Introduction to Oracle9i: SQL

Student Guide • Volume 2
Contents

Preface
Curriculum Map

Introduction
Objectives  I-2
Oracle9i  I-3
Oracle9i Application Server  I-5
Oracle9i Database  I-6
Oracle9i: Object Relational Database Management System  I-8
Oracle Internet Platform  I-9
System Development Life Cycle  I-10
Data Storage on Different Media  I-12
Relational Database Concept  I-13
Definition of a Relational Database  I-14
Data Models  I-15
Entity Relationship Model  I-16
Entity Relationship Modeling Conventions  I-17
Relating Multiple Tables  I-19
Relational Database Terminology  I-20
Relational Database Properties  I-21
Communicating with a RDBMS Using SQL  I-22
Relational Database Management System  I-23
SQL Statements  I-24
Tables Used in the Course  I-25
Summary  I-26

1 Writing Basic SQL SELECT Statements
Objectives  1-2
Capabilities of SQL SELECT Statements  1-3
Basic SELECT Statement  1-4
2 Restricting and Sorting Data

Objectives 2-2
Limiting Rows Using a Selection 2-3
Limiting the Rows Selected 2-4
Using the WHERE Clause 2-5
Character Strings and Dates 2-6
Comparison Conditions 2-7
Using Comparison Conditions 2-8
Other Comparison Conditions 2-9
Using the BETWEEN Condition 2-10
Using the IN Condition 2-11
Using the LIKE Condition 2-12
Using the NULL Conditions 2-14
Logical Conditions 2-15
Using the AND Operator 2-16
Using the OR Operator 2-17
Using the NOT Operator 2-18
Rules of Precedence 2-19
ORDER BY Clause 2-22
Sorting in Descending Order 2-23
Sorting by Column Alias 2-24
Sorting by Multiple Columns 2-25
Summary 2-26
Practice 2 Overview 2-27
3 Single-Row Functions

Objectives  3-2

SQL Functions  3-3

Two Types of SQL Functions  3-4

Single-Row Functions  3-5

Character Functions  3-7

Case Manipulation Functions  3-9

Using Case Manipulation Functions  3-10

Character-Manipulation Functions  3-11

Using the Character-Manipulation Functions  3-12

Number Functions  3-13

Using the ROUND Function  3-14

Using the TRUNC Function  3-15

Using the MOD Function  3-16

Working with Dates  3-17

Arithmetic with Dates  3-19

Using Arithmetic Operators with Dates  3-20

Date Functions  3-21

Using Date Functions  3-22

Practice 3, Part 1 Overview  3-24

Conversion Functions  3-25

Implicit Data-Type Conversion  3-26

Explicit Data-Type Conversion  3-28

Using the TO_CHAR Function with Dates  3-31

Elements of the Date Format Model  3-32

Using the TO_CHAR Function with Dates  3-36
4 Displaying Data from Multiple Tables

Objectives  4-2
Obtaining Data from Multiple Tables  4-3
Cartesian Products  4-4
Generating a Cartesian Product  4-5
Types of Joins  4-6
Joining Tables Using Oracle Syntax  4-7
What Is an Equijoin?  4-8
Retrieving Records with Equijoins  4-9
Additional Search Conditions Using the AND Operator  4-10
Qualifying Ambiguous Column Names  4-11
Using Table Aliases  4-12
Joining More than Two Tables  4-13
Nonequijoins  4-14
Retrieving Records with Nonequijoins  4-15
Outer Joins  4-16
Outer Joins Syntax  4-17
Using Outer Joins  4-18
Self Joins  4-19
Joining a Table to Itself  4-20
Practice 4, Part 1 Overview  4-21
Joining Tables Using SQL: 1999 Syntax  4-22
Creating Cross Joins  4-23
Creating Natural Joins  4-24
Retrieving Records with Natural Joins  4-25
Creating Joins with the USING Clause  4-26
Retrieving Records with the USING Clause  4-27
Creating Joins with the ON Clause  4-28
Retrieving Records with the ON Clause  4-29
Creating Three-Way Joins with the ON Clause  4-30
INNER versus OUTER Joins  4-31
LEFT OUTER JOIN  4-32
RIGHT OUTER JOIN  4-33

viii
5 Aggregating Data Using Group Functions

Objectives 5-2
What Are Group Functions? 5-3
Types of Group Functions 5-4
Group Functions Syntax 5-5
Using the AVG and SUM Functions 5-6
Using the MIN and MAX Functions 5-7
Using the COUNT Function 5-8
Using the DISTINCT Keyword 5-10
Group Functions and Null Values 5-11
Using the NVL Function with Group Functions 5-12
Creating Groups of Data 5-13
Creating Groups of Data: GROUP BY Clause Syntax 5-14
Using the GROUP BY Clause 5-15
Grouping by More Than One Column 5-17
Using the GROUP BY Clause on Multiple Columns 5-18
Illegal Queries Using Group Functions 5-19
Excluding Group Results 5-21
Excluding Group Results: The HAVING Clause 5-22
Using the HAVING Clause 5-23
Nesting Group Functions 5-25
Summary 5-26
Practice 5 Overview 5-27
6 Subqueries

Objectives 6-2

Using a Subquery to Solve a Problem 6-3

Subquery Syntax 6-4

Using a Subquery 6-5

Guidelines for Using Subqueries 6-6

Types of Subqueries 6-7

Single-Row Subqueries 6-8

Executing Single-Row Subqueries 6-9

Using Group Functions in a Subquery 6-10

The HAVING Clause with Subqueries 6-11

What Is Wrong with This Statement? 6-12

Will This Statement Return Rows? 6-13

Multiple-Row Subqueries 6-14

Using the ANY Operator in Multiple-Row Subqueries 6-15

Using the ALL Operator in Multiple-Row Subqueries 6-16

Null Values in a Subquery 6-17

Summary 6-18

Practice 6 Overview 6-19

7 Producing Readable Output with iSQL*Plus

Objectives 7-2

Substitution Variables 7-3

Using the & Substitution Variable 7-5

Character and Date Values with Substitution Variables 7-7

Specifying Column Names, Expressions, and Text 7-8
Defining Substitution Variables  7-10
DEFINE and UNDEFINE Commands  7-11
Using the DEFINE Command with & Substitution Variable  7-12
Using the VERIFY Command  7-14
Customizing the iSQL*Plus Environment  7-15
SET Command Variables  7-16
iSQL*Plus Format Commands  7-17
The COLUMN Command  7-18
Using the COLUMN Command  7-19
COLUMN Format Models  7-20
Using the BREAK Command  7-21
Using the TTITLE and BTITLE Commands  7-22
Creating a Script File to Run a Report  7-23
Sample Report  7-25
Summary  7-26
Practice 7 Overview  7-27

8 Manipulating Data
Objectives  8-2
Data Manipulation Language  8-3
Adding a New Row to a Table  8-4
The INSERT Statement Syntax  8-5
Inserting New Rows  8-6
Inserting Rows with Null Values  8-7
Inserting Special Values  8-8
Inserting Specific Date Values  8-9
Creating a Script  8-10
Copying Rows from Another Table  8-11
Changing Data in a Table  8-12
The UPDATE Statement Syntax  8-13
Updating Rows in a Table  8-14
Updating Two Columns with a Subquery  8-15
Updating Rows Based on Another Table  8-16
Updating Rows: Integrity Constraint Error  8-17
Removing a Row from a Table  8-18
The DELETE Statement  8-19
Deleting Rows from a Table  8-20
Deleting Rows Based on Another Table  8-21
Deleting Rows: Integrity Constraint Error  8-22
Using a Subquery in an INSERT Statement  8-23
Using the WITH CHECK OPTION Keyword on DML Statements  8-25
Overview of the Explicit Default Feature  8-26
Using Explicit Default Values  8-27
The MERGE Statement  8-28
MERGE Statement Syntax  8-29
Merging Rows  8-30
Database Transactions  8-32
Advantages of COMMIT and ROLLBACK Statements  8-34
Controlling Transactions  8-35
Rolling Back Changes to a Marker  8-36
Implicit Transaction Processing  8-37
State of the Data Before COMMIT or ROLLBACK  8-38
State of the Data After COMMIT  8-39
Committing Data  8-40
State of the Data After ROLLBACK  8-41
Statement-Level Rollback  8-42
Read Consistency  8-43
Implementation of Read Consistency  8-44
Locking  8-45
Implicit Locking  8-46
Summary  8-47
Practice 8 Overview  8-48

9 Creating and Managing Tables
Objectives  9-2
Database Objects  9-3
Naming Rules  9-4
The CREATE TABLE Statement  9-5
Referencing Another User's Tables  9-6
The DEFAULT Option  9-7
Creating Tables  9-8
Tables in the Oracle Database  9-9
Querying the Data Dictionary  9-10
Data Types  9-11
Datetime Data Types  9-13
TIMESTAMP WITH TIME ZONE Data Type  9-15
TIMESTAMP WITH LOCAL TIME Data Type  9-16
INTERVAL YEAR TO MONTH Data Type  9-17
Creating a Table by Using a Subquery Syntax  9-18
Creating a Table by Using a Subquery  9-19
The ALTER TABLE Statement  9-20
Adding a Column  9-22
Modifying a Column  9-24
Dropping a Column  9-25
The SET UNUSED Option  9-26
Dropping a Table  9-27
Changing the Name of an Object  9-28
Truncating a Table  9-29
Adding Comments to a Table  9-30
Summary  9-31
Practice 9 Overview  9-32

10  Including Constraints
Objectives  10-2
What Are Constraints?  10-3
Constraint Guidelines  10-4
Defining Constraints  10-5
The NOT NULL Constraint  10-7
The UNIQUE Constraint  10-9
The PRIMARY KEY Constraint  10-11
The FOREIGN KEY Constraint  10-13
FOREIGN KEY Constraint Keywords  10-15
The CHECK Constraint  10-16
Adding a Constraint Syntax  10-17
Adding a Constraint  10-18
Dropping a Constraint  10-19
11 Creating Views

Objectives 11-2

Database Objects 11-3

What Is a View? 11-4

Why Use Views? 11-5

Simple Views and Complex Views 11-6

Creating a View 11-7

Retrieving Data from a View 11-10

Querying a View 11-11

Modifying a View 11-12

Creating a Complex View 11-13

Rules for Performing DML Operations on a View 11-14

Using the WITH CHECK OPTION Clause 11-17

Denying DML Operations 11-18

Removing a View 11-20

Inline Views 11-21

Top-n Analysis 11-22

Performing Top-n Analysis 11-23
12 Other Database Objects

Objectives 12-2
Database Objects 12-3
What Is a Sequence? 12-4
The CREATE SEQUENCE Statement Syntax 12-5
Creating a Sequence 12-6
Confirming Sequences 12-7
NEXTVAL and CURRVAL Pseudocolumns 12-8
Using a Sequence 12-10
Modifying a Sequence 12-12
Guidelines for Modifying a Sequence 12-13
Removing a Sequence 12-14
What Is an Index? 12-15
How Are Indexes Created? 12-16
Creating an Index 12-17
When to Create an Index 12-18
When Not to Create an Index 12-19
Confirming Indexes 12-20
Function-Based Indexes 12-21
Removing an Index 12-22
Synonyms 12-23
Creating and Removing Synonyms 12-24
Summary 12-25
Practice 12 Overview 12-26

13 Controlling User Access
Objectives 13-2
Controlling User Access 13-3
Privileges 13-4
System Privileges 13-5
Creating Users 13-6
User System Privileges 13-7
Granting System Privileges 13-8
What Is a Role? 13-9
Creating and Granting Privileges to a Role 13-10
Changing Your Password 13-11
Object Privileges 13-12
Granting Object Privileges 13-14
Using the WITH GRANT OPTION and PUBLIC Keywords 13-15
Confirming Privileges Granted 13-16
How to Revoke Object Privileges 13-17
Revoking Object Privileges 13-18
Database Links 13-19
Summary 13-21
Practice 13 Overview 13-22

14 SQL Workshop Workshop Overview
Workshop Overview 14-2
15 Using SET Operators

Objectives 15-2
The SET Operators 15-3
Tables Used in This Lesson 15-4
The UNION SET Operator 15-7
Using the UNION Operator 15-8
The UNION ALL Operator 15-10
Using the UNION ALL Operator 15-11
The INTERSECT Operator 15-12
Using the INTERSECT Operator 15-13
The MINUS Operator 15-14
SET Operator Guidelines 15-16
The Oracle Server and SET Operators 15-17
Matching the SELECT Statements 15-18
Controlling the Order of Rows 15-20
Summary 15-21
Practice 15 Overview 15-22

16 Oracle 9i Datetime Functions

Objectives 16-2
TIME ZONES 16-3
Oracle 9i Datetime Support 16-4
CURRENT_DATE 16-6
CURRENT_TIMESTAMP 16-7
LOCALTIMESTAMP 16-8
DBTIMEZONE and SESSIONTIMEZONE 16-9
17 Enhancements to the GROUP BY Clause

Objectives 17-2

Review of Group Functions 17-3

Review of the GROUP BY Clause 17-4

Review of the HAVING Clause 17-5

GROUP BY with ROLLUP and CUBE Operators 17-6

ROLLUP Operator 17-7

ROLLUP Operator Example 17-8

CUBE Operator 17-9

CUBE Operator: Example 17-10

GROUPING Function 17-11

GROUPING Function: Example 17-12

GROUPING SETS 17-13

GROUPING SETS: Example 17-15

Composite Columns 17-17

Composite Columns: Example 17-19

Concatenated Groupings 17-21
18 Advanced Subqueries

Objectives 18-2
What Is a Subquery? 18-3
Subqueries 18-4
Using a Subquery 18-5
Multiple-Column Subqueries 18-6
Column Comparisons 18-7
Pairwise Comparison Subquery 18-8
Nonpairwise Comparison Subquery 18-9
Using a Subquery in the FROM Clause 18-10
Scalar Subquery Expressions 18-11
Correlated Subqueries 18-14
Using Correlated Subqueries 18-16
Using the EXISTS Operator 18-18
Using the NOT EXISTS Operator 18-20
Correlated UPDATE 18-21
Correlated DELETE 18-24
The WITH Clause 18-26
WITH Clause: Example 18-27
Summary 18-29
Practice 18 Overview 18-31
19 Hierarchical Retrieval

Objectives 19-2
Sample Data from the EMPLOYEES Table 19-3
Natural Tree Structure 19-4
Hierarchical Queries 19-5
Walking the Tree 19-6
Walking the Tree: From the Bottom Up 19-8
Walking the Tree: From the Top Down 19-9
Ranking Rows with the LEVEL Pseudocolumn 19-10
Formatting Hierarchical Reports Using LEVEL and LPAD 19-11
Pruning Branches 19-13
Summary 19-14
Practice 19 Overview 19-15

20 Oracle 9i Extensions to DML and DDL Statements

Objectives 20-2
Review of the INSERT Statement 20-3
Review of the UPDATE Statement 20-4
Overview of Multitable INSERT Statements 20-5
Types of Multitable INSERT Statements 20-7
Multitable INSERT Statements 20-8
Unconditional INSERT ALL 20-10
Conditional INSERT ALL 20-11
Conditional FIRST INSERT 20-13
Pivoting INSERT 20-15
External Tables 20-18
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

Lesson Aim

In this lesson, you learn to create and use views. You also learn to query the relevant data dictionary object to retrieve information about views. Finally, you learn to create and use inline views, and perform top-\(n\) analysis using inline views.
## Database Objects

<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?

You can present logical subsets or combinations of data by creating views of tables. A view is a logical table based on a table or another view. A view contains no data of its own but is like a window through which data from tables can be viewed or changed. The tables on which a view is based are called base tables. The view is stored as a SELECT statement in the data dictionary.
Why Use Views?

- To restrict data access
- To make complex queries easy
- To provide data independence
- To present different views of the same data

Advantages of Views

- Views restrict access to the data because the view can display selective columns from the table.
- Views can be used to make simple queries to retrieve the results of complicated queries. For example, views can be used to query information from multiple tables without the user knowing how to write a join statement.
- Views provide data independence for ad hoc users and application programs. One view can be used to retrieve data from several tables.
- Views provide groups of users access to data according to their particular criteria.

For more information, see *Oracle9i SQL Reference*, “CREATE VIEW.”
Simple Views and Complex Views

There are two classifications for views: simple and complex. The basic difference is related to the DML (**INSERT**, **UPDATE**, and **DELETE**) operations.

- A simple view is one that:
  - Derives data from only one table
  - Contains no functions or groups of data
  - Can perform DML operations through the view
- A complex view is one that:
  - Derives data from many tables
  - Contains functions or groups of data
  - Does not always allow DML operations through the view
Creating a View

You can create a view by embedding a subquery within the `CREATE VIEW` statement.

In the syntax:
```
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

You can create a view by embedding a subquery within the `CREATE VIEW` statement.

In the syntax:
```
CREATE [OR REPLACE] [FORCE|NOFORCE] VIEW view
[(alias[, alias]...)]
AS subquery
[WITH CHECK OPTION [CONSTRAINT constraint]]
[WITH READ ONLY [CONSTRAINT constraint]];
```

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

- The subquery can contain complex `SELECT` syntax.

- OR REPLACE re-creates the view if it already exists
- FORCE creates the view regardless of whether or not the base tables exist
- NOFORCE creates the view only if the base tables exist (This is the default.)
- view is the name of the view
- alias specifies names for the expressions selected by the view’s query (The number of aliases must match the number of expressions selected by the view.)
- subquery is a complete `SELECT` statement (You can use aliases for the columns in the `SELECT` list.)
- WITH CHECK OPTION specifies that only rows accessible to the view can be inserted or updated
- constraint is the name assigned to the `CHECK OPTION` constraint
- WITH READ ONLY ensures that no DML operations can be performed on this view
Creating a View

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

```sql
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.

```sql
DESCRIBE empvu80
```

Creating a View (continued)

The example in the slide creates a view that contains the employee number, last name, and salary for each employee in department 80.

You can display the structure of the view by using the iSQL*Plus DESCRIBE command.

<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>LAST_NAME</td>
<td>NOT NULL</td>
<td>VARCHAR2(25)</td>
</tr>
<tr>
<td>SALARY</td>
<td></td>
<td>NUMBER(8,2)</td>
</tr>
</tbody>
</table>

Guidelines for creating a view:

• The subquery that defines a view can contain complex SELECT syntax, including joins, groups, and subqueries.
• The subquery that defines the view cannot contain an ORDER BY clause. The ORDER BY clause is specified when you retrieve data from the view.
• If you do not specify a constraint name for a view created with the WITH CHECK OPTION, the system assigns a default name in the format SYS_Cn.
• You can use the OR REPLACE option to change the definition of the view without dropping and recreating it or regranting object privileges previously granted on it.
Creating a View

• Create a view by using column aliases in the subquery.

CREATE VIEW salvu50
AS SELECT employee_id ID_NUMBER, last_name NAME, salary*12 ANN_SALARY
FROM employees
WHERE department_id = 50;
View created.

• Select the columns from this view by the given alias names.

Creating a View (continued)

You can control the column names by including column aliases within the subquery.
The example in the slide creates a view containing the employee number (EMPLOYEE_ID) with the alias
ID_NUMBER, name (LAST_NAME) with the alias NAME, and annual salary (SALARY) with the alias
ANN_SALARY for every employee in department 50.
As an alternative, you can use an alias after the CREATE statement and prior to the SELECT subquery. The
number of aliases listed must match the number of expressions selected in the subquery.

CREATE VIEW salvu50 (ID_NUMBER, NAME, ANN_SALARY)
AS SELECT employee_id, last_name, salary*12
FROM employees
WHERE department_id = 50;
View created.
## Retrieving Data from a View

You can retrieve data from a view as you would from any table. You can display either the contents of the entire view or just specific rows and columns.

### SQL Example

```sql
SELECT *
FROM salvu50;
```

<table>
<thead>
<tr>
<th>ID_NUMER</th>
<th>NAME</th>
<th>ANN_SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>124</td>
<td>Murugas</td>
<td>68900</td>
</tr>
<tr>
<td>141</td>
<td>Rajia</td>
<td>42000</td>
</tr>
<tr>
<td>142</td>
<td>Damaas</td>
<td>37200</td>
</tr>
<tr>
<td>143</td>
<td>Nacoo</td>
<td>31200</td>
</tr>
<tr>
<td>144</td>
<td>Vargac</td>
<td>36300</td>
</tr>
</tbody>
</table>
Views in the Data Dictionary

Once your view has been created, you can query the data dictionary view called USER_VIEWS to see the name of the view and the view definition. The text of the SELECT statement that constitutes your view is stored in a LONG column.

Data Access Using Views

When you access data using a view, the Oracle Server performs the following operations:

1. It retrieves the view definition from the data dictionary table USER_VIEWS.
2. It checks access privileges for the view base table.
3. It converts the view query into an equivalent operation on the underlying base table or tables. In other words, data is retrieved from, or an update is made to, the base tables.
Modifying a View

• Modify the EMPVU80 view by using CREATE OR REPLACE VIEW clause. Add an alias for each column name.

```
CREATE OR REPLACE VIEW empvu80
  (id_number, name, sal, department_id)
AS SELECT employee_id, first_name || ' ' || last_name, salary, department_id
    FROM employees
WHERE department_id = 80;
View created.
```

• Column aliases in the CREATE VIEW clause are listed in the same order as the columns in the subquery.

Modifying a View
With the OR REPLACE option, a view can be created even if one exists with this name already, thus replacing the old version of the view for its owner. This means that the view can be altered without dropping, re-creating, and regranting object privileges.

Note: When assigning column aliases in the CREATE VIEW clause, remember that the aliases are listed in the same order as the columns in the subquery.
Creating a Complex View

Create a complex view that contains group functions to display values from two tables.

```
CREATE VIEW dept_sum_vu
    (name, minsal, maxsal, avgsal)
AS SELECT d.department_name, MIN(e.salary),
     MAX(e.salary), AVG(e.salary)
FROM      employees e, departments d
WHERE     e.department_id = d.department_id
GROUP BY  d.department_name;
View created.
```

Creating a Complex View

The example in the slide creates a complex view of department names, minimum salaries, maximum salaries, and average salaries by department. Note that alternative names have been specified for the view. This is a requirement if any column of the view is derived from a function or an expression.

