATE** Statement

- **Modify existing rows with the **UPDATE** statement.**

  ```
  UPDATE table
  SET column = value [, column = value, ...]
  [WHERE condition];
  ```

- **Update more than one row at a time, if required.**
- **Specific row or rows are modified if you specify the **WHERE** clause.**

  ```
  UPDATE employees
  SET department_id = 70
  WHERE employee_id = 142;
  1 row updated.
  ```
Overview of Multitable INSERT Statements

- The `INSERT...SELECT` statement can be used to insert rows into multiple tables as part of a single DML statement.

- Multitable `INSERT` statements can be used in data warehousing systems to transfer data from one or more operational sources to a set of target tables.

- They provide significant performance improvement over:
  - Single DML versus multiple `INSERT...SELECT` statements
  - Single DML versus a procedure to do multiple inserts using `IF...THEN` syntax
Types of Multitable INSERT Statements

Oracle9i introduces the following types of multitable insert statements:

- Unconditional INSERT
- Conditional ALL INSERT
- Conditional FIRST INSERT
- Pivoting INSERT
Multitable INSERT Statements

Syntax

```
INSERT [ALL] [conditional_insert_clause] [insert_into_clause values_clause] (subquery)
```

```
conditional_insert_clause

[ALL] [FIRST] [WHEN condition THEN] [insert_into_clause values_clause] [ELSE] [insert_into_clause values_clause]
```
Unconditional INSERT ALL

- Select the EMPLOYEE_ID, HIRE_DATE, SALARY, and MANAGER_ID values from the EMPLOYEES table for those employees whose EMPLOYEE_ID is greater than 200.
- Insert these values into the SAL_HISTORY and MGR_HISTORY tables using a multitable INSERT.

```
INSERT ALL
INTO sal_history VALUES(EMPID,HIREDATE,SAL)
INTO mgr_history VALUES(EMPID,MGR,SAL)
SELECT employee_id EMPID, hire_date HIREDATE,
     salary SAL, manager_id MGR
FROM employees
WHERE employee_id > 200;
8 rows created.
```
Conditional INSERT ALL

• Select the EMPLOYEE_ID, HIRE_DATE, SALARY and MANAGER_ID values from the EMPLOYEES table for those employees whose EMPLOYEE_ID is greater than 200.

• If the SALARY is greater than $10,000, insert these values into the SAL_HISTORY table using a conditional multitable INSERT statement.

• If the MANAGER_ID is greater than 200, insert these values into the MGR_HISTORY table using a conditional multitable INSERT statement.
Conditional INSERT ALL

```sql
INSERT ALL
  WHEN SAL > 10000 THEN
      INTO sal_history VALUES(EMPID, HIREDATE, SAL)
  WHEN MGR > 200 THEN
      INTO mgr_history VALUES(EMPID, MGR, SAL)
  SELECT employee_id EMPID, hire_date HIREDATE,
       salary SAL, manager_id MGR
  FROM employees
  WHERE employee_id > 200;
```

4 rows created.
Conditional FIRST INSERT

- Select the `DEPARTMENT_ID`, `SUM(SALARY)` and `MAX(HIRE_DATE)` from the `EMPLOYEES` table.
- If the `SUM(SALARY)` is greater than $25,000 then insert these values into the `SPECIAL_SAL`, using a conditional FIRST multitable INSERT.
- If the first `WHEN` clause evaluates to true, the subsequent `WHEN` clauses for this row should be skipped.
- For the rows that do not satisfy the first `WHEN` condition, insert into the `HIREDATE_HISTORY_00`, or `HIREDATE_HISTORY_99`, or `HIREDATE_HISTORY` tables, based on the value in the `HIRE_DATE` column using a conditional multitable INSERT.
Conditional **FIRST** INSERT

```
INSERT **FIRST**
  WHEN SAL > 25000 THEN
    INTO special_sal VALUES(DEPTID, SAL)
  WHEN HIREDATE like ('%00%') THEN
    INTO hiredate_history_00 VALUES(DEPTID, HIREDATE)
  WHEN HIREDATE like ('%99%') THEN
    INTO hiredate_history_99 VALUES(DEPTID, HIREDATE)
ELSE
  INTO hiredate_history VALUES(DEPTID, HIREDATE)
SELECT department_id DEPTID, SUM(salary) SAL,
    MAX(hire_date) HIREDATE
FROM employees
GROUP BY department_id;
8 rows created.
```
Pivoting `INSERT`