You can view the structure of the view by using the iSQL*Plus DESCRIBE command. Display the contents of the view by issuing a SELECT statement.

```
SELECT  *
FROM    dept_sum_vu;
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>MINSAL</th>
<th>MAXSAL</th>
<th>AVGSAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accounting</td>
<td>8300</td>
<td>12000</td>
<td>10150</td>
</tr>
<tr>
<td>Administration</td>
<td>4400</td>
<td>4400</td>
<td>4400</td>
</tr>
<tr>
<td>Executive</td>
<td>17000</td>
<td>24000</td>
<td>19333.3333</td>
</tr>
<tr>
<td>IT</td>
<td>4200</td>
<td>9000</td>
<td>6400</td>
</tr>
<tr>
<td>Marketing</td>
<td>6000</td>
<td>13000</td>
<td>9500</td>
</tr>
<tr>
<td>Sales</td>
<td>8600</td>
<td>11000</td>
<td>10033.3333</td>
</tr>
<tr>
<td>Shipping</td>
<td>2500</td>
<td>5800</td>
<td>3500</td>
</tr>
</tbody>
</table>

7 rows selected.
Rules for Performing DML Operations on a View

- You can perform DML operations on simple views.
- You cannot remove a row if the view contains the following:
  - Group functions
  - A GROUP BY clause
  - The DISTINCT keyword
  - The pseudocolumn ROWNUM keyword

Performing DML Operations on a View

You can perform DML operations on data through a view if those operations follow certain rules. You can remove a row from a view unless it contains any of the following:

- Group functions
- A GROUP BY clause
- The DISTINCT keyword
- The pseudocolumn ROWNUM keyword
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

Performing DML Operations on a View (continued)

You can modify data through a view unless it contains any of the conditions mentioned in the previous slide or columns defined by expressions: for example, SALARY * 12.
Rules for Performing DML Operations on a View

You cannot add data through a view if the view includes:

- Group functions
- A GROUP BY clause
- The DISTINCT keyword
- The pseudocolumn ROWNUM keyword
- Columns defined by expressions
- NOT NULL columns in the base tables that are not selected by the view

Performing DML Operations on a View (continued)

You can add data through a view unless it contains any of the items listed in the slide or there are NOT NULL columns, without default values, in the base table that are not selected by the view. All required values must be present in the view. Remember that you are adding values directly into the underlying table through the view.

For more information, see Oracle9i SQL Reference, “CREATE 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.

```sql
UPDATE empvu20
SET    department_id = 10
WHERE employee_id = 201;
```

*ERROR at line 1: ORA-01402: view WITH CHECK OPTION where-clause violation*

**Note:** No rows are updated because if the department number were to change to 10, the view would no longer be able to see that employee. Therefore, with the **WITH CHECK OPTION** clause, the view can see only employees in department 20 and does not allow the department number for those employees to be changed through the view.
Denying DML Operations

- You can ensure that no DML operations occur by adding the `WITH READ ONLY` option to your view definition.
- Any attempt to perform a DML on any row in the view results in an Oracle server error.

Denying DML Operations

You can ensure that no DML operations occur on your view by creating it with the `WITH READ ONLY` option. The example in the slide modifies the `EMPVU10` view to prevent any DML operations on the view.
Denying DML Operations

Any attempts to remove a row from a view with a read-only constraint results in an error.

```sql
DELETE FROM empvu10
    WHERE employee_number = 200;
```

```sql
ERROR at line 1:
ORA-01752: cannot delete from view without exactly one key-preserved table
```

Any attempts to insert a row or modify a row using the view with a read-only constraint results in the following Oracle Server error:

```sql
01733: virtual column not allowed here.
```
Removing a View

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

```sql
DROP VIEW view;

DROP VIEW empvu80;
View dropped.
```

Removing a View

You use the `DROP VIEW` statement to remove a view. The statement removes the view definition from the database. Dropping views has no effect on the tables on which the view was based. Views or other applications based on deleted views become invalid. Only the creator or a user with the `DROP ANY VIEW` privilege can remove a view.

In the syntax:

```sql
view
```

is the name of the view
Inline Views

- An inline view is a subquery with an alias (or correlation name) that you can use within a SQL statement.
- A named subquery in the FROM clause of the main query is an example of an inline view.
- An inline view is not a schema object.

```sql
SELECT a.last_name, a.salary, a.department_id, b.maxsal
FROM employees a,
    (SELECT department_id, max(salary) maxsal
     FROM employees
     GROUP BY department_id) b
WHERE a.department_id = b.department_id
    AND a.salary < b.maxsal;
```

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>SALARY</th>
<th>DEPARTMENT_ID</th>
<th>MAXSAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fay</td>
<td>6000</td>
<td>20</td>
<td>13000</td>
</tr>
<tr>
<td>Rajs</td>
<td>3500</td>
<td>50</td>
<td>5800</td>
</tr>
<tr>
<td>Davies</td>
<td>3100</td>
<td>50</td>
<td>5800</td>
</tr>
</tbody>
</table>

| Gietz     |        |               |        |

12 rows selected.
Top-$n$ Analysis

- Top-$n$ queries ask for the $n$ largest or smallest values of a column. For example:
  - What are the ten best selling products?
  - What are the ten worst selling products?
- Both largest values and smallest values sets are considered top-$n$ queries.

Top-$n$ Analysis

Top-$n$ queries are useful in scenarios where the need is to display only the $n$ top-most or the $n$ bottommost records from a table based on a condition. This result set can be used for further analysis. For example using top-$n$ analysis you can perform the following types of queries:

- The top three earners in the company
- The four most recent recruits in the company
- The top two sales representatives who have sold the maximum number of products
- The top three products that have had the maximum sales in the last six months
Performing Top-$n$ Analysis

The high-level structure of a top-$n$ analysis query is:

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

Performing Top-$n$ Analysis

Top-$n$ queries use a consistent nested query structure with the elements described below:

- A subquery or an inline view to generate the sorted list of data. The subquery or the inline view includes the `ORDER BY` clause to ensure that the ranking is in the desired order. For results retrieving the largest values, a `DESC` parameter is needed.

- An outer query to limit the number of rows in the final result set. The outer query includes the following components:
  - The `ROWNUM` pseudocolumn, which assigns a sequential value starting with 1 to each of the rows returned from the subquery.
  - A `WHERE` clause, which specifies the $n$ rows to be returned. The outer `WHERE` clause must use a $<$ or $\leq$ operator.
Example of Top-$n$ Analysis

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

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

<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 Heian</td>
<td>17000</td>
</tr>
</tbody>
</table>

Example of Top-$n$ Analysis

The example in the slide illustrates how to display the names and salaries of the top three earners from the EMPLOYEES table. The subquery returns the details of all employee names and salaries from the EMPLOYEES table, sorted in the descending order of the salaries. The `WHERE ROWNUM < 3` clause of the main query ensures that only the first three records from this result set are displayed.

Here is another example of top-$n$ analysis that uses an inline view. The example below uses the inline view `E` to display the four most senior employees in the company.

```
SELECT ROWNUM as SENIOR, E.last_name, E.hire_date
FROM (SELECT last_name, hire_date FROM employees
       ORDER BY hire_date) E
WHERE rownum <= 4;
```

<table>
<thead>
<tr>
<th>SENIOR</th>
<th>LAST_NAME</th>
<th>HIRE_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>King</td>
<td>17-JUN-87</td>
</tr>
<tr>
<td>2</td>
<td>Whalen</td>
<td>17-SEP-87</td>
</tr>
<tr>
<td>3</td>
<td>Kochhar</td>
<td>21-SEP-89</td>
</tr>
<tr>
<td>4</td>
<td>Hunkold</td>
<td>03-JAN-90</td>
</tr>
</tbody>
</table>
Summary

In this lesson you should have learned that a view is derived from data in other tables or other 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

What Is a View?

A view is based on a table or another view and acts as a window through which data on tables can be viewed or changed. A view does not contain data. The definition of the view is stored in the data dictionary. You can see the definition of the view in the \texttt{USER\_VIEWS} data dictionary table.

Advantages of Views

• Restrict database access
• Simplify queries
• Provide data independence
• Provide multiple views of the same data
• Can be removed without affecting the underlying data

View Options

• Can be a simple view, based on one table
• Can be a complex view based on more than one table or can contain groups of functions
• Can replace other views with the same name
• Can contain a check constraint
• Can be read-only
Practice 11 Overview

This practice covers the following topics:

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

Practice 11 Overview

In this practice, you create simple and complex views and attempt to perform DML statements on the views.
Practice 11

1. Create a view called EMPLOYEES_VU based on the employee numbers, employee names, and department numbers from the EMPLOYEES table. Change the heading for the employee name to EMPLOYEE.

2. Display the contents of the EMPLOYEES_VU view.

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>EMPLOYEE</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>Ernst</td>
<td>60</td>
</tr>
<tr>
<td>107</td>
<td>Lorentz</td>
<td>60</td>
</tr>
<tr>
<td>206</td>
<td>Gietz</td>
<td>10</td>
</tr>
</tbody>
</table>

20 rows selected.

3. Select the view name and text from the USER_VIEWS data dictionary view.

   Note: Another view already exists. The EMP_DETAILS_VIEW was created as part of your schema.

   Note: To see more contents of a LONG column, use the iSQL*Plus command SET LONG n, where n is the value of the number of characters of the LONG column that you want to see.

<table>
<thead>
<tr>
<th>VIEW_NAME</th>
<th>TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPLOYEES_VU</td>
<td>SELECT employee_id, last_name employee, department_id FROM employees</td>
</tr>
<tr>
<td>EMP_DETAILS_VIEW</td>
<td>SELECT e.employee_id, e.job_id, e.manager_id, e.department_id, d.location_id, l.country_id, e.first_name, e.last_name, e.salary, e.commission, n_pct, d.department_name, j.job_title, l.city, l.state_province, c.country_name, r.region_name FROM employees e, departments d, jobs j, locations l, countries c, regions r WHERE e.department_id = d.department_id AND d.location_id = l.location_id AND l.country_id = c.country_id AND r.region_id = r.region_id AND j.job_id = e.job_id WITH READ ONLY</td>
</tr>
</tbody>
</table>

4. Using your EMPLOYEES_VU view, enter a query to display all employee names and department numbers.

<table>
<thead>
<tr>
<th>EMPLOYEE</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td>90</td>
</tr>
<tr>
<td>Kochhar</td>
<td>90</td>
</tr>
<tr>
<td>De Haan</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Gietz</td>
<td>10</td>
</tr>
</tbody>
</table>

20 rows selected.
5. Create a view named `DEPT50` that contains the employee numbers, employee last names, and department numbers for all employees in department 50. Label the view columns `EMPNO`, `EMPLOYEE`, and `DEPTNO`. Do not allow an employee to be reassigned to another department through the view.

6. Display the structure and contents of the `DEPT50` view.

<table>
<thead>
<tr>
<th>Name</th>
<th>Null?</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPNO</td>
<td>NOT NULL</td>
<td>NUMBER(6)</td>
</tr>
<tr>
<td>EMPLOYEE</td>
<td>NOT NULL</td>
<td>VARCHAR2(25)</td>
</tr>
<tr>
<td>DEPTNO</td>
<td></td>
<td>NUMBER(4)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>EMPLOYEE</th>
<th>DEPTNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>124</td>
<td>Mourgos</td>
<td>50</td>
</tr>
<tr>
<td>141</td>
<td>Rajs</td>
<td>50</td>
</tr>
<tr>
<td>142</td>
<td>Davies</td>
<td>50</td>
</tr>
<tr>
<td>143</td>
<td>Matos</td>
<td>50</td>
</tr>
<tr>
<td>144</td>
<td>Vargas</td>
<td>50</td>
</tr>
</tbody>
</table>

7. Attempt to reassign Matos to department 80.

If you have time, complete the following exercise:

8. Create a view called `SALARY_VU` based on the employee last names, department names, salaries, and salary grades for all employees. Use the `EMPLOYEES`, `DEPARTMENTS`, and `JOB_GRADES` tables. Label the columns `Employee`, `Department`, `Salary`, and `Grade`, respectively.
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

Lesson Aim

In this lesson, you learn how to create and maintain some of the other commonly used database objects. These objects include sequences, indexes, and synonyms.
Many applications require the use of unique numbers as primary key values. You can either build code into the application to handle this requirement or use a sequence to generate unique numbers.

If you want to improve the performance of some queries, you should consider creating an index. You can also use indexes to enforce uniqueness on a column or a collection of columns.

You can provide alternative names for objects by using synonyms.
What Is a Sequence?

A sequence:
- Automatically generates unique numbers
- Is a sharable object
- Is typically used to create a primary key value
- Replaces application code
- Speeds up the efficiency of accessing sequence values when cached in memory

A sequence is a user created database object that can be shared by multiple users to generate unique integers.

A typical usage for sequences is to create a primary key value, which must be unique for each row. The sequence is generated and incremented (or decremented) by an internal Oracle routine. This can be a time-saving object because it can reduce the amount of application code needed to write a sequence-generating routine.

Sequence numbers are stored and generated independently of tables. Therefore, the same sequence can be used for multiple tables.
The `CREATE SEQUENCE` Statement Syntax

Define a sequence to generate sequential numbers automatically.

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

Creating a Sequence

Automatically generate sequential numbers by using the `CREATE SEQUENCE` statement.

In the syntax:

- `sequence` is the name of the sequence generator
- `INCREMENT BY n` specifies the interval between sequence numbers where \( n \) is an integer (If this clause is omitted, the sequence increments by 1.)
- `START WITH n` specifies the first sequence number to be generated (If this clause is omitted, the sequence starts with 1.)
- `MAXVALUE n` specifies the maximum value the sequence can generate
- `NOMAXVALUE` specifies a maximum value of \( 10^{27} \) for an ascending sequence and \( -1 \) for a descending sequence (This is the default option.)
- `MINVALUE n` specifies the minimum sequence value
- `NOMINVALUE` specifies a minimum value of 1 for an ascending sequence and \( -10^{26} \) for a descending sequence (This is the default option.)
- `CYCLE | NOCYCLE` specifies whether the sequence continues to generate values after reaching its maximum or minimum value (NOCYCLE is the default option.)
- `CACHE n | NOCACHE` specifies how many values the Oracle Server preallocates and keep in memory (By default, the Oracle Server caches 20 values.)
Creating a Sequence

- Create a sequence named `DEPT_DEPTID_SEQ` to be used for the primary key of the `DEPARTMENTS` table.
- Do not use the `CYCLE` option.

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

Sequence created.
```

Creating a Sequence (continued)

The example in the slide creates a sequence named `DEPT_DEPTID_SEQ` to be used for the `DEPARTMENT_ID` column of the `DEPARTMENTS` table. The sequence starts at 120, does not allow caching, and does not cycle.

Do not use the `CYCLE` option if the sequence is used to generate primary key values, unless you have a reliable mechanism that purges old rows faster than the sequence cycles.

For more information, see Oracle9i SQL Reference, “CREATE SEQUENCE.”

Note: The sequence is not tied to a table. Generally, you should name the sequence after its intended use; however the sequence can be used anywhere, regardless of its name.
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.

Confirming Sequences

Once you have created your sequence, it is documented in the data dictionary. Because a sequence is a database object, you can identify it in the USER_OBJECTS data dictionary table.

You can also confirm the settings of the sequence by selecting from the USER_SEQUENCES data dictionary view.

<table>
<thead>
<tr>
<th>SEQUENCE_NAME</th>
<th>MIN_VALUE</th>
<th>MAX_VALUE</th>
<th>INCREMENT_BY</th>
<th>LAST_NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEPARTMENTS_SEQ</td>
<td>1</td>
<td>9990</td>
<td>10</td>
<td>280</td>
</tr>
<tr>
<td>DEPT_DEPTID_SEQ</td>
<td>1</td>
<td>9999</td>
<td>10</td>
<td>120</td>
</tr>
<tr>
<td>EMPLOYEES_SEQ</td>
<td>1 1.0000E+27</td>
<td>1</td>
<td>207</td>
<td></td>
</tr>
<tr>
<td>LOCATIONS_SEQ</td>
<td>1</td>
<td>9900</td>
<td>100</td>
<td>3300</td>
</tr>
</tbody>
</table>
NEXTVAL and CURRVAL Pseudocolumns

- **NEXTVAL** returns the next available sequence value.
  It returns a unique value every time it is referenced, even for different users.
- **CURRVAL** obtains the current sequence value.
- **NEXTVAL** must be issued for that sequence before CURRVAL contains a value.

Using a Sequence

After you create your sequence, it generates sequential numbers for use in your tables. Reference the sequence values by using the NEXTVAL and CURRVAL pseudocolumns.

**NEXTVAL and CURRVAL Pseudocolumns**

The NEXTVAL pseudocolumn is used to extract successive sequence numbers from a specified sequence. You must qualify NEXTVAL with the sequence name. When you reference `sequence.NEXTVAL`, a new sequence number is generated and the current sequence number is placed in CURRVAL.

The CURRVAL pseudocolumn is used to refer to a sequence number that the current user has just generated. NEXTVAL must be used to generate a sequence number in the current user’s session before CURRVAL can be referenced. You must qualify CURRVAL with the sequence name. When `sequence.CURRVAL` is referenced, the last value returned to that user’s process is displayed.
Rules for Using NEXTVAL and CURRVAL

You can use NEXTVAL and CURRVAL in the following contexts:

- The SELECT list of a SELECT statement that is not part of a subquery
- The SELECT list of a subquery in an INSERT statement
- The VALUES clause of an INSERT statement
- The SET clause of an UPDATE statement

You cannot use NEXTVAL and CURRVAL in the following contexts:

- The SELECT list of a view
- A SELECT statement with the DISTINCT keyword
- A SELECT statement with GROUP BY, HAVING, or ORDER BY clauses
- A subquery in a SELECT, DELETE, or UPDATE statement
- The DEFAULT expression in a CREATE TABLE or ALTER TABLE statement

For more information, see *Oracle9i SQL Reference*, “Pseudocolumns” and “CREATE SEQUENCE section.”
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

The example in the slide inserts a new department in the DEPARTMENTS table. It uses the DEPT_DEPTID_SEQ sequence for generating a new department number as follows:

```sql
SELECT dept_deptid_seq.CURRVAL
FROM   dual;
```

CURRVAL
---------
120

Suppose now you want to hire employees to staff the new department. The INSERT statement to be executed for all new employees can include the following code:

```sql
INSERT INTO employees (employee_id, department_id, ...)
VALUES (employees_seq.NEXTVAL, dept_deptid_seq.CURRVAL, ...);
```

Note: The preceding example assumes that a sequence called EMPLOYEE_SEQ has already been created for generating new employee numbers.
Using a Sequence

- Caching sequence values in memory gives faster access to those values.
- Gaps in sequence values can occur when:
  - A rollback occurs
  - The system crashes
  - A sequence is used in another table
- If the sequence was created with NOCACHE, view the next available value, by querying the USER_SEQUENCES table.

Caching Sequence Values
Cache sequences in memory to provide faster access to those sequence values. The cache is populated the first time you refer to the sequence. Each request for the next sequence value is retrieved from the cached sequence. After the last sequence value is used, the next request for the sequence pulls another cache of sequences into memory.

Gaps in the Sequence
Although sequence generators issue sequential numbers without gaps, this action occurs independent of a commit or rollback. Therefore, if you roll back a statement containing a sequence, the number is lost.

Another event that can cause gaps in the sequence is a system crash. If the sequence caches values in the memory, then those values are lost if the system crashes.

Because sequences are not tied directly to tables, the same sequence can be used for multiple tables. If you do so, each table can contain gaps in the sequential numbers.

Viewing the Next Available Sequence Value without Incrementing It
If the sequence was created with NOCACHE, it is possible to view the next available sequence value without incrementing it by querying the USER_SEQUENCES table.
Modifying a Sequence

Change the increment value, maximum value, minimum value, cycle option, or cache option.

```
ALTER SEQUENCE dept_deptid_seq
    INCREMENT BY 20
    MAXVALUE 999999
    NOCACHE
    NOCYCLE;
```

Sequence altered.

Altering a Sequence

If you reach the MAXVALUE limit for your sequence, no additional values from the sequence are allocated and you will receive an error indicating that the sequence exceeds the MAXVALUE. To continue to use the sequence, you can modify it by using the ALTER SEQUENCE statement.

**Syntax**

```
ALTER  SEQUENCE   sequence
    [INCREMENT BY  n]
    [{MAXVALUE  n  |  NOMAXVALUE}]
    [{MINVALUE  n  |  NOMINVALUE}]
    [{CYCLE  |  NOCYCLE}]
    [{CACHE  n  |  NOCACHE}];
```

In the syntax:

- `sequence` is the name of the sequence generator

For more information, see Oracle9i SQL Reference, “ALTER SEQUENCE.”
Guidelines for Modifying a Sequence

- You must be the owner or have the **ALTER** privilege for the sequence.
- Only future sequence numbers are affected.
- The sequence must be dropped and re-created to restart the sequence at a different number.
- Some validation is performed.

Guidelines for Modifying Sequences

- You must be the owner or have the **ALTER** privilege for the sequence in order to modify it.
- Only future sequence numbers are affected by the **ALTER SEQUENCE** statement.
- The **START WITH** option cannot be changed using **ALTER SEQUENCE**. The sequence must be dropped and re-created in order to restart the sequence at a different number.
- Some validation is performed. For example, a new **MAXVALUE** that is less than the current sequence number cannot be imposed.

```
ALTER SEQUENCE dept_deptid_seq
  INCREMENT BY 20
  MAXVALUE 90
  NOCACHE
  NOCYCLE;

ALTER SEQUENCE dept_deptid_seq

*  
ERROR at line 1: 
ORA-04009: MAXVALUE cannot be made to be less than the current value
```
Removing a Sequence

- Remove a sequence from the data dictionary by using the `DROP SEQUENCE` statement.
- Once removed, the sequence can no longer be referenced.

```sql
DROP SEQUENCE dept_deptid_seq;
Sequence dropped.
```

Removing a Sequence

To remove a sequence from the data dictionary, use the `DROP SEQUENCE` statement. You must be the owner of the sequence or have the `DROP ANY SEQUENCE` privilege to remove it.

**Syntax**

```
DROP SEQUENCE sequence;
```

In the syntax:

- `sequence` is the name of the sequence generator

For more information, see *Oracle9i SQL Reference*, “DROP SEQUENCE.”
What Is an Index?

An index:

- Is a schema object
- Is used by the Oracle Server to speed up the retrieval of rows by using a pointer
- Can reduce disk I/O by using a rapid path access method to locate data quickly
- Is independent of the table it indexes
- Is used and maintained automatically by the Oracle Server

Indexes

An Oracle Server index is a schema object that can speed up the retrieval of rows by using a pointer. Indexes can be created explicitly or automatically. If you do not have an index on the column, then a full table scan occurs.

An index provides direct and fast access to rows in a table. Its purpose is to reduce the necessity of disk I/O by using an indexed path to locate data quickly. The index is used and maintained automatically by the Oracle Server. Once an index is created, no direct activity is required by the user.

Indexes are logically and physically independent of the table they index. This means that they can be created or dropped at any time and have no effect on the base tables or other indexes.

Note: When you drop a table, corresponding indexes are also dropped.

For more information, see Oracle9i Concepts, “Schema Objects” section, “Indexes” topic.
How Are Indexes Created?