- Suppose you receive a set of sales records from a nonrelational database table, `SALES_SOURCE_DATA` in the following format:

  `EMPLOYEE_ID, WEEK_ID, SALES_MON, SALES_TUE, SALES_WED, SALES_THUR, SALES_FRI`

- You would want to store these records in the `SALES_INFO` table in a more typical relational format:

  `EMPLOYEE_ID, WEEK, SALES`

- Using a **pivoting `INSERT`**, convert the set of sales records from the nonrelational database table to relational format.
INSERT ALL

INTO sales_info VALUES (employee_id, week_id, sales_MON)
INTO sales_info VALUES (employee_id, week_id, sales_TUE)
INTO sales_info VALUES (employee_id, week_id, sales_WED)
INTO sales_info VALUES (employee_id, week_id, sales_THUR)
INTO sales_info VALUES (employee_id, week_id, sales_FRI)

SELECT EMPLOYEE_ID, week_id, sales_MON, sales_TUE, sales_WED, sales_THUR, sales_FRI
FROM sales_source_data;

5 rows created.
External Tables

• External tables are read-only tables in which the data is stored outside the database in flat files.
• The metadata for an external table is created using a CREATE TABLE statement.
• With the help of external tables, Oracle data can be stored or unloaded as flat files.
• The data can be queried using SQL, but you cannot use DML and no indexes can be created.
Creating an External Table

- Use the `external_table_clause` along with the `CREATE TABLE` syntax to create an external table.
- Specify `ORGANIZATION AS EXTERNAL` to indicate that the table is located outside the database.
- The `external_table_clause` consists of the access driver `TYPE`, `external_data_properties`, and the `REJECT LIMIT`.
- The `external_data_properties` consist of the following:
  - `DEFAULT DIRECTORY`
  - `ACCESS PARAMETERS`
  - `LOCATION`
Example of Creating an External Table

Create a `DIRECTORY` object that corresponds to the directory on the file system where the external data source resides.

```
CREATE DIRECTORY emp_dir AS '/flat_files' ;
```
Example of Creating an External Table

```
CREATE TABLE oldemp (  
    empno NUMBER, empname CHAR(20), birthdate DATE)  
ORGANIZATION EXTERNAL  
    (TYPE ORACLE_LOADER  
    DEFAULT DIRECTORY emp_dir  
    ACCESS PARAMETERS  
    (RECORDS DELIMITED BY NEWLINE  
    BADFILE 'bad_emp'  
    LOGFILE 'log_emp'  
    FIELDS TERMINATED BY ','  
    (empno CHAR,  
    empname CHAR,  
    birthdate CHAR date_format date mask "dd-mon-yyyy"))  
LOCATION ('emp1.txt'))  
PARALLEL 5  
REJECT LIMIT 200;
```

Table created.
Querying External Tables

```
SELECT * 
FROM oldemp
```
CREATE TABLE NEW_EMP
(employee_id NUMBER(6)
  PRIMARY KEY USING INDEX
  (CREATE INDEX emp_id_idx ON
   NEW_EMP(employee_id)),
first_name  VARCHAR2(20),
last_name   VARCHAR2(25));
Table created.

SELECT INDEX_NAME, TABLE_NAME
FROM USER_INDEXES
WHERE TABLE_NAME = 'NEW_EMP';

<table>
<thead>
<tr>
<th>INDEX_NAME</th>
<th>TABLE_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMP_ID_IDX</td>
<td>NEW_EMP</td>
</tr>
</tbody>
</table>
Summary

In this lesson, you should have learned how to:

• Use the `INSERT...SELECT` statement to insert rows into multiple tables as part of a single DML statement
• Create external tables
• Name indexes using the `CREATE INDEX` statement along with the `CREATE TABLE` statement
Practice 20 Overview

This practice covers the following topics:

• Writing unconditional `INSERT` statements
• Writing conditional `ALL INSERT` statements
• Pivoting `INSERT` statements
• Creating indexes along with the `CREATE TABLE` command