- Automatically: A unique index is created automatically when you define a PRIMARY KEY or UNIQUE constraint in a table definition.
- Manually: Users can create nonunique indexes on columns to speed up access to the rows.

Types of Indexes

Two types of indexes can be created. One type is a unique index: the Oracle Server automatically creates this index when you define a column in a table to have a PRIMARY KEY or a UNIQUE key constraint. The name of the index is the name given to the constraint.

The other type of index is a nonunique index, which a user can create. For example, you can create a FOREIGN KEY column index for a join in a query to improve retrieval speed.

Note: You can manually create a unique index, but it is recommended that you create a unique constraint, which implicitly creates a unique index.
Creating an Index

- Create an index on one or more columns.

```
CREATE INDEX index
ON table (column[, column]...);
```

- Improve the speed of query access to the LAST_NAME column in the EMPLOYEES table.

```
CREATE INDEX emp_last_name_idx
ON employees(last_name);
Index created.
```

Creating an Index

Create an index on one or more columns by issuing the `CREATE INDEX` statement.

In the syntax:
- `index` is the name of the index
- `table` is the name of the table
- `column` is the name of the column in the table to be indexed

For more information, see `Oracle9i SQL Reference`, “CREATE INDEX.”
More Is Not Always Better

More indexes on a table does not mean faster queries. Each DML operation that is committed on a table with indexes means that the indexes must be updated. The more indexes you have associated with a table, the more effort the Oracle server must make to update all the indexes after a DML operation.

When to Create an Index

Therefore, you should create indexes only if:

• The column contains a wide range of values
• The column contains a large number of null values
• One or more columns are frequently used together in a WHERE clause or a join condition
• The table is large and most queries are expected to retrieve less than 2 to 4% of the rows

Remember that if you want to enforce uniqueness, you should define a unique constraint in the table definition. Then a unique index is created automatically.
When Not to Create an Index

It is usually not worth creating an index if:

• The table is small
• The columns are not often used as a condition in the query
• Most queries are expected to retrieve more than 2 to 4% 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';
```

Confirming Indexes

Confirm the existence of indexes from the USER_INDEXES data dictionary view. You can also check the columns involved in an index by querying the USER_IND_COLUMNS view.

The example on the slide displays all the previously created indexes, with the names of the affected column, and the index’s uniqueness, on the EMPLOYEES table.

<table>
<thead>
<tr>
<th>INDEX_NAME</th>
<th>COLUMN_NAME</th>
<th>COL_POS</th>
<th>UNIQUENESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMP_EMAIL_UK</td>
<td>EMAIL</td>
<td>1</td>
<td>UNIQUE</td>
</tr>
<tr>
<td>EMP_EMP_ID_PK</td>
<td>EMPLOYEE_ID</td>
<td>1</td>
<td>UNIQUE</td>
</tr>
<tr>
<td>EMP_DEPARTMENT_IX</td>
<td>DEPARTMENT_ID</td>
<td>1</td>
<td>NONUNIQUE</td>
</tr>
<tr>
<td>EMP_JOB_IX</td>
<td>JOB_ID</td>
<td>1</td>
<td>NONUNIQUE</td>
</tr>
<tr>
<td>EMP_MANAGER_IX</td>
<td>MANAGER_ID</td>
<td>1</td>
<td>NONUNIQUE</td>
</tr>
<tr>
<td>EMP_NAME_IX</td>
<td>LAST_NAME</td>
<td>1</td>
<td>NONUNIQUE</td>
</tr>
<tr>
<td>EMP_NAME_IX</td>
<td>FIRST_NAME</td>
<td>2</td>
<td>NONUNIQUE</td>
</tr>
<tr>
<td>EMP_LAST_NAME_IDX</td>
<td>LAST_NAME</td>
<td>1</td>
<td>NONUNIQUE</td>
</tr>
</tbody>
</table>

8 rows selected.
Function-Based Indexes

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

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

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

Function-Based Index

Function-based indexes defined with the `UPPER(column_name)` or `LOWER(column_name)` keywords allow case-insensitive searches. For example, the following index:

```sql
CREATE INDEX upper_last_name_idx ON employees (UPPER(last_name));
```

Facilitates processing queries such as:

```sql
SELECT * FROM employees WHERE UPPER(last_name) = 'KING';
```

To ensure that the Oracle Server uses the index rather than performing a full table scan, be sure that the value of the function is not null in subsequent queries. For example, the following statement is guaranteed to use the index, but without the `WHERE` clause the Oracle Server may perform a full table scan:

```sql
SELECT * FROM employees
WHERE UPPER(last_name) IS NOT NULL
ORDER BY UPPER(last_name);
```

The Oracle Server treats indexes with columns marked `DESC` as function-based indexes. The columns marked `DESC` are sorted in descending order.
Removing an Index

You cannot modify indexes. To change an index, you must drop it and then re-create it. Remove an index definition from the data dictionary by issuing the DROP INDEX statement. To drop an index, you must be the owner of the index or have the DROP ANY INDEX privilege.

In the syntax:

\[
\text{DROP INDEX index;} \\
\text{DROP INDEX upper_last_name_idx;} \\
\text{DROP INDEX upper_last_name_idx;} \text{ Index dropped.}
\]

Note: If you drop a table, indexes and constraints are automatically dropped, but views and sequences remain.
Synonyms

Simplify access to objects by creating a synonym (another name for an object). With synonyms, you can:
• Ease referring to a table owned by another user
• Shorten lengthy object names

CREATE [PUBLIC] SYNONYM synonym
FOR object;

Creating a Synonym for an Object

To refer to a table owned by another user, you need to prefix the table name with the name of the user who created it followed by a period. Creating a synonym eliminates the need to qualify the object name with the schema and provides you with an alternative name for a table, view, sequence, procedure, or other objects. This method can be especially useful with lengthy object names, such as views.

In the syntax:

PUBLIC creates a synonym accessible to all users
synonym is the name of the synonym to be created
object identifies the object for which the synonym is created

Guidelines

• The object cannot be contained in a package.
• A private synonym name must be distinct from all other objects owned by the same user.

For more information, see Oracle9i SQL Reference, “CREATE SYNONYM.”
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.
```

Creating a Synonym for an Object (continued)

The example in the slide creates a synonym for the DEPT_SUM_VU view for quicker reference.

The database administrator can create a public synonym accessible to all users. The following example creates a public synonym named DEPT for Alice’s DEPARTMENTS table:

```sql
CREATE PUBLIC SYNONYM dept
FOR alice.departments;
Synonym created.
```

Removing a Synonym

To drop a synonym, use the DROP SYNONYM statement. Only the database administrator can drop a public synonym.

```sql
DROP PUBLIC SYNONYM dept;
Synonym dropped.
```

For more information, see Oracle9i SQL Reference, “DROP SYNONYM.”
Summary

In this lesson, you should have learned how to:

• Generate sequence numbers automatically by using a sequence generator

• View sequence information in the USER_SEQUENCES data dictionary table

• Create indexes to improve query retrieval speed

• View index information in the USER_INDEXES dictionary table

• Use synonyms to provide alternative names for objects

Sequences

The sequence generator can be used to automatically generate sequence numbers for rows in tables. This can save time and can reduce the amount of application code needed.

A sequence is a database object that can be shared with other users. Information about the sequence can be found in the USER_SEQUENCES table of the data dictionary.

To use a sequence, reference it with either the NEXTVAL or the CURRVAL pseudocolumns.

• Retrieve the next number in the sequence by referencing sequence.NEXTVAL.
• Return the current available number by referencing sequence.CURRVAL.

Indexes

Indexes are used to improve query retrieval speed. Users can view the definitions of the indexes in the USER_INDEXES data dictionary view. An index can be dropped by the creator, or a user with the DROP ANY INDEX privilege, by using the DROP INDEX statement.

Synonyms

Database administrators can create public synonyms and users can create private synonyms for convenience, by using the CREATE SYNONYM statement. Synonyms permit short names or alternative names for objects. Remove synonyms by using the DROP SYNONYM statement.
Practice 12 Overview

This practice covers the following topics:

- Creating sequences
- Using sequences
- Creating nonunique indexes
- Displaying data dictionary information about sequences and indexes
- Dropping indexes

Practice 12 Overview
In this practice, you create a sequence to be used when populating your table. You also create implicit and explicit indexes.
Practice 12

1. Create a sequence to be used with the primary key column of the DEPT table. The sequence should start at 200 and have a maximum value of 1000. Have your sequence increment by ten numbers. Name the sequence DEPT_ID_SEQ.

2. Write a query in a script to display the following information about your sequences: sequence name, maximum value, increment size, and last number. Name the script lab12_2.sql. Run the statement in your script.

<table>
<thead>
<tr>
<th>SEQUENCE_NAME</th>
<th>MAX_VALUE</th>
<th>INCREMENT_BY</th>
<th>LAST_NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEPARTMENTS_SEQ</td>
<td>9990</td>
<td>10</td>
<td>280</td>
</tr>
<tr>
<td>DEPT_ID_SEQ</td>
<td>1000</td>
<td>10</td>
<td>200</td>
</tr>
<tr>
<td>EMPLOYEES_SEQ</td>
<td>1.0000E+27</td>
<td>1</td>
<td>207</td>
</tr>
<tr>
<td>LOCATIONS_SEQ</td>
<td>9900</td>
<td>100</td>
<td>3300</td>
</tr>
</tbody>
</table>

3. Write a script to insert two rows into the DEPT table. Name your script lab12_3.sql. Be sure to use the sequence that you created for the ID column. Add two departments named Education and Administration. Confirm your additions. Run the commands in your script.

4. Create a nonunique index on the foreign key column (DEPT_ID) in the EMP table.

5. Display the indexes and uniqueness that exist in the data dictionary for the EMP table. Save the statement into a script named lab12_5.sql.

<table>
<thead>
<tr>
<th>INDEX_NAME</th>
<th>TABLE_NAME</th>
<th>UNIQUENES</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMP_DEPT_ID_IDX</td>
<td>EMP</td>
<td>NONUNIQUE</td>
</tr>
<tr>
<td>EMP_ID_PK</td>
<td>EMP</td>
<td>UNIQUE</td>
</tr>
</tbody>
</table>
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

Lesson Aim

In this lesson, you learn how to control database access to specific objects and add new users with different levels of access privileges.
Controlling User Access

In a multiple-user environment, you want to maintain security of the database access and use. With Oracle server database security, you can do the following:

- Control database access
- Give access to specific objects in the database
- Confirm given and received privileges with the Oracle data dictionary
- Create synonyms for database objects

Database security can be classified into two categories: system security and data security. System security covers access and use of the database at the system level, such as the username and password, the disk space allocated to users, and the system operations that users can perform. Database security covers access and use of the database objects and the actions that those users can have on the objects.
Privileges

- **Database security:**
  - System security
  - Data security

- **System privileges:** Gaining access to the database

- **Object privileges:** Manipulating the content of the database objects

- **Schemas:** Collections of objects, such as tables, views, and sequences

Privileges

Privileges are the right to execute particular SQL statements. The database administrator (DBA) is a high-level user with the ability to grant users access to the database and its objects. The users require system privileges to gain access to the database and object privileges to manipulate the content of the objects in the database. Users can also be given the privilege to grant additional privileges to other users or to roles, which are named groups of related privileges.

Schemas

A schema is a collection of objects, such as tables, views, and sequences. The schema is owned by a database user and has the same name as that user.

System Privileges

- More than 100 privileges are available.
- The database administrator has high-level system privileges for tasks such as:
  - Creating new users
  - Removing users
  - Removing tables
  - Backing up tables

System Privileges

More than 100 distinct system privileges are available for users and roles. System privileges typically are provided by the database administrator.

**Typical DBA Privileges**

<table>
<thead>
<tr>
<th>System Privilege</th>
<th>Operations Authorized</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE USER</td>
<td>Grantee can create other Oracle users (a privilege required for a DBA role).</td>
</tr>
<tr>
<td>DROP USER</td>
<td>Grantee can drop another user.</td>
</tr>
<tr>
<td>DROP ANY TABLE</td>
<td>Grantee can drop a table in any schema.</td>
</tr>
<tr>
<td>BACKUP ANY TABLE</td>
<td>Grantee can back up any table in any schema with the export utility.</td>
</tr>
<tr>
<td>SELECT ANY TABLE</td>
<td>Grantee can query tables, views, or snapshots in any schema.</td>
</tr>
<tr>
<td>CREATE ANY TABLE</td>
<td>Grantee can create tables in any schema.</td>
</tr>
</tbody>
</table>
Creating Users

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

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

```
CREATE USER scott
IDENTIFIED BY tiger;
User created.
```

Creating a User

The DBA creates the user by executing the `CREATE USER` statement. The user does not have any privileges at this point. The DBA can then grant privileges to that user. These privileges determine what the user can do at the database level.

The slide gives the abridged syntax for creating a user.

In the syntax:
- `user` is the name of the user to be created
- `password` specifies that the user must log in with this password

For more information, see Oracle9i SQL Reference, “GRANT” and “CREATE USER.”
User System Privileges

- Once a user is created, the DBA can grant specific system privileges to a user.

```
GRANT privilege [, privilege...] 
TO user [, user| role, PUBLIC...];
```

- An application developer, for example, may have the following system privileges:
  - CREATE SESSION
  - CREATE TABLE
  - CREATE SEQUENCE
  - CREATE VIEW
  - CREATE PROCEDURE

Typical User Privileges

Now that the DBA has created a user, the DBA can assign privileges to that user.

<table>
<thead>
<tr>
<th>System Privilege</th>
<th>Operations Authorized</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE SESSION</td>
<td>Connect to the database</td>
</tr>
<tr>
<td>CREATE TABLE</td>
<td>Create tables in the user’s schema</td>
</tr>
<tr>
<td>CREATE SEQUENCE</td>
<td>Create a sequence in the user’s schema</td>
</tr>
<tr>
<td>CREATE VIEW</td>
<td>Create a view in the user’s schema</td>
</tr>
<tr>
<td>CREATE PROCEDURE</td>
<td>Create a stored procedure, function, or package in the user’s schema</td>
</tr>
</tbody>
</table>

In the syntax:

- `privilege` is the system privilege to be granted
- `user | role | PUBLIC` is the name of the user, the name of the role, or PUBLIC designates that every user is granted the privilege

**Note:** Current system privileges can be found in the dictionary view `SESSION_PRIVS`. 

Introduction to Oracle9i: SQL 13-7
Granting System Privileges

The DBA can grant a user specific system privileges.

```
GRANT create session, create table,
create sequence, create view
TO scott;
Grant succeeded.
```

Granting System Privileges

The DBA uses the `GRANT` statement to allocate system privileges to the user. Once the user has been granted the privileges, the user can immediately use those privileges.

In the example in the slide, user Scott has been assigned the privileges to create sessions, tables, sequences, and views.
What Is a Role?

A role is a named group of related privileges that can be granted to the user. This method makes it easier to revoke and maintain privileges.

A user can have access to several roles, and several users can be assigned the same role. Roles are typically created for a database application.

Creating and Assigning a Role

First, the DBA must create the role. Then the DBA can assign privileges to the role and users to the role.

Syntax

```
CREATE ROLE role;
```

In the syntax:

- `role` is the name of the role to be created

Now that the role is created, the DBA can use the `GRANT` statement to assign users to the role as well as assign privileges to the role.
Creating and Granting Privileges to a Role

- **Create a role**
  
  ```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.
  ```

---

**Creating a Role**

The example in the slide creates a manager role and then allows managers to create tables and views. It then grants DeHaan and Kochhar the role of managers. Now DeHaan and Kochhar can create tables and views.

If users have multiple roles granted to them, they receive all of the privileges associated with all of the roles.
Changing Your Password

- The DBA creates your user account and initializes your password.
- You can change your password by using the ALTER USER statement.

```sql
ALTER USER scott
IDENTIFIED BY lion;
User altered.
```

### Changing Your Password

The DBA creates an account and initializes a password for every user. You can change your password by using the ALTER USER statement.

**Syntax**

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

In the syntax:

- `user` is the name of the user
- `password` specifies the new password

Although this statement can be used to change your password, there are many other options. You must have the ALTER USER privilege to change any other option.

For more information, see *Oracle9i SQL Reference*, “ALTER USER.”
### Object Privileges

An **object privilege** is a privilege or right to perform a particular action on a specific table, view, sequence, or procedure. Each object has a particular set of grantable privileges. The table in the slide lists the privileges for various objects. Note that the only privileges that apply to a sequence are **SELECT** and **ALTER**. **DELETE**, **UPDATE**, **REFERENCES**, and **INSERT** can be restricted by specifying a subset of updatable columns. A **SELECT** privilege can be restricted by creating a view with a subset of columns and granting the **SELECT** privilege only on the view. A privilege granted on a synonym is converted to a privilege on the base table referenced by the synonym.

<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

Different object privileges are available for different types of schema objects. A user automatically has all object privileges for schema objects contained in the user’s schema. A user can grant any object privilege on any schema object that the user owns to any other user or role. If the grant includes WITH GRANT OPTION, then the grantee can further grant the object privilege to other users; otherwise, the grantee can use the privilege but cannot grant it to other users.

In the syntax:

- `object_priv` is an object privilege to be granted
- `ALL` specifies all object privileges
- `columns` specifies the column from a table or view on which privileges are granted
- `ON object` is the object on which the privileges are granted
- `TO` identifies to whom the privilege is granted
- `PUBLIC` grants object privileges to all users
- `WITH GRANT OPTION` allows the grantee to grant the object privileges to other users and roles
Granting Object Privileges

- Grant query privileges on the EMPLOYEES table.

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

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

```
GRANT update (department_name, location_id)
ON departments
TO scott, manager;
Grant succeeded.
```

Guidelines

- To grant privileges on an object, the object must be in your own schema, or you must have been granted the object privileges WITH GRANT OPTION.
- An object owner can grant any object privilege on the object to any other user or role of the database.
- The owner of an object automatically acquires all object privileges on that object.

The first example in the slide grants users Sue and Rich the privilege to query your EMPLOYEES table. The second example grants UPDATE privileges on specific columns in the DEPARTMENTS table to Scott and to the manager role.

If Sue or Rich now want to SELECT data from the employees table, the syntax they must use is:

```
SELECT *
FROM scott.employees;
```

Alternatively, they can create a synonym for the table and SELECT from the synonym:

```
CREATE SYNONYM emp FOR scott.employees;
SELECT * FROM emp;
```

**Note:** DBAs generally allocate system privileges; any user who owns an object can grant object privileges.
**Using the WITH GRANT OPTION and PUBLIC Keywords**

- Give a user authority to pass along privileges.

```
GRANT select, insert
ON departments
TO scott
WITH GRANT OPTION;
Grant succeeded.
```

- Allow all users on the system to query data from Alice’s DEPARTMENTS table.

```
GRANT select
ON alice.departments
TO PUBLIC;
Grant succeeded.
```

**The WITH GRANT OPTION Keyword**

A privilege that is granted with the WITH GRANT OPTION clause can be passed on to other users and roles by the grantee. Object privileges granted with the WITH GRANT OPTION clause are revoked when the grantor’s privilege is revoked.

The example in the slide gives user Scott access to your DEPARTMENTS table with the privileges to query the table and add rows to the table. The example also allows Scott to give others these privileges.

**The PUBLIC Keyword**

An owner of a table can grant access to all users by using the PUBLIC keyword.

The second example allows all users on the system to query data from Alice’s DEPARTMENTS table.
## Confirming Privileges Granted

<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>

### Confirming Granted Privileges

If you attempt to perform an unauthorized operation (for example, deleting a row from a table for which you do not have the `DELETE` privilege) the Oracle Server does not permit the operation to take place.

If you receive the Oracle Server error message `table or view does not exist`, you have done either of the following:

- Named a table or view that does not exist
- Attempted to perform an operation on a table or view for which you do not have the appropriate privilege

You can access the data dictionary to view the privileges that you have. The chart in the slide describes various data dictionary views.
How to Revoke Object Privileges

- You use the `REVOKE` statement to revoke privileges granted to other users.
- Privileges granted to others through the `WITH GRANT OPTION` clause are also revoked.

```
REVOKE {privilege [, privilege...] | ALL}
ON object
FROM {user[, user...] | role | PUBLIC}
[CASCADE CONSTRAINTS];
```

Revoking Object Privileges

Remove privileges granted to other users by using the `REVOKE` statement. When you use the `REVOKE` statement, the privileges that you specify are revoked from the users you name and from any other users to whom those privileges were granted through the `WITH GRANT OPTION` clause.

In the syntax:

- `CASCADE` is required to remove any referential integrity constraints made to the object by means of the `REFERENCES` privilege.

For more information, see *Oracle9i SQL Reference*, “REVOKE.”
Revoking Object Privileges

As user Alice, revoke the **SELECT** and **INSERT** privileges given to user Scott on the **DEPARTMENTS** table.

```
REVOKE select, insert
ON departments
FROM scott;
```

Revoke succeeded.

**Revoking Object Privileges (continued)**

The example in the slide revokes **SELECT** and **INSERT** privileges given to user Scott on the **DEPARTMENTS** table.

**Note:** If a user is granted a privilege with the **WITH GRANT OPTION** clause, that user can also grant the privilege with the **WITH GRANT OPTION** clause, so that a long chain of grantees is possible, but no circular grants are permitted. If the owner revokes a privilege from a user who granted the privilege to other users, the revoking cascades to all privileges granted.

For example, if user A grants **SELECT** privilege on a table to user B including the **WITH GRANT OPTION** clause, user B can grant to user C the **SELECT** privilege with the **WITH GRANT OPTION** clause as well, and user C can then grant to user D the **SELECT** privilege. If user A revokes privilege from user B, then the privileges granted to users C and D are also revoked.
Database Links

A database link connection allows local users to access data on a remote database.

A database link is a pointer that defines a one-way communication path from an Oracle database server to another database server. The link pointer is actually defined as an entry in a data dictionary table. To access the link, you must be connected to the local database that contains the data dictionary entry.

A database link connection is one-way in the sense that a client connected to local database A can use a link stored in database A to access information in remote database B, but users connected to database B cannot use the same link to access data in database A. If local users on database B want to access data on database A, they must define a link that is stored in the data dictionary of database B.

A database link connection gives local users access to data on a remote database. For this connection to occur, each database in the distributed system must have a unique global database name. The global database name uniquely identifies a database server in a distributed system.

The great advantage of database links is that they allow users to access another user’s objects in a remote database so that they are bounded by the privilege set of the object's owner. In other words, a local user can access a remote database without having to be a user on the remote database.

The example shows a user SCOTT accessing the EMP table on the remote database with the global name HQ.ACME.COM.

Note: Typically, the DBA is responsible for creating the database link. The dictionary view USER_DB_LINKS contains information on links to which a user has access.
Database Links

- Create the database link.

```
CREATE PUBLIC DATABASE LINK hq.acme.com
USING 'sales';
Database link created.
```

- Write SQL statements that use the database link.

```
SELECT *
FROM fred.emp@HQ.ACME.COM;
```

Using Database Links

The example in the slide creates a database link. The `USING` clause identifies the service name of a remote database.

Once the database link is created, you can write SQL statements against the data in the remote site. If a synonym is set up, you can write SQL statements using the synonym.

For example:

```
CREATE PUBLIC SYNONYM HQ_EMP FOR emp@HQ.ACME.COM;
```

Then write a SQL statement that uses the synonym:

```
SELECT * FROM HQ_EMP;
```

You cannot grant privileges on remote objects.
Summary

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

<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 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>

Summary

DBAs establish initial database security for users by assigning privileges to the users.

- The DBA creates users who must have a password. The DBA is also responsible for establishing the initial system privileges for a user.
- Once the user has created an object, the user can pass along any of the available object privileges to other users or to all users by using the GRANT statement.
- A DBA can create roles by using the CREATE ROLE statement to pass along a collection of system or object privileges to multiple users. Roles make granting and revoking privileges easier to maintain.
- Users can change their password by using the ALTER USER statement.
- You can remove privileges from users by using the REVOKE statement.
- With data dictionary views, users can view the privileges granted to them and those that are granted on their objects.
- With database links, you can access data on remote databases. Privileges cannot be granted on remote objects.
Practice 13 Overview

This practice covers the following topics:

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

Practice 13 Overview

Team up with other students for this exercise about controlling access to database objects.
Practice 13

1. What privilege should a user be given to log on to the Oracle Server? Is this a system or an object privilege?

_____________________________________________________________________

2. What privilege should a user be given to create tables?

_____________________________________________________________________

3. If you create a table, who can pass along privileges to other users on your table?

_____________________________________________________________________

4. You are the DBA. You are creating many users who require the same system privileges. What should you use to make your job easier?

_____________________________________________________________________

5. What command do you use to change your password?

_____________________________________________________________________

6. Grant another user access to your DEPARTMENTS table. Have the user grant you query access to his or her DEPARTMENTS table.

7. Query all the rows in your DEPARTMENTS table.

<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.

8. Add a new row to your DEPARTMENTS table. Team 1 should add Education as department number 500. Team 2 should add Human Resources department number 510. Query the other team’s table.

9. Create a synonym for the other team’s DEPARTMENTS table.
10. Query all the rows in the other team’s DEPARTMENTS table by using your synonym.

**Team 1 SELECT statement results:**

<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>510</td>
<td>Human Resources</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>
<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>

9 rows selected.

**Team 2 SELECT statement results:**

<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>500</td>
<td>Education</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>
<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>

9 rows selected.
Practice 13 (continued)

11. Query the USER_TABLES data dictionary to see information about the tables that you own.

<table>
<thead>
<tr>
<th>TABLE_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNTRIES</td>
</tr>
<tr>
<td>DEPARTMENTS</td>
</tr>
<tr>
<td>EMPLOYEES</td>
</tr>
<tr>
<td>JOBS</td>
</tr>
<tr>
<td>JOB_GRADES</td>
</tr>
<tr>
<td>JOB_HISTORY</td>
</tr>
<tr>
<td>LOCATIONS</td>
</tr>
<tr>
<td>REGIONS</td>
</tr>
</tbody>
</table>

8 rows selected.

12. Query the ALL_TABLES data dictionary view to see information about all the tables that you can access. Exclude tables that are you own.

   Note: Your list may not exactly match the list shown below.

<table>
<thead>
<tr>
<th>TABLE_NAME</th>
<th>OWNER</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEPARTMENTS</td>
<td>owner</td>
</tr>
</tbody>
</table>

13. Revoke the SELECT privilege on your table from the other team.
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

Workshop Overview

In this workshop you build a set of database tables for a video application. After you create the tables, you insert, update, and delete records in a video store database and generate a report. The database contains only the essential tables.

**Note:** If you want to build the tables, you can execute the commands in the `buildtab.sql` script in iSQL*Plus. If you want to drop the tables, you can execute the commands in `dropvid.sql` script in iSQL*Plus. Then you can execute the commands in `buildvid.sql` script in iSQL*Plus to create and populate the tables. If you use the `buildvid.sql` script to build and populate the tables, start with step 6b.
Video Application Entity Relationship Diagram

**TITLE**
- **ID**
- *title
- *description
- o rating
- o category
- o release date

**TITLE_COPY**
- **ID**
- *status

**MEMBER**
- **ID**
- *last name
- o first name
- o address
- o city
- o phone
- *join date

**RENTAL**
- **ID**
- *book date
- o act ret date
- o exp ret date

**RESERVATION**
- **ID**
- reservation date

Set up for

Available as a copy

Made against

Created for

Responsible for

Responsible for
Practice 14

1. Create the tables based on the following table instance charts. Choose the appropriate data types and be sure to add integrity constraints.

   a. Table name: MEMBER

<table>
<thead>
<tr>
<th>Column Name</th>
<th>MEMBER_ID</th>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>ADDRESS</th>
<th>CITY</th>
<th>PHONE</th>
<th>JOIN_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Type</td>
<td>PK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Null/Unique</td>
<td>NN, U</td>
<td>NN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>System Date</td>
</tr>
<tr>
<td>Default Value</td>
<td>System Date</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Type</td>
<td>NUMBER</td>
<td>VARCHAR2</td>
<td>VARCHAR2</td>
<td>VARCHAR2</td>
<td>VARCHAR2</td>
<td>VARCHAR2</td>
<td>DATE</td>
</tr>
<tr>
<td>Length</td>
<td>10</td>
<td>25</td>
<td>25</td>
<td>100</td>
<td>30</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

   b. Table name: TITLE

<table>
<thead>
<tr>
<th>Column Name</th>
<th>TITLE_ID</th>
<th>TITLE</th>
<th>DESCRIPTION</th>
<th>RATING</th>
<th>CATEGORY</th>
<th>RELEASE_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Type</td>
<td>PK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Null/Unique</td>
<td>NN, U</td>
<td>NN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Check</td>
<td>G, PG, R, NC17, NR</td>
<td>DRAMA, COMEDY, ACTION, CHILD, SCIFI, DOCUMENTARY</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Type</td>
<td>NUMBER</td>
<td>VARCHAR2</td>
<td>VARCHAR2</td>
<td>VARCHAR2</td>
<td>VARCHAR2</td>
<td>DATE</td>
</tr>
<tr>
<td>Length</td>
<td>10</td>
<td>60</td>
<td>400</td>
<td>4</td>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>
### Practice 14 (continued)

**c. Table name: TITLE_COPY**

<table>
<thead>
<tr>
<th>Column Name</th>
<th>COPY_ID</th>
<th>TITLE_ID</th>
<th>STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Key Type</strong></td>
<td>PK</td>
<td>PK,FK</td>
<td></td>
</tr>
<tr>
<td><strong>Null/Unique</strong></td>
<td>NN,U</td>
<td>NN,U</td>
<td>NN</td>
</tr>
<tr>
<td><strong>Check</strong></td>
<td></td>
<td>AVAILABLE, DESTROYED, RENTED, RESERVED</td>
<td></td>
</tr>
<tr>
<td><strong>FK Ref Table</strong></td>
<td>TITLIE</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>FK Ref Col</strong></td>
<td>TITLE_ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Data Type</strong></td>
<td>NUMBER</td>
<td>NUMBER</td>
<td>VARCHAR2</td>
</tr>
<tr>
<td><strong>Length</strong></td>
<td>10</td>
<td>10</td>
<td>15</td>
</tr>
</tbody>
</table>

**d. Table name: RENTAL**

<table>
<thead>
<tr>
<th>Column Name</th>
<th>BOOK_DATE</th>
<th>MEMBER_ID</th>
<th>COPY_ID</th>
<th>ACT_RET_DATE</th>
<th>EXP_RET_DATE</th>
<th>TITLE_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Key Type</strong></td>
<td>PK</td>
<td>PK,FK1</td>
<td>PK,FK2</td>
<td></td>
<td></td>
<td>PK,FK2</td>
</tr>
<tr>
<td><strong>Default Value</strong></td>
<td>System Date</td>
<td>System Date + 2 days</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>FK Ref Table</strong></td>
<td>MEMBER</td>
<td>TITLE_COPY</td>
<td></td>
<td>TITLE_COPY</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>FK Ref Col</strong></td>
<td>MEMBER_ID</td>
<td>COPY_ID</td>
<td></td>
<td>TITLE_ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Data Type</strong></td>
<td>DATE</td>
<td>NUMBER</td>
<td>NUMBER</td>
<td>DATE</td>
<td>DATE</td>
<td>NUMBER</td>
</tr>
<tr>
<td><strong>Length</strong></td>
<td>10</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>
Practice 14 (continued)

e. Table name: RESERVATION

<table>
<thead>
<tr>
<th>Column Name</th>
<th>RES_DATE</th>
<th>MEMBER_ID</th>
<th>TITLE_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Type</td>
<td>PK</td>
<td>PK,FK1</td>
<td>PK,FK2</td>
</tr>
<tr>
<td>Null/Unique</td>
<td>NN,U</td>
<td>NN,U</td>
<td>NN</td>
</tr>
<tr>
<td>FK Ref Table</td>
<td></td>
<td>MEMBER</td>
<td>TITLE</td>
</tr>
<tr>
<td>FK Ref Column</td>
<td></td>
<td>MEMBER_ID</td>
<td>TITLE_ID</td>
</tr>
<tr>
<td>Data Type</td>
<td>DATE</td>
<td>NUMBER</td>
<td>NUMBER</td>
</tr>
<tr>
<td>Length</td>
<td>10</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

2. Verify that the tables and constraints were created properly by checking the data dictionary.

<table>
<thead>
<tr>
<th>TABLE_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEMBER</td>
</tr>
<tr>
<td>RENTAL</td>
</tr>
<tr>
<td>RESERVATION</td>
</tr>
<tr>
<td>TITLE</td>
</tr>
<tr>
<td>TITLE_COPY</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CONSTRAINT_NAME</th>
<th>C</th>
<th>TABLE_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEMBER_LAST_NAME_NN</td>
<td>C</td>
<td>MEMBER</td>
</tr>
<tr>
<td>MEMBER_JOIN_DATE_NN</td>
<td>C</td>
<td>MEMBER</td>
</tr>
<tr>
<td>MEMBER_MEMBER_ID_PK</td>
<td>P</td>
<td>MEMBER</td>
</tr>
<tr>
<td>RENTAL_BOOK_DATE_COPY_TITLE_PK</td>
<td>P</td>
<td>RENTAL</td>
</tr>
<tr>
<td>RENTAL_MEMBER_ID_FK</td>
<td>R</td>
<td>RENTAL</td>
</tr>
<tr>
<td>RENTAL_COPY_ID_TITLE_ID_FK</td>
<td>R</td>
<td>RENTAL</td>
</tr>
<tr>
<td>RESERVATION_RESDATE_MEM_TIT_PK</td>
<td>P</td>
<td>RESERVATION</td>
</tr>
<tr>
<td>RESERVATION_MEMBER_ID</td>
<td>R</td>
<td>RESERVATION</td>
</tr>
<tr>
<td>TITLE_COPY_COPY_ID_TITLE_ID_PK</td>
<td>P</td>
<td>TITLE_COPY</td>
</tr>
</tbody>
</table>

18 rows selected.
Practice 14 (continued)

3. Create sequences to uniquely identify each row in the MEMBER table and the TITLE table.
   a. Member number for the MEMBER table: Start with 101; do not allow caching of the values. Name the sequence MEMBER_ID_SEQ.
   b. Title number for the TITLE table: Start with 92; no caching. Name the sequence TITLE_ID_SEQ.
   c. Verify the existence of the sequences in the data dictionary.

<table>
<thead>
<tr>
<th>SEQUENCE_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEMBER_ID_SEQ</td>
</tr>
<tr>
<td>TITLE_ID_SEQ</td>
</tr>
</tbody>
</table>

4. Add data to the tables. Create a script for each set of data to add.
   a. Add movie titles to the TITLE table. Write a script to enter the movie information.
      Save the statements in a script named lab14_4a.sql. Use the sequences to uniquely identify each title. Enter the release dates in the DD-MON-YYYY format. Remember that single quotation marks in a character field must be specially handled. Verify your additions.

<table>
<thead>
<tr>
<th>TITLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Willie and Christmas Too</td>
</tr>
<tr>
<td>Alien Again</td>
</tr>
<tr>
<td>The Glob</td>
</tr>
<tr>
<td>My Day Off</td>
</tr>
<tr>
<td>Miracles on Ice</td>
</tr>
<tr>
<td>Soda Gang</td>
</tr>
</tbody>
</table>

6 rows selected.
<table>
<thead>
<tr>
<th>Title</th>
<th>Description</th>
<th>Rating</th>
<th>Category</th>
<th>Release_date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Willie and Christmas Too</td>
<td>All of Willie’s friends make a Christmas list for Santa, but Willie has yet to add his own wish list.</td>
<td>G</td>
<td>CHILD</td>
<td>05-OCT-1995</td>
</tr>
<tr>
<td>Alien Again</td>
<td>Yet another installation of science fiction history. Can the heroine save the planet from the alien life form?</td>
<td>R</td>
<td>SCIFI</td>
<td>19-MAY-1995</td>
</tr>
<tr>
<td>The Glob</td>
<td>A meteor crashes near a small American town and unleashes carnivorous goo in this classic.</td>
<td>NR</td>
<td>SCIFI</td>
<td>12-AUG-1995</td>
</tr>
<tr>
<td>My Day Off</td>
<td>With a little luck and a lot of ingenuity, a teenager skips school for a day in New York</td>
<td>PG</td>
<td>COMEDY</td>
<td>12-JUL-1995</td>
</tr>
<tr>
<td>Miracles on Ice</td>
<td>A six-year-old has doubts about Santa Claus, but she discovers that miracles really do exist.</td>
<td>PG</td>
<td>DRAMA</td>
<td>12-SEP-1995</td>
</tr>
<tr>
<td>Soda Gang</td>
<td>After discovering a cache of drugs, a young couple find themselves pitted against a vicious gang.</td>
<td>NR</td>
<td>ACTION</td>
<td>01-JUN-1995</td>
</tr>
</tbody>
</table>
b. Add data to the MEMBER table. Place the insert statements in a script named `lab14_4b.sql`. Execute commands in the script. Be sure to use the sequence to add the member numbers.

<table>
<thead>
<tr>
<th>First_Name</th>
<th>Last_Name</th>
<th>Address</th>
<th>City</th>
<th>Phone</th>
<th>Join_Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carmen</td>
<td>Velasquez</td>
<td>283 King Street</td>
<td>Seattle</td>
<td>206-899-6666</td>
<td>08-MAR-1990</td>
</tr>
<tr>
<td>LaDoris</td>
<td>Ngao</td>
<td>5 Modrany</td>
<td>Bratislava</td>
<td>586-355-8882</td>
<td>08-MAR-1990</td>
</tr>
<tr>
<td>Midori</td>
<td>Nagayama</td>
<td>68 Via Centrale</td>
<td>Sao Paolo</td>
<td>254-852-5764</td>
<td>17-JUN-1991</td>
</tr>
<tr>
<td>Mark</td>
<td>Quick-to-See</td>
<td>6921 King Way</td>
<td>Lagos</td>
<td>63-559-7777</td>
<td>07-APR-1990</td>
</tr>
<tr>
<td>Audry</td>
<td>Ropeburn</td>
<td>86 Chu Street</td>
<td>Hong Kong</td>
<td>41-559-87</td>
<td>18-JAN-1991</td>
</tr>
<tr>
<td>Molly</td>
<td>Urguhart</td>
<td>3035 Laurier</td>
<td>Quebec</td>
<td>418-542-9988</td>
<td>18-JAN-1991</td>
</tr>
</tbody>
</table>
Practice 14 (continued)

c. Add the following movie copies in the TITLE_COPY table:
   
   **Note:** Have the TITLE_ID numbers available for this exercise.

<table>
<thead>
<tr>
<th>Title</th>
<th>Copy_Id</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Willie and Christmas Too</td>
<td>1</td>
<td>AVAILABLE</td>
</tr>
<tr>
<td>Alien Again</td>
<td>1</td>
<td>AVAILABLE</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>RENTED</td>
</tr>
<tr>
<td>The Glob</td>
<td>1</td>
<td>AVAILABLE</td>
</tr>
<tr>
<td>My Day Off</td>
<td>1</td>
<td>AVAILABLE</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>AVAILABLE</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>RENTED</td>
</tr>
<tr>
<td>Miracles on Ice</td>
<td>1</td>
<td>AVAILABLE</td>
</tr>
<tr>
<td>Soda Gang</td>
<td>1</td>
<td>AVAILABLE</td>
</tr>
</tbody>
</table>


d. Add the following rentals to the RENTAL table:
   
   **Note:** Title number may be different depending on sequence number.

<table>
<thead>
<tr>
<th>Title_Id</th>
<th>Copy_Id</th>
<th>Member_Id</th>
<th>Book_date</th>
<th>Exp_Ret_Date</th>
<th>Act_Ret_Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>92</td>
<td>1</td>
<td>101</td>
<td>3 days ago</td>
<td>1 day ago</td>
<td>2 days ago</td>
</tr>
<tr>
<td>93</td>
<td>2</td>
<td>101</td>
<td>1 day ago</td>
<td>1 day from now</td>
<td>2 days ago</td>
</tr>
<tr>
<td>95</td>
<td>3</td>
<td>102</td>
<td>2 days ago</td>
<td>Today</td>
<td></td>
</tr>
<tr>
<td>97</td>
<td>1</td>
<td>106</td>
<td>4 days ago</td>
<td>2 days ago</td>
<td>2 days ago</td>
</tr>
</tbody>
</table>
Practice 14 (continued)

5. Create a view named TITLE_AVAIL to show the movie titles and the availability of each copy and its expected return date if rented. Query all rows from the view. Order the results by title.

   Note: Your results may be different.

<table>
<thead>
<tr>
<th>TITLE</th>
<th>COPY_ID</th>
<th>STATUS</th>
<th>EXP_RET_D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alien Again</td>
<td>1</td>
<td>AVAILABLE</td>
<td></td>
</tr>
<tr>
<td>Alien Again</td>
<td>2</td>
<td>RENTED</td>
<td>15-MAR-01</td>
</tr>
<tr>
<td>Miracles on Ice</td>
<td>1</td>
<td>AVAILABLE</td>
<td></td>
</tr>
<tr>
<td>My Day Off</td>
<td>1</td>
<td>AVAILABLE</td>
<td></td>
</tr>
<tr>
<td>My Day Off</td>
<td>2</td>
<td>AVAILABLE</td>
<td></td>
</tr>
<tr>
<td>My Day Off</td>
<td>3</td>
<td>RENTED</td>
<td>16-MAR-01</td>
</tr>
<tr>
<td>Soda Gang</td>
<td>1</td>
<td>AVAILABLE</td>
<td>14-MAR-01</td>
</tr>
<tr>
<td>The Glob</td>
<td>1</td>
<td>AVAILABLE</td>
<td></td>
</tr>
<tr>
<td>Willie and Christmas Too</td>
<td>1</td>
<td>AVAILABLE</td>
<td>15-MAR-01</td>
</tr>
</tbody>
</table>

9 rows selected.

6. Make changes to data in the tables.
   a. Add a new title. The movie is “Interstellar Wars,” which is rated PG and classified as a sci-fi movie. The release date is 07-JUL-77. The description is “Futuristic interstellar action movie. Can the rebels save the humans from the evil empire?” Be sure to add a title copy record for two copies.

   b. Enter two reservations. One reservation is for Carmen Velasquez, who wants to rent “Interstellar Wars.” The other is for Mark Quick-to-See, who wants to rent “Soda Gang.”
Practice 14 (continued)

c. Customer Carmen Velasquez rents the movie “Interstellar Wars,” copy 1. Remove her reservation for the movie. Record the information about the rental. Allow the default value for the expected return date to be used. Verify that the rental was recorded by using the view you created.

**Note:** Your results may be different.

<table>
<thead>
<tr>
<th>TITLE</th>
<th>COPY_ID</th>
<th>STATUS</th>
<th>EXP_RET_D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alien Again</td>
<td>1</td>
<td>AVAILABLE</td>
<td></td>
</tr>
<tr>
<td>Alien Again</td>
<td>2</td>
<td>RENTED</td>
<td>15-MAR-01</td>
</tr>
<tr>
<td>Interstellar Wars</td>
<td>1</td>
<td>RENTED</td>
<td>18-MAR-01</td>
</tr>
<tr>
<td>Interstellar Wars</td>
<td>2</td>
<td>AVAILABLE</td>
<td></td>
</tr>
<tr>
<td>Miracles on Ice</td>
<td>1</td>
<td>AVAILABLE</td>
<td></td>
</tr>
<tr>
<td>My Day Off</td>
<td>1</td>
<td>AVAILABLE</td>
<td></td>
</tr>
<tr>
<td>My Day Off</td>
<td>2</td>
<td>AVAILABLE</td>
<td></td>
</tr>
<tr>
<td>My Day Off</td>
<td>3</td>
<td>RENTED</td>
<td>16-MAR-01</td>
</tr>
<tr>
<td>Soda Gang</td>
<td>1</td>
<td>AVAILABLE</td>
<td>14-MAR-01</td>
</tr>
<tr>
<td>The Glob</td>
<td>1</td>
<td>AVAILABLE</td>
<td></td>
</tr>
<tr>
<td>Willie and Christmas Too</td>
<td>1</td>
<td>AVAILABLE</td>
<td>15-MAR-01</td>
</tr>
</tbody>
</table>

11 rows selected.

7. Make a modification to one of the tables.

a. Add a `PRICE` column to the `TITLE` table to record the purchase price of the video. The column should have a total length of eight digits and two decimal places. Verify your modifications.

<table>
<thead>
<tr>
<th>Name</th>
<th>Null?</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>TITLE_ID</td>
<td>NOT NULL</td>
<td>NUMBER(10)</td>
</tr>
<tr>
<td>TITLE</td>
<td>NOT NULL</td>
<td>VARCHAR2(60)</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>NOT NULL</td>
<td>VARCHAR2(400)</td>
</tr>
<tr>
<td>RATING</td>
<td>NOT NULL</td>
<td>VARCHAR2(4)</td>
</tr>
<tr>
<td>CATEGORY</td>
<td></td>
<td>VARCHAR2(20)</td>
</tr>
<tr>
<td>RELEASE_DATE</td>
<td></td>
<td>DATE</td>
</tr>
<tr>
<td>PRICE</td>
<td></td>
<td>NUMBER(8,2)</td>
</tr>
</tbody>
</table>
Practice 14 (continued)

b. Create a script named **lab14_7b.sql** that contains update statements that update each video with a price according to the following list. Run the commands in the script.

**Note:** Have the TITLE_ID numbers available for this exercise.

<table>
<thead>
<tr>
<th>Title</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Willie and Christmas Too</td>
<td>25</td>
</tr>
<tr>
<td>Alien Again</td>
<td>35</td>
</tr>
<tr>
<td>The Glob</td>
<td>35</td>
</tr>
<tr>
<td>My Day Off</td>
<td>35</td>
</tr>
<tr>
<td>Miracles on Ice</td>
<td>30</td>
</tr>
<tr>
<td>My Day Off</td>
<td>35</td>
</tr>
<tr>
<td>Interstellar Wars</td>
<td>29</td>
</tr>
</tbody>
</table>

![constraint image]

<table>
<thead>
<tr>
<th>CONSTRAINT_NAME</th>
<th>SEARCH_CONDITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>TITLEPRICE_NN</td>
<td>&quot;PRICE&quot; IS NOT NULL</td>
</tr>
</tbody>
</table>

8. Create a report titled Customer History Report. This report contains each customer’s history of renting videos. Be sure to include the customer name, movie rented, dates of the rental, and duration of rentals. Total the number of rentals for all customers for the reporting period. Save the commands that generate the report in a script file named **lab14_8.sql**.

**Note:** Your results may be different.

<table>
<thead>
<tr>
<th>Fri Mar 16</th>
<th>Customer History Report</th>
<th>page 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEMBER</td>
<td>TITLE</td>
<td>BOOK_DATE</td>
</tr>
<tr>
<td>Carmen Velasquez</td>
<td>Willie and Christmas Too</td>
<td>13-MAR-01</td>
</tr>
<tr>
<td></td>
<td>Alien Again</td>
<td>15-MAR-01</td>
</tr>
<tr>
<td></td>
<td>Interstellar Wars</td>
<td>16-MAR-01</td>
</tr>
<tr>
<td>LaDoris Ngao</td>
<td>My Day Off</td>
<td>14-MAR-01</td>
</tr>
<tr>
<td>Molly Urguhart</td>
<td>Soda Gang</td>
<td>12-MAR-01</td>
</tr>
</tbody>
</table>

Introduction to Oracle9i: SQL 14-13
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

Lesson Aim

In this lesson, you learn how to write queries by using SET operators.
The SET Operators

The SET operators combine the results of two or more component queries into one result. Queries containing SET operators are called compound queries.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNION</td>
<td>All distinct rows selected by either query</td>
</tr>
<tr>
<td>UNION ALL</td>
<td>All rows selected by either query, including all duplicates</td>
</tr>
<tr>
<td>INTERSECT</td>
<td>All distinct rows selected by both queries</td>
</tr>
<tr>
<td>MINUS</td>
<td>All distinct rows that are selected by the first SELECT statement and that are not selected in the second SELECT statement</td>
</tr>
</tbody>
</table>

All SET operators have equal precedence. If a SQL statement contains multiple SET operators, the Oracle server evaluates them from left (top) to right (bottom) if no parentheses explicitly specify another order. You should use parentheses to specify the order of evaluation explicitly in queries that use the INTERSECT operator with other SET operators.

Note: In the slide, the light color (grey) in the diagram represents the query result.
Tables Used in This Lesson

The tables used in this lesson are:

- **EMPLOYEES**: Provides details regarding all current employees
- **JOB_HISTORY**: When an employee switches jobs, the details of the start date and end date of the former job, the job identification number and department are recorded in this table.

Two tables are used in this lesson. They are the EMPLOYEES table and the JOB_HISTORY table.

The EMPLOYEES table stores the employee details. For the human resource records, this table stores a unique identification number and email address for each employee. The details of the employee’s job identification number, salary, and manager are also stored. Some of the employees earn a commission in addition to their salary; this information is tracked too. The company organizes the roles of employees into jobs. Some of the employees have been with the company for a long time and have switched to different jobs. This is monitored using the JOB_HISTORY table. When an employee switches jobs, the details of the start date and end date of the former job, the job identification number and department are recorded in the JOB_HISTORY table.

The structure and the data from the EMPLOYEES and the JOB_HISTORY tables are shown on the next page.

There have been instances in the company of people who have held the same position more than once during their tenure with the company. For example, consider the employee Taylor, who joined the company on 24-MAR-1998. Taylor held the job title SA_REP for the period 24-MAR-98 to 31-DEC-98 and the job title SA_MAN for the period 01-JAN-99 to 31-DEC-99. Taylor moved back into the job title of SA_REP, which is his current job title.

Similarly consider the employee Whalen, who joined the company on 17-SEP-1987. Whalen held the job title AD_ASST for the period 17-SEP-87 to 17-JUN-93 and the job title AC_ACCOUNT for the period 01-JUL-94 to 31-DEC-98. Taylor moved back into the job title of AD_ASST, which is his current job title.
### Tables Used in This Lesson (continued)

**DESC 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>NOT NULL</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>

```sql
SELECT employee_id, last_name, job_id, hire_date, department_id
FROM employees;
```

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>JOB_ID</th>
<th>HIRE_DATE</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>King</td>
<td>AD_PRES</td>
<td>17-JUN-87</td>
<td>90</td>
</tr>
<tr>
<td>101</td>
<td>Kochhar</td>
<td>AD_VP</td>
<td>21-SEP-89</td>
<td>90</td>
</tr>
<tr>
<td>102</td>
<td>De Haan</td>
<td>AD_VP</td>
<td>13-JAN-93</td>
<td>90</td>
</tr>
<tr>
<td>103</td>
<td>Hunold</td>
<td>IT_PROG</td>
<td>03-JAN-90</td>
<td>60</td>
</tr>
<tr>
<td>104</td>
<td>Ernst</td>
<td>IT_PROG</td>
<td>21-MAY-91</td>
<td>60</td>
</tr>
<tr>
<td>107</td>
<td>Lorentz</td>
<td>IT_PROG</td>
<td>07-FEB-99</td>
<td>60</td>
</tr>
<tr>
<td>124</td>
<td>Mourgos</td>
<td>ST_MAN</td>
<td>16-NOV-99</td>
<td>50</td>
</tr>
<tr>
<td>141</td>
<td>Rajs</td>
<td>ST_CLERK</td>
<td>17-OCT-95</td>
<td>50</td>
</tr>
<tr>
<td>142</td>
<td>Davies</td>
<td>ST_CLERK</td>
<td>29-JAN-97</td>
<td>50</td>
</tr>
<tr>
<td>143</td>
<td>Matos</td>
<td>ST_CLERK</td>
<td>15-MAR-96</td>
<td>50</td>
</tr>
<tr>
<td>144</td>
<td>Vargas</td>
<td>ST_CLERK</td>
<td>09-JUL-98</td>
<td>50</td>
</tr>
<tr>
<td>149</td>
<td>Zlotkoy</td>
<td>SA_MAN</td>
<td>29-JAN-00</td>
<td>80</td>
</tr>
<tr>
<td>174</td>
<td>Abel</td>
<td>SA_REP</td>
<td>11-MAY-96</td>
<td>80</td>
</tr>
<tr>
<td>176</td>
<td>Taylor</td>
<td>SA_REP</td>
<td>24-MAR-96</td>
<td>80</td>
</tr>
<tr>
<td>176</td>
<td>Grant</td>
<td>SA_REP</td>
<td>24-MAY-96</td>
<td>80</td>
</tr>
<tr>
<td>200</td>
<td>Whalen</td>
<td>AD_ASST</td>
<td>17-SEP-87</td>
<td>10</td>
</tr>
<tr>
<td>201</td>
<td>Hartstein</td>
<td>MK_MAN</td>
<td>17-FEB-86</td>
<td>20</td>
</tr>
<tr>
<td>202</td>
<td>Fay</td>
<td>MK_REP</td>
<td>17-AUG-87</td>
<td>20</td>
</tr>
<tr>
<td>205</td>
<td>Higginso</td>
<td>AC_MGR</td>
<td>07-JUN-94</td>
<td>110</td>
</tr>
<tr>
<td>206</td>
<td>Gietz</td>
<td>AC_ACCOUNT</td>
<td>07-JUN-94</td>
<td>110</td>
</tr>
</tbody>
</table>

20 rows selected.

---

Introduction to Oracle9i: SQL 15-5
### Tables Used in This Lesson (continued)

**DESC job_history**

<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>START_DATE</td>
<td>NOT NULL</td>
<td>DATE</td>
</tr>
<tr>
<td>END_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>DEPARTMENT_ID</td>
<td></td>
<td>NUMBER(4)</td>
</tr>
</tbody>
</table>

```sql
SELECT * FROM job_history;
```

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>START_DATE</th>
<th>END_DATE</th>
<th>JOB_ID</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>102</td>
<td>13-JAN-93</td>
<td>24-JUL-98</td>
<td>IT_PROG</td>
<td>60</td>
</tr>
<tr>
<td>101</td>
<td>21-SEP-89</td>
<td>27-OCT-93</td>
<td>AC_ACCOUNT</td>
<td>110</td>
</tr>
<tr>
<td>101</td>
<td>28-OCT-93</td>
<td>15-MAR-97</td>
<td>AC_MGR</td>
<td>110</td>
</tr>
<tr>
<td>201</td>
<td>17-FEB-96</td>
<td>19-DEC-99</td>
<td>MK_REP</td>
<td>20</td>
</tr>
<tr>
<td>114</td>
<td>24-MAR-90</td>
<td>31-DEC-99</td>
<td>ST_CLERK</td>
<td>50</td>
</tr>
<tr>
<td>122</td>
<td>01-JAN-99</td>
<td>31-DEC-99</td>
<td>ST_CLERK</td>
<td>50</td>
</tr>
<tr>
<td>200</td>
<td>17-SEP-87</td>
<td>17-JUN-93</td>
<td>AD_ASST</td>
<td>90</td>
</tr>
<tr>
<td>176</td>
<td>24-MAR-98</td>
<td>31-DEC-98</td>
<td>SA_REP</td>
<td>80</td>
</tr>
<tr>
<td>176</td>
<td>01-JAN-99</td>
<td>31-DEC-99</td>
<td>SA_MAN</td>
<td>80</td>
</tr>
<tr>
<td>200</td>
<td>01-JUL-94</td>
<td>31-DEC-98</td>
<td>AC_ACCOUNT</td>
<td>90</td>
</tr>
</tbody>
</table>

10 rows selected.
The **UNION SET Operator**

The **UNION** operator returns all rows selected by either query. Use the **UNION** operator to return all rows from multiple tables and eliminate any duplicate rows.

**Guidelines**

- The number of columns and the data types of the columns being selected must be identical in all the `SELECT` statements used in the query. The names of the columns need not be identical.
- **UNION** operates over all of the columns being selected.
- **NULL** values are not ignored during duplicate checking.
- The **IN** operator has a higher precedence than the **UNION** operator.
- By default, the output is sorted in ascending order of the first column of the `SELECT` clause.
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}

Using the \texttt{UNION SET} Operator

The \texttt{UNION} operator eliminates any duplicate records. If there are records that occur both in the \texttt{EMPLOYEES} and the \texttt{JOB\_HISTORY} tables and are identical, the records will be displayed only once. Observe in the output shown on the slide that the record for the employee with the \texttt{EMPLOYEE\_ID} 200 appears twice as the \texttt{JOB\_ID} is different in each row.

Consider the following example:

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

\begin{tabular}{|c|c|}
\hline
\textbf{EMPLOYEE\_ID} & \textbf{JOE\_ID} \\
\hline
100 & AD\_PRES \\
101 & AC\_ACCOUNT \\
101 & SA\_REP \\
200 & AC\_ACCOUNT \\
200 & AD\_ASST \\
\hline
\end{tabular}

28 rows selected.

\begin{tabular}{|c|c|c|}
\hline
\textbf{EMPLOYEE\_ID} & \textbf{JOE\_ID} & \textbf{DEPARTMENT\_ID} \\
\hline
100 & AD\_PRES & 90 \\
101 & AC\_ACCOUNT & 110 \\
101 & AC\_MGR & 110 \\
101 & AD\_VP & 90 \\
200 & AC\_ACCOUNT & 90 \\
200 & AD\_ASST & 10 \\
200 & AD\_ASST & 90 \\
201 & MK\_MAN & 20 \\
\hline
\end{tabular}

29 rows selected.
Using the **UNION SET** Operator (continued)

In the preceding output, employee 200 appears three times. Why? Notice the `DEPARTMENT_ID` values for employee 200. One row has a `DEPARTMENT_ID` of 90, another 10, and the third 90. Because of these unique combinations of job IDs and department IDs, each row for employee 200 is unique and therefore not considered a duplicate. Observe that the output is sorted in ascending order of the first column of the `SELECT` clause, `EMPLOYEE_ID` in this case.
The **UNION ALL** Operator

Use the **UNION ALL** operator to return all rows from multiple queries.

**Guidelines**

- Unlike **UNION**, duplicate rows are not eliminated and the output is not sorted by default.
- The **DISTINCT** keyword cannot be used.

**Note:** With the exception of the above, the guidelines for **UNION** and **UNION ALL** are the same.
Using the UNION ALL Operator

Display the current and previous departments of all employees.

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

<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>174</td>
<td>SA_MGR</td>
<td></td>
</tr>
<tr>
<td>170</td>
<td>SA_REP</td>
<td>90</td>
</tr>
<tr>
<td>176</td>
<td>SA_MAN</td>
<td>80</td>
</tr>
<tr>
<td>174</td>
<td>SA_REP</td>
<td>80</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 UNION ALL Operator (continued)

In the example, 30 rows are selected. The combination of the two tables totals to 30 rows. The UNION ALL operator does not eliminate duplicate records. The duplicate records are highlighted in the output shown in the slide. UNION returns all distinct rows selected by either query. UNION ALL returns all rows selected by either query, including all duplicates. Consider the query on the slide, now written with the UNION clause:

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

The preceding query returns 29 rows. This is because it eliminates the following row (as it is a duplicate):

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>JOB_ID</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>176</td>
<td>SA_REP</td>
<td>80</td>
</tr>
</tbody>
</table>
The **INTERSECT** Operator

Use the **INTERSECT** operator to return all rows common to multiple queries.

**Guidelines**

- The number of columns and the data types of the columns being selected by the `SELECT` statements in the queries must be identical in all the `SELECT` statements used in the query. The names of the columns need not be identical.
- Reversing the order of the intersected tables does not alter the result.
- **INTERSECT** does not ignore **NULL** values.
Using the **INTERSECT** Operator

Display the employee IDs and job IDs of employees who are currently in a job title that they have held once before during their tenure with the company

```sql
SELECT employee_id, job_id
FROM employees
INTERSECT
SELECT employee_id, job_id
FROM job_history;
```

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>JOB_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>176</td>
<td>SA_REP</td>
</tr>
<tr>
<td>200</td>
<td>AD_ASST</td>
</tr>
</tbody>
</table>

The **INTERSECT** Operator (continued)

In the example in this slide, the query returns only the records that have the same values in the selected columns in both tables.

What will be the results if you add the `DEPARTMENT_ID` column to the `SELECT` statement from the `EMPLOYEES` table and add the `DEPARTMENT_ID` column to the `SELECT` statement from the `JOB_HISTORY` table and run this query? The results may be different because of the introduction of another column whose values may or may not be duplicates.

**Example**

```sql
SELECT employee_id, job_id, department_id
FROM employees
INTERSECT
SELECT employee_id, job_id, department_id
FROM job_history;
```

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>JOB_ID</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>176</td>
<td>SA_REP</td>
<td>80</td>
</tr>
</tbody>
</table>

Employee 200 is no longer part of the results because the `EMPLOYEES.DEPARTMENT_ID` value is different from the `JOB_HISTORY.DEPARTMENT_ID` value.
The **MINUS** Operator

Use the **MINUS** operator to return rows returned by the first query that are not present in the second query (the first `SELECT` statement **MINUS** the second `SELECT` statement).

**Guidelines**

- The number of columns and the data types of the columns being selected by the `SELECT` statements in the queries must be identical in all the `SELECT` statements used in the query. The names of the columns need not be identical.
- All of the columns in the `WHERE` clause must be in the `SELECT` clause for the **MINUS** operator to work.
The **MINUS** Operator

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

```
SELECT employee_id
FROM   employees
MINUS
SELECT employee_id
FROM   job_history;
```

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>JOB_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>AD_PRES</td>
</tr>
<tr>
<td>101</td>
<td>AD_VP</td>
</tr>
<tr>
<td>102</td>
<td>SA_SAL</td>
</tr>
<tr>
<td>201</td>
<td>MK_MAN</td>
</tr>
<tr>
<td>202</td>
<td>MK_REP</td>
</tr>
<tr>
<td>205</td>
<td>AC_MGR</td>
</tr>
<tr>
<td>206</td>
<td>AC_ACCOUNT</td>
</tr>
</tbody>
</table>

18 rows selected

The **MINUS** Operator (continued)

In the example in the slide, the employee IDs in the JOB_HISTORY table are subtracted from those in the EMPLOYEES table. The results set displays the employees remaining after the subtraction; they are represented by rows that exist in the EMPLOYEES table but do not exist in the JOB_HISTORY table. These are the records of the employees who have not changed their jobs even once.
SET Operator Guidelines

• The expressions in the SELECT lists must match in number and data type.

• Parentheses can be used to alter the sequence of execution.

• The ORDER BY clause:
  – Can appear only at the very end of the statement
  – Will accept the column name, aliases from the first SELECT statement, or the positional notation

SET Operator Guidelines

• The expressions in the select lists of the queries must match in number and datatype. Queries that use UNION, UNION ALL, INTERSECT, and MINUS SET operators in their WHERE clause must have the same number and type of columns in their SELECT list. For example:

  ```sql
  SELECT employee_id, department_id
  FROM employees
  WHERE (employee_id, department_id) IN (SELECT employee_id, department_id
                                          FROM employees
                                          UNION
                                          SELECT employee_id, department_id
                                          FROM job_history);
  ```

• The ORDER BY clause:
  – Can appear only at the very end of the statement
  – Will accept the column name, an alias, or the positional notation

• The column name or alias, if used in an ORDER BY clause, must be from the first SELECT list.

• SET operators can be used in subqueries.
The Oracle Server and \texttt{SET} Operators

- Duplicate rows are automatically eliminated except in \texttt{UNION ALL}.
- Column names from the first query appear in the result.
- The output is sorted in ascending order by default except in \texttt{UNION ALL}.

The corresponding expressions in the select lists of the component queries of a compound query must match in number and datatype. If component queries select character data, the data type of the return values are determined as follows:

- If both queries select values of datatype \texttt{CHAR}, the returned values have datatype \texttt{CHAR}.
- If either or both of the queries select values of datatype \texttt{VARCHAR2}, the returned values have datatype \texttt{VARCHAR2}.
Matching the select statements

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

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

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>LOCATION</th>
<th>HIRE_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1730</td>
<td>17-sep-87</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>17-sep-87</td>
</tr>
<tr>
<td>20</td>
<td>1000</td>
<td>17-feb-96</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>17-feb-96</td>
</tr>
<tr>
<td>190</td>
<td></td>
<td>17-may-94</td>
</tr>
<tr>
<td></td>
<td></td>
<td>24-may-99</td>
</tr>
</tbody>
</table>

27 rows selected.

Matching the SELECT statements

As the expressions in the select lists of the queries must match in number, you can use dummy columns and the data type conversion functions to comply with this rule. In the slide, the name location is given as the dummy column heading. The TO_NUMBER function is used in the first query to match the NUMBER data type of the LOCATION_ID column retrieved by the second query. Similarly, the TO_DATE function in the second query is used to match the DATE datatype of the HIRE_DATE column retrieved by the second query.
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;
```

### Example

The `EMPLOYEES` and `JOB_HISTORY` tables have several columns in common; for example, `EMPLOYEE_ID`, `JOB_ID` and `DEPARTMENT_ID`. But what if you want the query to display the `EMPLOYEE_ID`, `JOB_ID`, and `SALARY` using the `UNION` operator, knowing that the salary exists only in the `EMPLOYEES` table?

The code example in the slide matches the `EMPLOYEE_ID` and the `JOB_ID` columns in the `EMPLOYEES` and in the `JOB_HISTORY` tables. A literal value of 0 is added to the `JOB_HISTORY` SELECT statement to match the numeric `SALARY` column in the `EMPLOYEES` SELECT statement.

In the preceding results, each row in the output that corresponds to a record from the `JOB_HISTORY` table contains a 0 in the `SALARY` column.
Controlling the Order of Rows

Produce an English sentence using two UNION operators.

```
COLUMN a_dummy NOPRINT
SELECT 'sing' AS "My dream", 3 a_dummy
FROM dual
UNION
SELECT 'I''d like to teach', 1
FROM dual
UNION
SELECT 'the world to', 2
FROM dual
ORDER BY 2;
```

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

Controlling the Order of Rows

By default, the output is sorted in ascending order on the first column. You can use the ORDER BY clause to change this.

**Using ORDER BY to Order Rows**

The ORDER BY clause can be used only once in a compound query. If used, the ORDER BY clause must be placed at the end of the query. The ORDER BY clause accepts the column name, an alias, or the positional notation. Without the ORDER BY clause, the code example in the slide produces the following output in the alphabetical order of the first column:

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

**Note:** Consider a compound query where the UNION SET operator is used more than once. In this case, the ORDER BY clause can use only positions rather than explicit expressions.
Summary

In this lesson, you should have learned the following:

- **UNION** returns all distinct rows.
- **UNION ALL** returns all rows, including duplicates.
- **INTERSECT** returns all rows shared by both queries.
- **MINUS** returns all distinct rows selected by the first query but not by the second.
- **ORDER BY** can appear only at the very end of the statement.

Summary

- The **UNION** operator returns all rows selected by either query. Use the **UNION** operator to return all rows from multiple tables and eliminate any duplicate rows.
- Use the **UNION ALL** operator to return all rows from multiple queries. Unlike with the **UNION** operator, duplicate rows are not eliminated and the output is not sorted by default.
- Use the **INTERSECT** operator to return all rows common to multiple queries.
- Use the **MINUS** operator to return rows returned by the first query that are not present in the second query.
- Remember to use the **ORDER BY** clause only at the very end of the compound statement.
- Make sure that the corresponding expressions in the **SELECT** lists match in number and data type.
Practice 15 Overview

This practice covers the following topics:
• Writing queries using the SET operators
• Discovering alternative join methods

Practice 15 Overview
In this practice, you write queries using the SET operators.
Practice 15

1. List the department IDs for departments that do not contain the job ID ST_CLERK, using SET operators.

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>60</td>
</tr>
<tr>
<td>80</td>
</tr>
<tr>
<td>90</td>
</tr>
<tr>
<td>110</td>
</tr>
<tr>
<td>190</td>
</tr>
</tbody>
</table>

7 rows selected.

2. Display the country ID and the name of the countries that have no departments located in them, using SET operators.

<table>
<thead>
<tr>
<th>CO</th>
<th>COUNTRY_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>DE</td>
<td>Germany</td>
</tr>
</tbody>
</table>

3. Produce a list of jobs for departments 10, 50, and 20, in that order. Display job ID and department ID, using SET operators.

<table>
<thead>
<tr>
<th>JOB_ID</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>AD_ASST</td>
<td>10</td>
</tr>
<tr>
<td>ST_CLERK</td>
<td>50</td>
</tr>
<tr>
<td>ST_MAN</td>
<td>50</td>
</tr>
<tr>
<td>MK_MAN</td>
<td>20</td>
</tr>
<tr>
<td>MK_REP</td>
<td>20</td>
</tr>
</tbody>
</table>

4. List the employee IDs and job IDs of those employees who are currently in the job title that they have held once before during their tenure with the company.

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>JOB_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>176</td>
<td>SA_REP</td>
</tr>
<tr>
<td>200</td>
<td>AD_ASST</td>
</tr>
</tbody>
</table>
5. Write a compound query that lists the following:

- Last names and department ID of all the employees from the EMPLOYEES table, irrespective of the fact whether they belong to any department or not
- Department ID and department name of all the departments from the DEPARTMENTS table, irrespective of the fact whether they have employees working in them or not.

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>DEPARTMENT_ID</th>
<th>TO_CHAR(NULL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abel</td>
<td>80</td>
<td></td>
</tr>
<tr>
<td>Davies</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>De Haan</td>
<td>90</td>
<td></td>
</tr>
<tr>
<td>Ermet</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>Fay</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Gietz</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>Grant</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hartstein</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Higgins</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>Huncll</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>King</td>
<td>90</td>
<td></td>
</tr>
<tr>
<td>Kochhar</td>
<td>90</td>
<td></td>
</tr>
<tr>
<td>Lorentz</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>Matcs</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>Mougos</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>Rajs</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>Taylor</td>
<td>80</td>
<td></td>
</tr>
<tr>
<td>Vargas</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>Whalen</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Zlotkey</td>
<td>80</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>Administration</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>Marketing</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>Shipping</td>
</tr>
<tr>
<td></td>
<td>60</td>
<td>IT</td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>Sales</td>
</tr>
<tr>
<td></td>
<td>90</td>
<td>Executive</td>
</tr>
<tr>
<td></td>
<td>110</td>
<td>Accounting</td>
</tr>
<tr>
<td></td>
<td>190</td>
<td>Contracting</td>
</tr>
</tbody>
</table>

28 rows selected.
Oracle 9i Datetime Functions
Objectives

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

• CURRENT_DATE
• CURRENT_TIMESTAMP
• LOCALTIMESTAMP
• DBTIMEZONE
• SESSIONTIMEZONE
• EXTRACT
• FROM_TZ
• TO_TIMESTAMP
• TO_TIMESTAMP_TZ
• TO_YMINTERVAL
• TZ_OFFSET

Lesson Aim

This lesson addresses some of the datetime functions introduced in Oracle9i.
Time Zones

In Oracle9i, you can include the time zone in your date and time data, as well as provide support for fractional seconds. This lesson focuses on how to manipulate the new datetime data types included with Oracle9i using the new datetime functions. To understand the working of these functions, it is necessary to be familiar with the concept of time zones and Greenwich mean time, or GMT.

The hours of the day are measured by the turning of the earth. The time of day at any particular moment depends on where you are. When it is noon in Greenwich, England, it is midnight along the international date line. The earth is divided into 24 time zones, one for each hour of the day. The time along the prime meridian in Greenwich, England is known as Greenwich mean time, or GMT. GMT is the time standard against which all other time zones in the world are referenced. It is the same all year round and is not effected by summer time or daylight savings time. The meridian line is an imaginary line that runs from the North Pole to the South Pole. It is known as zero longitude and it is the line from which all other lines of longitude are measured. All time is measured relative to Greenwich mean time (GMT) and all places have a latitude (their distance north or south of the equator) and a longitude (their distance east or west of the Greenwich meridian).

Daylight Saving Time

Most western nations advance the clock ahead one hour during the summer months. This period is called daylight saving time. Daylight saving time lasts from the first Sunday in April to the last Sunday in October in the most of the United States, Mexico and Canada. The nations of the European Union observe daylight saving time, but they call it the summer time period. Europe's summer time period begins a week earlier than its North American counterpart, but ends at the same time.
Oracle 9i 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.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Time Zone</th>
<th>Fractional Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>TIMESTAMP (fractional_seconds_precision) WITH TIMEZONE</td>
<td>All values of TIMESTAMP as well as the time zone displacement value which indicates the hours and minutes before or after UTC (Coordinated Universal Time, formerly Greenwich mean time).</td>
<td>fractional_seconds_precision is the number of digits in the fractional part of the SECOND datetime field. Accepted values are 0 to 9. The default is 6.</td>
</tr>
</tbody>
</table>
| TIMESTAMP (fractional_seconds_precision) WITH LOCAL TIME ZONE | All values of TIMESTAMP WITH TIME ZONE, with the following exceptions:  
  - Data is normalized to the database time zone when it is stored in the database.  
  - When the data is retrieved, users see the data in the session time zone. | Yes |
**Oracle 9i Datetime Support (continued)**

TIMESTAMP WITH LOCAL TIME ZONE is stored in the database time zone. When a user selects the data, the value is adjusted to the user's session time zone.

**Example:**

A San Francisco database has system time zone = -8:00. When a New York client (session time zone = -5:00) inserts into or selects from the San Francisco database, TIMESTAMP WITH LOCAL TIME ZONE data is adjusted as follows:

- The New York client inserts TIMESTAMP '1998-1-23 6:00:00-5:00' into a TIMESTAMP WITH LOCAL TIME ZONE column in the San Francisco database. The inserted data is stored in San Francisco as binary value 1998-1-23 3:00:00.
- When the New York client selects that inserted data from the San Francisco database, the value displayed in New York is '1998-1-23 6:00:00'.
- A San Francisco client, selecting the same data, see the value '1998-1-23 3:00:00'.

**Support for Daylight Savings Times**

The Oracle Server automatically determines, for any given time zone region, whether daylight savings is in effect and returns local time values based accordingly. The datetime value is sufficient for the server to determine whether daylight savings time is in effect for a given region in all cases except boundary cases. A boundary case occurs during the period when daylight savings goes into or comes out of effect. For example, in the U.S.-Pacific region, when daylight savings comes into effect, the time changes from 2:00 a.m. to 3:00 a.m. The one hour interval between 2 and 3 a.m. does not exist. When daylight savings goes out of effect, the time changes from 2:00 a.m. back to 1:00 a.m., and the one-hour interval between 1 and 2 a.m. is repeated.

Oracle 9i also significantly reduces the cost of developing and deploying applications globally on a single database instance. Requirements for multigeographic applications include named time zones and multilanguage support through Unicode. The datetime data types TSLTZ and TSTZ are time-zone-aware. Datetime values can be specified as local time in a particular region (rather than a particular offset). Using the time zone rules tables for a given region, the time zone offset for a local time is calculated, taking into consideration daylight savings time adjustments, and used in further operations.

This lesson addresses some of the new datetime functions introduced in Oracle 9i.
CURRENT_DATE

The CURRENT_DATE function returns the current date in the session’s time zone. The return value is a date in the Gregorian calendar.

The examples in the slide illustrate that CURRENT_DATE is sensitive to the session time zone. In the first example, the session is altered to set the TIME_ZONE parameter to –5:0. The TIME_ZONE parameter specifies the default local time zone displacement for the current SQL session. TIME_ZONE is a session parameter only, not an initialization parameter. The TIME_ZONE parameter is set as follows:

\[
\text{TIME\_ZONE} = \ [+ \ | \ -\ ] \ \text{hh}:\text{mm}
\]

The format mask \( [+ \ | \ -\ ] \ \text{hh}:\text{mm} \) indicates the hours and minutes before or after UTC (Coordinated Universal Time, formerly known as Greenwich mean time).

Observe in the output that the value of CURRENT_DATE changes when the TIME_ZONE parameter value is changed to −8:0 in the second example.

\[\text{CURRENT\_DATE is sensitive to the session time zone}\]

\[
\begin{array}{|c|c|}
\hline
\text{SESSIONTIMEZONE} & \text{CURRENT\_DATE} \\
\hline
-05:00 & 07-MAR-2001 03:31:50 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{SESSIONTIMEZONE} & \text{CURRENT\_DATE} \\
\hline
-08:00 & 07-MAR-2001 00:37:52 \\
\hline
\end{array}
\]

\[\text{CURRENT\_DATE}\]

The CURRENT_DATE function returns the current date in the session’s time zone. The return value is a date in the Gregorian calendar.

The examples in the slide illustrate that CURRENT_DATE is sensitive to the session time zone. In the first example, the session is altered to set the TIME_ZONE parameter to –5:0. The TIME_ZONE parameter specifies the default local time zone displacement for the current SQL session. TIME_ZONE is a session parameter only, not an initialization parameter. The TIME_ZONE parameter is set as follows:

\[
\text{TIME\_ZONE} = \ [+ \ | \ -\ ] \ \text{hh}:\text{mm}
\]

The format mask \( [+ \ | \ -\ ] \ \text{hh}:\text{mm} \) indicates the hours and minutes before or after UTC (Coordinated Universal Time, formerly known as Greenwich mean time).

Observe in the output that the value of CURRENT_DATE changes when the TIME_ZONE parameter value is changed to −8:0 in the second example.

Note: The ALTER SESSION command sets the date format of the session to 'DD-MON-YYYY HH24:MI:SS' that is Day of month (1-31)-Abbreviated name of month-4-digit year Hour of day (0-23):Minute (0-59):Second (0-59).
The `CURRENT_TIMESTAMP` function returns the current date and time in the session time zone, as a value of the data type `TIMESTAMP WITH TIME ZONE`. The time zone displacement reflects the current local time of the SQL session. The syntax of the `CURRENT_TIMESTAMP` function is:

```
CURRENT_TIMESTAMP (precision)
```

Where, `precision` is an optional argument that specifies the fractional second precision of the time value returned. If you omit precision, the default is 6.

The examples in the slide illustrate that `CURRENT_TIMESTAMP` is sensitive to the session time zone. In the first example, the session is altered to set the `TIME_ZONE` parameter to –5:0. Observe in the output that the value of `CURRENT_TIMESTAMP` changes when the `TIME_ZONE` parameter value is changed to –8:0 in the second example.
LOCALTIMESTAMP

The LOCALTIMESTAMP function returns the current date and time in the session time zone in a value of data type TIMESTAMP. The difference between this function and CURRENT_TIMESTAMP is that LOCALTIMESTAMP returns a TIMESTAMP value, while CURRENT_TIMESTAMP returns a TIMESTAMP WITH TIME ZONE value. TIMESTAMP WITH TIME ZONE is a variant of TIMESTAMP that includes a time zone displacement in its value. The time zone displacement is the difference (in hours and minutes) between local time and UTC. The TIMESTAMP WITH TIME ZONE data type has the following format:

    TIMESTAMP [ (fractional_seconds_precision) ] WITH TIME ZONE

where fractional_seconds_precision optionally specifies the number of digits in the fractional part of the SECOND datetime field and can be a number in the range 0 to 9. The default is 6. For example, you specify TIMESTAMP WITH TIME ZONE as a literal as follows:

    TIMESTAMP '1997-01-31 09:26:56.66 +02:00'

The syntax of the LOCAL_TIMESTAMP function is:

    LOCAL_TIMESTAMP (TIMESTAMP_precision)

Where, TIMESTAMP_precision is an optional argument that specifies the fractional second precision of the TIMESTAMP value returned.

The examples in the slide illustrates the difference between LOCALTIMESTAMP and CURRENT_TIMESTAMP. Observe that the LOCALTIMESTAMP does not display the time zone value, while the CURRENT_TIMESTAMP does.
**DBTIMEZONE and SESSIONTIMEZONE**

The default database time zone is the same as the operating system's time zone. You set the database's default time zone by specifying the `SET TIME_ZONE` clause of the `CREATE DATABASE` statement. If omitted, the default database time zone is the operating system time zone. The database time zone can be changed for a session with an `ALTER SESSION` statement.

The `DBTIMEZONE` function returns the value of the database time zone. The return type is a time zone offset (a character type in the format `'[+-]TZH:TZM'`) or a time zone region name, depending on how the user specified the database time zone value in the most recent `CREATE DATABASE` or `ALTER DATABASE` statement. The example on the slide shows that the database time zone is set to UTC, as the `TIME_ZONE` parameter is in the format:

```
TIME_ZONE = '[+ | -] hh:mm'
```

The `SESSIONTIMEZONE` function returns the value of the current session’s time zone. The return type is a time zone offset (a character type in the format `'[+-]TZH:TZM'`) or a time zone region name, depending on how the user specified the session time zone value in the most recent `ALTER SESSION` statement. The example in the slide shows that the session time zone is set to UTC.

Observe that the database time zone is different from the current session’s time zone.
The `EXTRACT` expression extracts and returns the value of a specified datetime field from a datetime or interval value expression. You can extract any of the components mentioned in the following syntax using the `EXTRACT` function:

```sql
SELECT EXTRACT ([YEAR] [MONTH] [DAY] [HOUR] [MINUTE] [SECOND]
            [TIMEZONE_HOUR] [TIMEZONE_MINUTE]
            [TIMEZONE_REGION] [TIMEZONE_ABBR]
FROM [datetime_value_expression]
```

When you extract a `TIMEZONE_REGION` or `TIMEZONE_ABBR` (abbreviation), the value returned is a string containing the appropriate time zone name or abbreviation. When you extract any of the other values, the value returned is in the Gregorian calendar. When extracting from a datetime with a time zone value, the value returned is in UTC. For a listing of time zone names and their corresponding abbreviations, query the `V$TIMEZONE_NAMES` dynamic performance view. In the first example on the slide, the `EXTRACT` function is used to extract the `YEAR` from `SYSDATE`.

In the second example in the slide, the `EXTRACT` function is used to extract the `MONTH` from `HIRE_DATE` column of the `EMPLOYEES` table, for those employees who report to the manager whose `EMPLOYEE_ID` is 100.
The `FROM_TZ` function converts a time stamp value to a `TIMESTAMP WITH TIME ZONE` value.

The syntax of the `FROM_TZ` function is as follows:

```
FROM_TZ(timestamp_value, time_zone_value)
```

where `time_zone_value` is a character string in the format `'TZH:TZM'` or a character expression that returns a string in TZR (time zone region) with optional TZD format. TZR represents the time zone region in datetime input strings. Examples are 'Australia/North', 'UTC', and 'Singapore'. TZD represents an abbreviated form of the time zone region with daylight savings information. Examples are 'PST' for US/Pacific standard time and 'PDT' for US/Pacific daylight time. To see a listing of valid values for the TZR and TZD format elements, query the `V$TIMEZONE_NAMES` dynamic performance view.

The example in the slide converts a time stamp value to `TIMESTAMP WITH TIME ZONE`.

```
SELECT FROM_TZ(TIMESTAMP '2000-03-28 08:00:00', '3:00')
FROM DUAL;
```
**TO_TIMESTAMP and TO_TIMESTAMP_TZ**

The `TO_TIMESTAMP` function converts a string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type to a value of `TIMESTAMP` data type. The syntax of the `TO_TIMESTAMP` function is:

```
TO_TIMESTAMP (char, [fmt], ['nlsparam'])
```

The optional `fmt` specifies the format of `char`. If you omit `fmt`, the string must be in the default format of the `TIMESTAMP` data type. The optional `nlsparam` specifies the language in which month and day names and abbreviations are returned. This argument can have this form:

```
'NLS_DATE_LANGUAGE = language'
```

If you omit `nlsparam`, this function uses the default date language for your session. The example on the slide converts a character string to a value of `TIMESTAMP`.

The `TO_TIMESTAMP_TZ` function converts a string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type to a value of `TIMESTAMP WITH TIME ZONE` data type. The syntax of the `TO_TIMESTAMP_TZ` function is:

```
TO_TIMESTAMP_TZ (char, [fmt], ['nlsparam'])
```

The optional `fmt` specifies the format of `char`. If omitted, a string must be in the default format of the `TIMESTAMP WITH TIME ZONE` data type. The optional `nlsparam` has the same purpose in this function as in the `TO_TIMESTAMP` function. The example in the slide converts a character string to a value of `TIMESTAMP WITH TIME ZONE`.

**Note:** The `TO_TIMESTAMP_TZ` function does not convert character strings to `TIMESTAMP WITH LOCAL TIME ZONE`.

Introduction to Oracle9i: SQL 16-12
**TO_YMINTERVAL**

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

<table>
<thead>
<tr>
<th>HIRE_DATE</th>
<th>HIRE_DATE_YMINTERVAL</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>

**TO_YMINTERVAL**

The `TO_YMINTERVAL` function converts a character string of `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2` data type to an `INTERVAL YEAR TO MONTH` data type. The `INTERVAL YEAR TO MONTH` data type stores a period of time using the `YEAR` and `MONTH` datetime fields. The format of `INTERVAL YEAR TO MONTH` is as follows:

```
INTERVAL YEAR [(year_precision)] TO MONTH
```

where `year_precision` is the number of digits in the `YEAR` datetime field. The default value of `year_precision` is 2.

The syntax of the `TO_YMINTERVAL` function is:

```
TO_YMINTERVAL (char)
```

where `char` is the character string to be converted.

The example in the slide calculates a date that is one year two months after the hire date for the employees working in the department 20 of the `EMPLOYEES` table.

A reverse calculation can also be done using the `TO_YMINTERVAL` function. For example:

```
SELECT hire_date, hire_date + TO_YMINTERVAL('-02-04') AS HIRE_DATE_YMINTERVAL
FROM EMPLOYEES WHERE department_id = 20;
```

Observe that the character string passed to the `TO_YMINTERVAL` function has a negative value. The example returns a date that is two years and four months before the hire date for the employees working in the department 20 of the `EMPLOYEES` table.

*Introduction to Oracle9i: SQL 16-13*
The `TZ_OFFSET` function returns the time zone offset corresponding to the value entered. The return value is dependent on the date when the statement is executed. For example, if the `TZ_OFFSET` function returns a value of -08:00, the return value can be interpreted as the time zone from where the command was executed is eight hours after UTC. You can enter a valid time zone name, a time zone offset from UTC (which simply returns itself), or the keyword `SESSIONTIMEZONE` or `DBTIMEZONE`. The syntax of the `TZ_OFFSET` function is:

```
TZ_OFFSET ( ['time_zone_name'] ['+ | -'] hh:mm
[ SESSIONTIMEZONE] [DBTIMEZONE])
```

The examples in the slide can be interpreted as follows:

- The time zone 'US/Eastern' is five hours behind UTC
- The time zone 'Canada/Yukon' is eight hours behind UTC
- The time zone 'Europe/London' is in the UTC

For a listing of valid time zone name values, query the `V$TIMEZONE_NAMES` dynamic performance view.

```
DESC V$TIMEZONE_NAMES
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Null?</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>TZNAME</td>
<td></td>
<td>VARCHAR2(4)</td>
</tr>
<tr>
<td>TZABBREV</td>
<td></td>
<td>VARCHAR2(4)</td>
</tr>
</tbody>
</table>
### TZ_OFFSET (continued)

```sql
SELECT * FROM V$TIMEZONE_NAMES;
```

<table>
<thead>
<tr>
<th>TZNAME</th>
<th>TZABBREV</th>
</tr>
</thead>
<tbody>
<tr>
<td>Africa/Cairo</td>
<td>LMT</td>
</tr>
<tr>
<td>Africa/Cairo</td>
<td>EET</td>
</tr>
<tr>
<td>Africa/Cairo</td>
<td>EEST</td>
</tr>
<tr>
<td>Africa/Tripoli</td>
<td>LMT</td>
</tr>
<tr>
<td>Africa/Tripoli</td>
<td>CET</td>
</tr>
<tr>
<td>Africa/Tripoli</td>
<td>CEST</td>
</tr>
<tr>
<td>Africa/Tripoli</td>
<td>EET</td>
</tr>
<tr>
<td>America/Adak</td>
<td>LMT</td>
</tr>
<tr>
<td>America/Adak</td>
<td>NST</td>
</tr>
<tr>
<td>America/Adak</td>
<td>NAT</td>
</tr>
<tr>
<td>America/Adak</td>
<td>BST</td>
</tr>
<tr>
<td>America/Adak</td>
<td>BDT</td>
</tr>
<tr>
<td>America/Adak</td>
<td>HAST</td>
</tr>
</tbody>
</table>

| US/Samoa  | BST     |
| US/Samoa  | SST     |
| W-SU      | LMT     |
| W-SU      | MMT     |
| W-SU      | MST     |
| W-SU      | MDT     |
| W-SU      | S       |
| W-SU      | MSD     |
| W-SU      | MSK     |
| W-SU      | EET     |
| W-SU      | EEST    |
| WET       | WEST    |
| WET       | WET     |

616 rows selected.
Summary

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

- FROM_TZ
- TO_TIMESTAMP
- TO_TIMESTAMP_TZ
- TO_YMINTERVAL
- TZ_OFFSET
- CURRENT_DATE
- CURRENT_TIMESTAMP
- LOCALTIMESTAMP
- DBTIMEZONE
- SESSIONTIMEZONE
- EXTRACT

This lesson addressed some of the new datetime functions introduced in Oracle9i.
Practice 16 Overview

This practice covers using the Oracle9i datetime functions.

Practice 16 Overview

In this practice, you display time zone offsets, CURRENT_DATE, CURRENT_TIMESTAMP, and the LOCALTIMESTAMP. You also set time zones and use the EXTRACT function.
Practice 16

1. Alter the session to set the NLS_DATE_FORMAT to DD-MON-YYYY HH24:MI:SS.

2. a. Write queries to display the time zone offsets (TZ_OFFSET), for the following time zones.

   - US/Pacific-New

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

   - Singapore

<table>
<thead>
<tr>
<th>TZ_OFFSET</th>
</tr>
</thead>
<tbody>
<tr>
<td>+08:00</td>
</tr>
</tbody>
</table>

   - Egypt

<table>
<thead>
<tr>
<th>TZ_OFFSET</th>
</tr>
</thead>
<tbody>
<tr>
<td>+02:00</td>
</tr>
</tbody>
</table>

   b. Alter the session to set the TIME_ZONE parameter value to the time zone offset of US/Pacific-New.

   c. Display the CURRENT_DATE, CURRENT_TIMESTAMP, and LOCALTIMESTAMP for this session.

   **Note:** The output might be different based on the date when the command is executed.

<table>
<thead>
<tr>
<th>CURRENT_DATE</th>
<th>CURRENT_TIMESTAMP</th>
<th>LOCALTIMESTAMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>07-MAR-2001 01:45:13</td>
<td>07-MAR-01 01:45.12.931393 AM -08:00</td>
<td>07-MAR-01 01:45.12.931393 AM</td>
</tr>
</tbody>
</table>

   d. Alter the session to set the TIME_ZONE parameter value to the time zone offset of Singapore.

   e. Display the CURRENT_DATE, CURRENT_TIMESTAMP, and LOCALTIMESTAMP for this session. Note: The output might be different based on the date when the command is executed.

<table>
<thead>
<tr>
<th>CURRENT_DATE</th>
<th>CURRENT_TIMESTAMP</th>
<th>LOCALTIMESTAMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>07-MAR-2001 17:46:35</td>
<td>07-MAR-01 05:46.34.628618 PM +08:00</td>
<td>07-MAR-01 05:46.34.628618 PM</td>
</tr>
</tbody>
</table>

   **Note:** Observe in the preceding practice that CURRENT_DATE, CURRENT_TIMESTAMP, and LOCALTIMESTAMP are all sensitive to the session time zone.

3. Write a query to display the DBTIMEZONE and SESSIONTIMEZONE.

<table>
<thead>
<tr>
<th>DBTIMEZONE</th>
<th>SESSIONTIMEZONE</th>
</tr>
</thead>
<tbody>
<tr>
<td>+05:30</td>
<td>+08:00</td>
</tr>
</tbody>
</table>
Practice 16 (continued)

4. Write a query to extract the YEAR from HIRE_DATE column of the EMPLOYEES table for those employees who work in department 80.

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>EXTRACT(YEARFROMHIRE_DATE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zlotkey</td>
<td>2000</td>
</tr>
<tr>
<td>Abel</td>
<td>1996</td>
</tr>
<tr>
<td>Taylor</td>
<td>1996</td>
</tr>
</tbody>
</table>
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

Lesson Aim

In this lesson you learn how to:

• Group data for obtaining the following:
  – Subtotal values by using the **ROLLUP** operator
  – Cross-tabulation values by using the **CUBE** operator
• Use the **GROUPING** function to identify the level of aggregation in the results set produced by a **ROLLUP** or **CUBE** operator.
• Use **GROUPING SETS** to produce a single result set that is equivalent to a **UNION ALL** approach
Review of Group Functions

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

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

Example:
```
SELECT AVG(salary), STDDEV(salary),
       COUNT(commission_pct), MAX(hire_date)
FROM employees
WHERE job_id LIKE 'SA%';
```

Group Functions

You can use the GROUP BY clause to divide the rows in a table into groups. You can then use the group functions to return summary information for each group. Group functions can appear in select lists and in ORDER BY and HAVING clauses. The Oracle Server applies the group functions to each group of rows and returns a single result row for each group.

Types of Group Functions

Each of the group functions AVG, SUM, MAX, MIN, COUNT, STDDEV, and VARIANCE accept one argument. The functions AVG, SUM, STDDEV, and VARIANCE operate only on numeric values. MAX and MIN can operate on numeric, character, or date data values. COUNT returns the number of nonnull rows for the given expression. The example in the slide calculates the average salary, standard deviation on the salary, number of employees earning a commission and the maximum hire date for those employees whose JOB_ID begins with SA.

Guidelines for Using Group Functions

- The data types for the arguments can be CHAR, VARCHAR2, NUMBER, or DATE.
- All group functions except COUNT(*) ignore null values. To substitute a value for null values, use the NVL function. COUNT returns either a number or zero.
- The Oracle Server implicitly sorts the result set in ascending order of the grouping columns specified, when you use a GROUP BY clause. To override this default ordering, you can use DESC in an ORDER BY clause.
Review of the \texttt{GROUP BY} Clause

Syntax:

\begin{verbatim}
SELECT \[column,\] \text{group} function\(column\). . .
FROM \text{table}
\[\text{WHERE} \ \text{condition}\]
\[\text{GROUP BY} \ \text{group}_\text{by}_\text{expression}\]
\[\text{ORDER BY} \ \text{column};\]
\end{verbatim}

Example:

\begin{verbatim}
SELECT department_id, job_id, \text{SUM(salary)}, \text{COUNT(employee_id)}
FROM employees
\text{GROUP BY department_id, job_id;}
\end{verbatim}

The example illustrated in the slide is evaluated by the Oracle Server as follows:

- The \texttt{SELECT} clause specifies that the following columns are to be retrieved:
  - Department ID and job ID columns from the \texttt{EMPLOYEES} table
  - The sum of all the salaries and the number of employees in each group that you have specified in the \texttt{GROUP BY} clause

- The \texttt{GROUP BY} clause specifies how the rows should be grouped in the table. The total salary and the number of employees are calculated for each job ID within each department. The rows are grouped by department ID and then grouped by job within each department.

\begin{tabular}{|c|c|c|c|}
\hline
DEPARTMENT ID & JOB_ID & SUM(SALARY) & COUNT(EMPLOYEE_ID) \\
\hline
10 & AD_ASSIST & 4400 & 1 \\
20 & MK_MAN & 13000 & 1 \\
20 & MK_REP & 6000 & 1 \\
50 & ST_CLERK & 11700 & 4 \\
\hline
90 & AD_VP & 34000 & 2 \\
110 & AC_ACCOUNT & 8300 & 1 \\
110 & AC_MGR & 12000 & 1 \\
& SA_REP & 7000 & 1 \\
\hline
\end{tabular}

13 rows selected.
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];
```

The **HAVING** Clause

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

The Oracle Server performs the following steps when you use the **HAVING** clause:

1. Groups rows
2. Applies the group functions to the groups and displays the groups that match the criteria in the **HAVING** clause

```sql
SELECT department_id, AVG(salary)
FROM employees
GROUP BY department_id
HAVING AVG(salary) >9500;
```

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>AVG(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>10033.333</td>
</tr>
<tr>
<td>90</td>
<td>19333.333</td>
</tr>
<tr>
<td>110</td>
<td>10150</td>
</tr>
</tbody>
</table>

The example displays department ID and average salary for those departments whose average salary is greater than $9,500.
GROUP BY with ROLLUP and CUBE Operators

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

GROUP BY with the ROLLUP and CUBE Operators

You specify ROLLUP and CUBE operators in the GROUP BY clause of a query. ROLLUP grouping produces a results set containing the regular grouped rows and subtotal rows. The CUBE operation in the GROUP BY clause groups the selected rows based on the values of all possible combinations of expressions in the specification and returns a single row of summary information for each group. You can use the CUBE operator to produce cross-tabulation rows.

Note: When working with ROLLUP and CUBE, make sure that the columns following the GROUP BY clause have meaningful, real-life relationships with each other; otherwise the operators return irrelevant information.

The ROLLUP and CUBE operators are available only in Oracle8i and later releases.
The ROLLUP Operator

The ROLLUP operator delivers aggregates and superaggregates for expressions within a GROUP BY statement. The ROLLUP operator can be used by report writers to extract statistics and summary information from results sets. The cumulative aggregates can be used in reports, charts, and graphs.

The ROLLUP operator creates groupings by moving in one direction, from right to left, along the list of columns specified in the GROUP BY clause. It then applies the aggregate function to these groupings.

Note: To produce subtotals in \( n \) dimensions (that is, \( n \) columns in the GROUP BY clause) without a ROLLUP operator, \( n+1 \) SELECT statements must be linked with UNION ALL. This makes the query execution inefficient, because each of the SELECT statements causes table access. The ROLLUP operator gathers its results with just one table access. The ROLLUP operator is useful if there are many columns involved in producing the subtotals.
Example of a ROLLUP Operator

In the example in the slide:

- Total salaries for every job ID within a department for those departments whose department ID is less than 60 are displayed by the GROUP BY clause (labeled 1)
- The ROLLUP operator displays:
  - Total salary for those departments whose department ID is less than 60 (labeled 2)
  - Total salary for all departments whose department ID is less than 60, irrespective of the job IDs (labeled 3)
- All rows indicated as 1 are regular rows and all rows indicated as 2 and 3 are superaggregate rows.

The ROLLUP operator creates subtotals that roll up from the most detailed level to a grand total, following the grouping list specified in the GROUP BY clause. First it calculates the standard aggregate values for the groups specified in the GROUP BY clause (in the example, the sum of salaries grouped on each job within a department). Then it creates progressively higher-level subtotals, moving from right to left through the list of grouping columns. (In the preceding example, the sum of salaries for each department is calculated, followed by the sum of salaries for all departments.)

- Given $n$ expressions in the ROLLUP operator of the GROUP BY clause, the operation results in $n + 1 = 2 + 1 = 3$ groupings.
- Rows based on the values of the first $n$ expressions are called rows or regular rows and the others are called superaggregate rows.

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

The CUBE Operator

The CUBE operator is an additional switch in the GROUP BY clause in a SELECT statement. The CUBE operator can be applied to all aggregate functions, including AVG, SUM, MAX, MIN, and COUNT. It is used to produce result sets that are typically used for cross-tabular reports. While ROLLUP produces only a fraction of possible subtotal combinations, CUBE produces subtotals for all possible combinations of groupings specified in the GROUP BY clause, and a grand total.

The CUBE operator is used with an aggregate function to generate additional rows in a results set. Columns included in the GROUP BY clause are cross-referenced to produce a superset of groups. The aggregate function specified in the select list is applied to these groups to produce summary values for the additional superaggregate rows. The number of extra groups in the results set is determined by the number of columns included in the GROUP BY clause.

In fact, every possible combination of the columns or expressions in the GROUP BY clause is used to produce superaggregates. If you have \( n \) columns or expressions in the GROUP BY clause, there will be \( 2^n \) possible superaggregate combinations. Mathematically, these combinations form an \( n \)-dimensional cube, which is how the operator got its name.

By using application or programming tools, these superaggregate values can then be fed into charts and graphs that convey results and relationships visually and effectively.
### Example of a CUBE Operator

The output of the SELECT statement in the example can be interpreted as follows:

- The total salary for every job within a department (for those departments whose department ID is less than 50) is displayed by the GROUP BY clause (labeled 1).
- The total salary for those departments whose department ID is less than 50 (labeled 2).
- The total salary for every job irrespective of the department (labeled 3).
- Total salary for those departments whose department ID is less than 50, irrespective of the job titles (labeled 4).

In the preceding example, all rows indicated as 1 are regular rows, all rows indicated as 2 and 4 are superaggregate rows, and all rows indicated as 3 are cross-tabulation values.

The CUBE operator has also performed the ROLLUP operation to display the subtotals for those departments whose department ID is less than 50 and the total salary for those departments whose department ID is less than 50, irrespective of the job titles. Additionally, the CUBE operator displays the total salary for every job irrespective of the department.

**Note:** Similar to the ROLLUP operator, producing subtotals in $n$ dimensions (that is, $n$ columns in the GROUP BY clause) without a CUBE operator requires $2^n$ SELECT statements to be linked with UNION ALL. Thus, a report with three dimensions requires $2^3 = 8$ SELECT statements to be linked with UNION ALL.

---

**CUBE Operator: Example**

```sql
SELECT department_id, job_id, SUM(salary)
FROM employees
WHERE department_id < 60
GROUP BY CUBE (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>A_D_ASST</td>
<td>4400</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>4400</td>
</tr>
<tr>
<td>20</td>
<td>M_K_MAN</td>
<td>13000</td>
</tr>
<tr>
<td>20</td>
<td>M_K_REP</td>
<td>6000</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>13000</td>
</tr>
<tr>
<td>50</td>
<td>S_T_CLERK</td>
<td>11700</td>
</tr>
<tr>
<td>60</td>
<td>S_T_MAN</td>
<td>5800</td>
</tr>
<tr>
<td>50</td>
<td></td>
<td>17500</td>
</tr>
<tr>
<td>A_D_ASST</td>
<td></td>
<td>4400</td>
</tr>
<tr>
<td>M_K_MAN</td>
<td></td>
<td>13000</td>
</tr>
<tr>
<td>M_K_REP</td>
<td></td>
<td>6000</td>
</tr>
<tr>
<td>S_T_CLERK</td>
<td></td>
<td>11700</td>
</tr>
<tr>
<td>S_T_MAN</td>
<td></td>
<td>5800</td>
</tr>
</tbody>
</table>

14 rows selected.
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 it, you can find the groups forming the subtotal in a row.
• Using it, you can differentiate stored NULL values from NULL values created by ROLLUP or CUBE.
• It returns 0 or 1.

The GROUPING Function

The GROUPING function can be used with either the CUBE or ROLLUP operator to help you understand how a summary value has been obtained.

The GROUPING function uses a single column as its argument. The expr in the GROUPING function must match one of the expressions in the GROUP BY clause. The function returns a value of 0 or 1.

The values returned by the GROUPING function are useful to:

• Determine the level of aggregation of a given subtotal; that is, the group or groups on which the subtotal is based
• Identify whether a NULL value in the expression column of a row of the result set indicates:
  – A NULL value from the base table (stored NULL value)
  – A NULL value created by ROLLUP/CUBE (as a result of a group function on that expression)

A value of 0 returned by the GROUPING function based on an expression indicates one of the following:

• The expression has been used to calculate the aggregate value.
• The NULL value in the expression column is a stored NULL value.

A value of 1 returned by the GROUPING function based on an expression indicates one of the following:

• The expression has not been used to calculate the aggregate value.
• The NULL value in the expression column is created by ROLLUP or CUBE as a result of grouping.
Example of a GROUPING Function

In the example in the slide, consider the summary value 4400 in the first row. This summary value is the total salary for the job ID of AD_ASST within department 10. To calculate this summary value, both the columns DEPARTMENT_ID and JOB_ID have been taken into account. Thus a value of 0 is returned for both the expressions GROUPING(department_id) and GROUPING(job_id).

Consider the summary value 4400 in the second row. This value is the total salary for department 10 and has been calculated by taking into account the column DEPARTMENT_ID; thus a value of 0 has been returned by GROUPING(department_id). Because the column JOB_ID has not been taken into account to calculate this value, a value of 1 has been returned for GROUPING(job_id). You can observe similar output in the fifth row.

In the last row, consider the summary value 23400. This is the total salary for those departments whose department ID is less than 50 and all job titles. To calculate this summary value, neither of the columns DEPARTMENT_ID and JOB_ID have been taken into account. Thus a value of 1 is returned for both the expressions GROUPING(department_id) and GROUPING(job_id).
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 higher the performance benefit is.

A single SELECT statement can now be written using GROUPING SETS to specify various groupings (that can also include ROLLUP or CUBE operators), rather than multiple SELECT statements combined by UNION ALL operators. For example, you can say:

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

This statement calculates aggregates over three groupings:

- (department_id, job_id, manager_id), (department_id, manager_id)
- and (job_id, manager_id)

Without this enhancement in Oracle9i, multiple queries combined together with UNION ALL are required to get the output of the preceding SELECT statement. A multiquery approach is inefficient, for it requires multiple scans of the same data.
GROUPING SETS (continued)

Compare the preceding statement with this alternative:

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

The preceding statement computes all the 8 (2 * 2 * 2) groupings, though only the groups
(department_id, job_id, manager_id), (department_id, manager_id) and
(job_id, manager_id) are of interest to you.

Another alternative is the following statement:

```sql
SELECT department_id, job_id, manager_id, AVG(salary)
FROM employees
GROUP BY department_id, job_id, manager_id
UNION ALL
SELECT department_id, NULL, manager_id, AVG(salary)
FROM employees
GROUP BY department_id, manager_id
UNION ALL
SELECT NULL, job_id, manager_id, AVG(salary)
FROM employees
GROUP BY job_id, manager_id;
```

This statement requires three scans of the base table, making it inefficient.

CUBE and ROLLUP can be thought of as grouping sets with very specific semantics. The following
equivalencies show this fact:

<table>
<thead>
<tr>
<th>CUBE(a, b, c) is equivalent to</th>
<th>GROUPING SETS ((a, b, c), (a, b), (a, c), (b, c), (a), (b), (c), ())</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROLLUP(a, b, c) is equivalent to</td>
<td>GROUPING SETS ((a, b, c), (a, b), (a), ())</td>
</tr>
</tbody>
</table>
GROUPING SETS: Example

The query in the slide calculates aggregates over two groupings. The table is divided into the following groups:

- Department ID, Job ID
- Job ID, Manager ID

The average salaries for each of these groups are calculated. The results set displays average salary for each of the two groups.

In the output, the group marked as 1 can be interpreted as:

- The average salary of all employees with the job ID `AD_ASST` in the department 10 is 4400.
- The average salary of all employees with the job ID `MK_MAN` in the department 20 is 13000.
- The average salary of all employees with the job ID `MK_REP` in the department 20 is 6000.
- The average salary of all employees with the job ID `ST_CLERK` in the department 50 is 2925 and so on.
GROUPING SETS: Example (continued)

The group marked as 2 in the output is interpreted as:

- The average salary of all employees with the job ID MK_MAN, who report to the manager with the manager ID 100, is 13000.
- The average salary of all employees with the job ID MK_REP, who report to the manager with the manager ID 201, is 6000, and so on.

The example in the slide can also be written as:

```
SELECT department_id, job_id, NULL as manager_id,
       AVG(salary) as AVGSAL
FROM employees
GROUP BY department_id, job_id
UNION ALL
SELECT NULL, job_id, manager_id, avg(salary) as AVGSAL
FROM employees
GROUP BY job_id, manager_id;
```

In the absence of an optimizer that looks across query blocks to generate the execution plan, the preceding query would need two scans of the base table, EMPLOYEES. This could be very inefficient. Hence the usage of the GROUPING SETS statement is recommended.
A composite column is a collection of columns that are treated as a unit.

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

To specify composite columns, in the GROUP BY clause you 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.
Composite Columns (continued)

Compare this with the normal ROLLUP as in:

```sql
GROUP BY ROLLUP(a, b, c)
```

which would be

```sql
GROUP BY a, b, c UNION ALL
GROUP BY a, b UNION ALL
GROUP BY a UNION ALL
GROUP BY ()
```

Similarly,

```sql
GROUP BY CUBE((a, b), c)
```

would be equivalent to

```sql
GROUP BY a, b, c UNION ALL
GROUP BY a, b UNION ALL
GROUP BY c UNION ALL
GROUP BY ()
```

The following table shows grouping sets specification and equivalent GROUP BY specification.

<table>
<thead>
<tr>
<th>GROUPING SETS Statements</th>
<th>Equivalent GROUP BY Statements</th>
</tr>
</thead>
</table>
| GROUP BY GROUPING SETS(a, b, c)                   | GROUP BY a UNION ALL
|                                                   | GROUP BY b UNION ALL
|                                                   | GROUP BY c                                          |
| GROUP BY GROUPING SETS(a, b, (b, c)) (The GROUPING SETS expression has a composite column) | GROUP BY a UNION ALL
|                                                   | GROUP BY b UNION ALL
|                                                   | GROUP BY b, c                                      |
| GROUP BY GROUPING SETS((a, b, c))                 | GROUP BY a, b, c                                   |
| GROUP BY GROUPING SETS(a, (b), ())                | GROUP BY a UNION ALL
|                                                   | GROUP BY b UNION ALL
|                                                   | GROUP BY ()                                         |
| GROUP BY GROUPING SETS(a,ROLLUP(b, c)) (The GROUPING SETS expression has a composite column) | GROUP BY a UNION ALL
|                                                   | GROUP BY ROLLUP(b, c)                              |
Composite Columns: Example

Consider the example:

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

The preceding query results in the Oracle Server computing the following groupings:

1. (department_id, job_id, manager_id)
2. (department_id, job_id)
3. (department_id)
4. ( )

If you are just interested in grouping of lines (1), (3), and (4) in the preceding example, you cannot limit the calculation to those groupings without using composite columns. With composite columns, this is possible by treating JOB_ID and MANAGER_ID columns as a single unit while rolling up. Columns enclosed in parentheses are treated as a unit while computing ROLLUP and CUBE. This is illustrated in the example on the slide. By enclosing JOB_ID and MANAGER_ID columns in parenthesis, we indicate to the Oracle Server to treat JOB_ID and MANAGER_ID as a single unit, as a composite column.
Composite Columns Example (continued)

The example in the slide computes the following groupings:

- (department_id, job_id, manager_id)
- (department_id)
- ()

The example in the slide displays the following:

- Total salary for every department (labeled 1)
- Total salary for every department, job ID, and manager (labeled 2)
- Grand total (labeled 3)

The example in the slide can also be written as:

```sql
SELECT department_id, job_id, manager_id, SUM(salary)
FROM employees
GROUP BY department_id, job_id, manager_id
UNION ALL
SELECT department_id, TO_CHAR(NULL), TO_NUMBER(NULL), SUM(salary)
FROM employees
GROUP BY department_id
UNION ALL
SELECT TO_NUMBER(NULL), TO_CHAR(NULL), TO_NUMBER(NULL), SUM(salary)
FROM employees
GROUP BY ()
```

In the absence of an optimizer that looks across query blocks to generate the execution plan, the preceding query would need three scans of the base table, EMPLOYEES. This could be very inefficient. Hence, the use of composite columns is recommended.
Concatenated Groupings

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

```
GROUP BY GROUPING SETS(a, b), GROUPING SETS(c, d)
```

Concatenated Columns

Concatenated groupings offer a concise way to generate useful combinations of groupings. The concatenated groupings are specified simply by listing multiple grouping sets, cubes, and rollups, and separating them with commas. Here is an example of concatenated grouping sets:

```
GROUP BY GROUPING SETS(a, b), GROUPING SETS(c, d)
```

The preceding SQL defines the following groupings:

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

Concatenation of grouping sets is very helpful for these reasons:

- Ease of query development: you need not enumerate all groupings manually
- Use by applications: SQL generated by OLAP applications often involves concatenation of grouping sets, with each grouping set defining groupings needed for a dimension
Concatenated Groupings Example

The example in the slide results in the following groupings:

- (department_id, manager_id, job_id)
- (department_id, manager_id)
- (department_id, job_id)
- (department_id)

The total salary for each of these groups is calculated.

The example in the slide displays the following:

- Total salary for every department, job ID, manager (labeled 1)
- Total salary for every department, manager ID (labeled 2)
- Total salary for every department, job ID (labeled 3)
- Total salary for every department (labeled 4)

For easier understanding, the details for the department 10 are highlighted in the output.
Summary

In this lesson, you should have learned how to:
• Use the ROLLUP operation to produce subtotal values
• Use the CUBE operation to produce cross-tabulation values
• Use the GROUPING function to identify the row values created by ROLLUP or CUBE
• Use the GROUPING SETS syntax to define multiple groupings in the same query.
• Use the GROUP BY clause, to combine expressions in various ways:
  – Composite columns
  – Concatenated grouping sets

Summary

• ROLLUP and CUBE are extensions of the GROUP BY clause.
• ROLLUP is used to display subtotal and grand total values.
• CUBE is used to display cross-tabulation values.
• The GROUPING function helps you determine whether a row is an aggregate produced by a CUBE or ROLLUP operator.
• With the GROUPING SETS syntax, you can define multiple groupings in the same query. GROUP BY computes all the groupings specified and combines them with UNION ALL.
• Within the GROUP BY clause, you can combine expressions in various ways:
  – To specify composite columns, you group columns within parentheses so that the Oracle Server treats them as a unit while computing ROLLUP or CUBE operations.
  – To specify concatenated grouping sets, you separate multiple grouping sets, ROLLUP, and CUBE operations with commas so that the Oracle Server combines them into a single GROUP BY clause. The result is a cross-product of groupings from each grouping set.
Practice 17 Overview

This practice covers the following topics:
• Using the ROLLUP operator
• Using the CUBE operator
• Using the GROUPING function
• Using GROUPING SETS

Practice 17 Overview
In this practice, you use the ROLLUP and CUBE operators as extensions of the GROUP BY clause. You will also use GROUPING SETS.
Practice 17

1. Write a query to display the following for those employees whose manager ID is less than 120:
   - Manager ID
   - Job ID and total salary for every job ID for employees who report to the same manager
   - Total salary of those managers
   - Total salary of those managers, irrespective of the job IDs

<table>
<thead>
<tr>
<th>MANAGER_ID</th>
<th>JOB_ID</th>
<th>SUM(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>AD_VP</td>
<td>34000</td>
</tr>
<tr>
<td>100</td>
<td>MK_MAN</td>
<td>13000</td>
</tr>
<tr>
<td>100</td>
<td>SA_MAN</td>
<td>10500</td>
</tr>
<tr>
<td>100</td>
<td>ST_MAN</td>
<td>5900</td>
</tr>
<tr>
<td>100</td>
<td></td>
<td>63300</td>
</tr>
<tr>
<td>101</td>
<td>AC_MGR</td>
<td>12000</td>
</tr>
<tr>
<td>101</td>
<td>AD_ASST</td>
<td>4400</td>
</tr>
<tr>
<td>101</td>
<td></td>
<td>16400</td>
</tr>
<tr>
<td>102</td>
<td>IT_FREG</td>
<td>9000</td>
</tr>
<tr>
<td>102</td>
<td></td>
<td>9000</td>
</tr>
<tr>
<td>103</td>
<td>IT_FREG</td>
<td>10200</td>
</tr>
<tr>
<td>103</td>
<td></td>
<td>10200</td>
</tr>
<tr>
<td></td>
<td></td>
<td>98900</td>
</tr>
</tbody>
</table>

13 rows selected.
2. Observe the output from question 1. Write a query using the GROUPING function to determine whether the NULL values in the columns corresponding to the GROUP BY expressions are caused by the ROLLUP operation.

```
<table>
<thead>
<tr>
<th>MGR</th>
<th>JOB</th>
<th>SUM(SALARY)</th>
<th>GROUPING(MANAGER_ID)</th>
<th>GROUPING(JOB_ID)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>AD_VP</td>
<td>34000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>MK_MAN</td>
<td>13000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>SA_MAN</td>
<td>10500</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>ST_MAN</td>
<td>5800</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td></td>
<td>63300</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>101</td>
<td>AC_MGR</td>
<td>12000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>101</td>
<td>AD_ASST</td>
<td>4400</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>101</td>
<td></td>
<td>16400</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>102</td>
<td>IT_PROG</td>
<td>9000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>102</td>
<td></td>
<td>9000</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>103</td>
<td>IT_PROG</td>
<td>10200</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>103</td>
<td></td>
<td>10200</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>98900</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

13 rows selected.
Practice 17 (continued)

3. Write a query to display the following for those employees whose manager ID is less than 120:
   - Manager ID
   - Job and total salaries for every job for employees who report to the same manager
   - Total salary of those managers
   - Cross-tabulation values to display the total salary for every job, irrespective of the manager
   - Total salary irrespective of all job titles

<table>
<thead>
<tr>
<th>MANAGER_ID</th>
<th>JOB_ID</th>
<th>SUM(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>AD_VP</td>
<td>34000</td>
</tr>
<tr>
<td>100</td>
<td>MK_MAN</td>
<td>13000</td>
</tr>
<tr>
<td>100</td>
<td>SA_MAN</td>
<td>10600</td>
</tr>
<tr>
<td>100</td>
<td>ST_MAN</td>
<td>5300</td>
</tr>
<tr>
<td>100</td>
<td></td>
<td>63300</td>
</tr>
<tr>
<td>101</td>
<td>AC_MGR</td>
<td>12000</td>
</tr>
<tr>
<td>101</td>
<td>AD_ASST</td>
<td>4400</td>
</tr>
<tr>
<td>101</td>
<td></td>
<td>16400</td>
</tr>
<tr>
<td>102</td>
<td>IT_PROG</td>
<td>9000</td>
</tr>
<tr>
<td>102</td>
<td></td>
<td>9000</td>
</tr>
<tr>
<td>103</td>
<td>IT_PROG</td>
<td>10200</td>
</tr>
<tr>
<td>103</td>
<td></td>
<td>10200</td>
</tr>
<tr>
<td></td>
<td>AC_MGR</td>
<td>12000</td>
</tr>
<tr>
<td></td>
<td>AD_ASST</td>
<td>4400</td>
</tr>
<tr>
<td></td>
<td>AD_VP</td>
<td>34000</td>
</tr>
<tr>
<td></td>
<td>IT_PROG</td>
<td>19200</td>
</tr>
<tr>
<td></td>
<td>MK_MAN</td>
<td>13000</td>
</tr>
<tr>
<td></td>
<td>SA_MAN</td>
<td>10500</td>
</tr>
<tr>
<td></td>
<td>ST_MAN</td>
<td>5300</td>
</tr>
<tr>
<td></td>
<td></td>
<td>99900</td>
</tr>
</tbody>
</table>

20 rows selected.
4. Observe the output from question 3. Write a query using the `GROUPING` function to determine whether the `NULL` values in the columns corresponding to the `GROUP BY` expressions are caused by the `CUBE` operation.

<table>
<thead>
<tr>
<th>MGR</th>
<th>JOB</th>
<th>SUM(SALARY)</th>
<th>GROUPING(MANAGER_ID)</th>
<th>GROUPING(JOB_ID)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>AD_VP</td>
<td>34000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>MK_MAN</td>
<td>12000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>SA_MAN</td>
<td>10500</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>ST_MAN</td>
<td>6800</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td></td>
<td>63300</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>101</td>
<td>AC_MGR</td>
<td>12000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>101</td>
<td>AD_ASST</td>
<td>4400</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>101</td>
<td></td>
<td>16400</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>102</td>
<td>IT_PRG</td>
<td>9000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>102</td>
<td></td>
<td>9000</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>103</td>
<td>IT_PRG</td>
<td>10200</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>103</td>
<td></td>
<td>10200</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>AC_MGR</td>
<td>12000</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>AD_ASST</td>
<td>4400</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>AD_VP</td>
<td>34000</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>IT_PRG</td>
<td>19200</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>MK_MAN</td>
<td>13000</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>SA_MAN</td>
<td>10600</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>ST_MAN</td>
<td>5800</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>96900</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

20 rows selected.
Practice 17 (continued)

5. Using GROUPING SETS, write a query to display the following groupings:
   - department_id, manager_id, job_id
   - department_id, job_id
   - manager_id, job_id

The query should calculate the sum of the salaries for each of these groups.

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>MANAGER_ID</th>
<th>JOB_ID</th>
<th>SUM(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>101</td>
<td>AD_ASST</td>
<td>4400</td>
</tr>
<tr>
<td>20</td>
<td>100</td>
<td>MK_MAN</td>
<td>13000</td>
</tr>
<tr>
<td>20</td>
<td>201</td>
<td>MK_REP</td>
<td>6000</td>
</tr>
<tr>
<td>50</td>
<td>124</td>
<td>ST_CLERK</td>
<td>11700</td>
</tr>
<tr>
<td>50</td>
<td>100</td>
<td>ST_MAN</td>
<td>5800</td>
</tr>
<tr>
<td>60</td>
<td>102</td>
<td>IT_PROG</td>
<td>9000</td>
</tr>
<tr>
<td>60</td>
<td>103</td>
<td>IT_PROG</td>
<td>10200</td>
</tr>
<tr>
<td>80</td>
<td>100</td>
<td>SA_MAN</td>
<td>10600</td>
</tr>
<tr>
<td>80</td>
<td>149</td>
<td>SA_REP</td>
<td>19600</td>
</tr>
<tr>
<td>90</td>
<td></td>
<td>AD_PRES</td>
<td>24000</td>
</tr>
<tr>
<td>90</td>
<td>100</td>
<td>AD_VP</td>
<td>34000</td>
</tr>
<tr>
<td>110</td>
<td>205</td>
<td>AC_ACCOUNT</td>
<td>8300</td>
</tr>
<tr>
<td>110</td>
<td>101</td>
<td>AC_MGR</td>
<td>12000</td>
</tr>
<tr>
<td>110</td>
<td>149</td>
<td>SA_REP</td>
<td>7000</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>AD_ASST</td>
<td>4400</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>MK_MAN</td>
<td>13000</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>MK_REP</td>
<td>6000</td>
</tr>
<tr>
<td>50</td>
<td></td>
<td>ST_CLERK</td>
<td>11700</td>
</tr>
<tr>
<td>50</td>
<td></td>
<td>ST_MAN</td>
<td>5800</td>
</tr>
<tr>
<td>60</td>
<td></td>
<td>IT_PROG</td>
<td>19200</td>
</tr>
<tr>
<td>80</td>
<td></td>
<td>SA_MAN</td>
<td>10600</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>AD_ASST</td>
<td>4400</td>
</tr>
<tr>
<td>124</td>
<td></td>
<td>ST_CLERK</td>
<td>11700</td>
</tr>
<tr>
<td>149</td>
<td></td>
<td>SA_REP</td>
<td>26600</td>
</tr>
<tr>
<td>201</td>
<td></td>
<td>MK_REP</td>
<td>6000</td>
</tr>
<tr>
<td>205</td>
<td></td>
<td>AC_ACCOUNT</td>
<td>6300</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AD_PRES</td>
<td>24000</td>
</tr>
</tbody>
</table>

40 rows selected.
Advanced Subqueries
Objectives

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

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

Lesson Aim

In this lesson, you learn how to write multiple-column subqueries and subqueries in the FROM clause of a SELECT statement. You also learn how to solve problems by using scalar, correlated subqueries and the WITH clause.
What Is a Subquery?

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

Subqueries can be used for the following purposes:

- To provide values for conditions in WHERE, HAVING, and START WITH clauses of SELECT statements
- To define the set of rows to be inserted into the target table of an INSERT or CREATE TABLE statement
- To define the set of rows to be included in a view or snapshot in a CREATE VIEW or CREATE SNAPSHOT statement
- To define one or more values to be assigned to existing rows in an UPDATE statement
- To define a table to be operated on by a containing query. (You do this by placing the subquery in the FROM clause. This can be done in INSERT, UPDATE, and DELETE statements as well.)

Note: A subquery is evaluated once for the entire parent statement.
Subqueries

You can build powerful statements out of simple ones by using subqueries. Subqueries can be very useful when you need to select rows from a table with a condition that depends on the data in the table itself or some other table. Subqueries are very useful for writing SQL statements that need values based on one or more unknown conditional values.

In the syntax:

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

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

**Note:** Comparison operators fall into two classes: single-row operators (>, =, >=, <, <>, <=) and multiple-row operators (IN, ANY, ALL).

The subquery is often referred to as a nested SELECT, sub-SELECT, or inner SELECT statement. The inner and outer queries can retrieve data from either the same table or different tables.
Using a Subquery

In the example in the slide, the inner query returns the salary of the employee with employee number 149. The outer query uses the result of the inner query to display the names of all the employees who earn more than this amount.

Example

Display the names of all employees who earn less than the average salary in the company.

```
SELECT last_name, job_id, salary
FROM employees
WHERE salary < (SELECT AVG(salary)
                FROM employees);
```
Multiple-Column Subqueries

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

Multiple-Column Subqueries

So far you have written single-row subqueries and multiple-row subqueries where only one column is returned by the inner SELECT statement and this is used to evaluate the expression in the parent select statement. If you want to compare two or more columns, you must write a compound WHERE clause using logical operators. Using multiple-column subqueries, you can combine duplicate WHERE conditions into a single WHERE clause.

Syntax

```
SELECT  column, column, ...
FROM    table
WHERE   (column, column, ...) IN
        (SELECT  column, column, ...
         FROM    table
         WHERE   condition);
```

The graphic in the slide illustrates that the values of the MANAGER_ID and DEPARTMENT_ID from the main query are being compared with the MANAGER_ID and DEPARTMENT_ID values retrieved by the subquery. Since the number of columns that are being compared are more than one, the example qualifies as a multiple-column subquery.
Column Comparisons

Column comparisons in a multiple-column subquery can be:

- Pairwise comparisons
- Nonpairwise comparisons

Pairwise Versus Nonpairwise Comparisons

Column comparisons in a multiple-column subquery can be pairwise comparisons or nonpairwise comparisons.

In the example on the next slide, a pairwise comparison was executed in the WHERE clause. Each candidate row in the SELECT statement must have both the same MANAGER_ID column and the DEPARTMENT_ID as the employee with the EMPLOYEE_ID 178 or 174.

A multiple-column subquery can also be a nonpairwise comparison. In a nonpairwise comparison, each of the columns from the WHERE clause of the parent SELECT statement are individually compared to multiple values retrieved by the inner select statement. The individual columns can match any of the values retrieved by the inner select statement. But collectively, all the multiple conditions of the main SELECT statement must be satisfied for the row to be displayed. The example on the next page illustrates a nonpairwise comparison.
Pairwise Comparison Subquery

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

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

Pairwise Comparison Subquery

The example in the slide is that of a multiple-column subquery because the subquery returns more than one column. It compares the values in the MANAGER_ID column and the DEPARTMENT_ID column of each row in the EMPLOYEES table with the values in the MANAGER_ID column and the DEPARTMENT_ID column for the employees with the EMPLOYEE_ID 178 or 174.

First, the subquery to retrieve the MANAGER_ID and DEPARTMENT_ID values for the employees with the EMPLOYEE_ID 178 or 174 is executed. These values are compared with the MANAGER_ID column and the DEPARTMENT_ID column of each row in the EMPLOYEES table. If the values match, the row is displayed. In the output, the records of the employees with the EMPLOYEE_ID 178 or 174 will not be displayed. The output of the query in the slide follows.

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>MANAGER_ID</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>176</td>
<td>149</td>
<td>80</td>
</tr>
</tbody>
</table>
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);
```

Nonpairwise Comparison Subquery

The example shows a nonpairwise comparison of the columns. It displays the EMPLOYEE_ID, MANAGER_ID, and DEPARTMENT_ID of any employee whose manager ID matches any of the manager IDs of employees whose employee IDs are either 174 or 141 and DEPARTMENT_ID match any of the department IDs of employees whose employee IDs are either 174 or 141.

First, the subquery to retrieve the MANAGER_ID values for the employees with the EMPLOYEE_ID 174 or 141 is executed. Similarly, the second subquery to retrieve the DEPARTMENT_ID values for the employees with the EMPLOYEE_ID 174 or 141 is executed. The retrieved values of the MANAGER_ID and DEPARTMENT_ID columns are compared with the MANAGER_ID and DEPARTMENT_ID column for each row in the EMPLOYEES table. If the MANAGER_ID column of the row in the EMPLOYEES table matches with any of the values of the MANAGER_ID retrieved by the inner subquery and if the DEPARTMENT_ID column of the row in the EMPLOYEES table matches with any of the values of the DEPARTMENT_ID retrieved by the second subquery, the record is displayed. The output of the query in the slide follows.

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>MANAGER_ID</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>142</td>
<td>124</td>
<td>50</td>
</tr>
<tr>
<td>143</td>
<td>124</td>
<td>50</td>
</tr>
<tr>
<td>144</td>
<td>124</td>
<td>50</td>
</tr>
<tr>
<td>176</td>
<td>149</td>
<td>80</td>
</tr>
</tbody>
</table>
Using a Subquery in the FROM Clause

You can use a subquery in the FROM clause of a SELECT statement, which is very similar to how views are used. A subquery in the FROM clause of a SELECT statement is also called an inline view. A subquery in the FROM clause of a SELECT statement defines a data source for that particular SELECT statement, and only that SELECT statement. The example on the slide displays employee last names, salaries, department numbers, and average salaries for all the employees who earn more than the average salary in their department. The subquery in the FROM clause is named b, and the outer query references the SALAVG column using this alias.

```
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, 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 in SQL

A subquery that returns exactly one column value from one row is also referred to as a scalar subquery. Multiple-column subqueries written to compare two or more columns, using a compound WHERE clause and logical operators, do not qualify as scalar subqueries.

The value of the scalar subquery expression is the value of the select list item of the subquery. If the subquery returns 0 rows, the value of the scalar subquery expression is NULL. If the subquery returns more than one row, the Oracle Server returns an error. The Oracle Server has always supported the usage of a scalar subquery in a SELECT statement. The usage of scalar subqueries has been enhanced in Oracle9i.

You can now use scalar subqueries in:

• Condition and expression part of DECODE and CASE
• All clauses of SELECT except GROUP BY
• In the left-hand side of the operator in the SET clause and WHERE clause of UPDATE statement

However, scalar subqueries are not valid expressions in the following places:

• As default values for columns and hash expressions for clusters
• In the RETURNING clause of DML statements
• As the basis of a function-based index
• In GROUP BY clauses, CHECK constraints, WHEN conditions
• HAVING clauses
• In START WITH and CONNECT BY clauses
• In statements that are unrelated to queries, such as CREATE PROFILE

Introduction to Oracle9i: SQL 18-11
Scalar Subqueries: Examples

Scalar Subqueries in \textsc{case} Expressions

\begin{verbatim}
SELECT employee_id, last_name,
       (CASE WHEN department_id = 20
               (SELECT department_id FROM departments
                   WHERE location_id = 1800)
       THEN 'Canada' ELSE 'USA' END) location
FROM employees;
\end{verbatim}

Scalar Subqueries in \textsc{order by} Clause

\begin{verbatim}
SELECT employee_id, last_name
FROM employees e
ORDER BY (SELECT department_name
           FROM departments d
           WHERE e.department_id = d.department_id);
\end{verbatim}

Scalar Subqueries: Examples

The first example in the slide demonstrates that scalar subqueries can be used in \textsc{case} expressions. The inner query returns the value 20, which is the department ID of the department whose location ID is 1800. The \textsc{case} expression in the outer query uses the result of the inner query to display the employee ID, last names, and a value of Canada or USA, depending on whether the department ID of the record retrieved by the outer query is 20 or not.

The result of the preceding example follows:

\begin{tabular}{|c|c|c|}
\hline
\textsc{employee id} & \textsc{last name} & \textsc{location} \\
\hline
100 & King & USA \\
101 & Kochhar & USA \\
102 & De Haan & USA \\
103 & Hunold & USA \\
143 & Makris & USA \\
\hline
201 & Hartstein & Canada \\
202 & Fay & Canada \\
205 & Higgins & USA \\
206 & Gietz & USA \\
\hline
\end{tabular}

20 rows selected.
Scalar Subqueries: Examples (Continued)

The second example in the slide demonstrates that scalar subqueries can be used in the ORDER BY clause. The example orders the output based on the DEPARTMENT_NAME by matching the DEPARTMENT_ID from the EMPLOYEES table with the DEPARTMENT_ID from the DEPARTMENTS table. This comparison is done in a scalar subquery in the ORDER BY clause. The result of the second example follows:

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>205</td>
<td>Higgins</td>
</tr>
<tr>
<td>206</td>
<td>Gietz</td>
</tr>
<tr>
<td>200</td>
<td>Whalen</td>
</tr>
<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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>142</td>
<td>Davies</td>
</tr>
<tr>
<td>143</td>
<td>Matos</td>
</tr>
<tr>
<td>144</td>
<td>Vargas</td>
</tr>
<tr>
<td>176</td>
<td>Grant</td>
</tr>
</tbody>
</table>

20 rows selected.

The second example uses a correlated subquery. In a correlated subquery, the subquery references a column from a table referred to in the parent statement. Correlated subqueries are explained later in this lesson.
Correlated Subqueries

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

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

The Oracle Server performs a correlated subquery when the subquery references a column from a table referred to in the parent statement. A correlated subquery is evaluated once for each row processed by the parent statement. The parent statement can be a SELECT, UPDATE, or DELETE statement.

Nested Subqueries Versus Correlated Subqueries

With a normal nested subquery, the inner SELECT query runs first and executes once, returning values to be used by the main query. A correlated subquery, however, executes once for each candidate row considered by the outer query. In other words, the inner query is driven by the outer query.

Nested Subquery Execution

- The inner query executes first and finds a value.
- The outer query executes once, using the value from the inner query.

Correlated Subquery Execution

- Get a candidate row (fetched by the outer query).
- Execute the inner query using the value of the candidate row.
- Use the values resulting from the inner query to qualify or disqualify the candidate.
- Repeat until no candidate row remains.
Correlated Subqueries

A correlated subquery is one way of reading every row in a table and comparing values in each row against related data. It is used whenever a subquery must return a different result or set of results for each candidate row considered by the main query. In other words, you use a correlated subquery to answer a multipart question whose answer depends on the value in each row processed by the parent statement.

The Oracle Server performs a correlated subquery when the subquery references a column from a table in the parent query.

**Note:** You can use the **ANY** and **ALL** operators in a correlated subquery.
Using Correlated Subqueries

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

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

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>SALARY</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td>24000</td>
<td>90</td>
</tr>
<tr>
<td>Hundal</td>
<td>9000</td>
<td>60</td>
</tr>
<tr>
<td>Maurogas</td>
<td>58000</td>
<td>50</td>
</tr>
<tr>
<td>Zinke</td>
<td>10500</td>
<td>80</td>
</tr>
<tr>
<td>Abel</td>
<td>11000</td>
<td>80</td>
</tr>
<tr>
<td>Hartman</td>
<td>13000</td>
<td>20</td>
</tr>
<tr>
<td>Higgins</td>
<td>12000</td>
<td>110</td>
</tr>
</tbody>
</table>

7 rows selected.

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

Using Correlated Subqueries (continued)

The example in the slide determines which employees earn more than the average salary of their department. In this case, the correlated subquery specifically computes the average salary for each department.

Because both the outer query and inner query use the EMPLOYEES table in the FROM clause, an alias is given to EMPLOYEES in the outer SELECT statement, for clarity. Not only does the alias make the entire SELECT statement more readable, but without the alias the query would not work properly, because the inner statement would not be able to distinguish the inner table column from the outer table column.
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 Correlated Subqueries

The example in the slide displays the details of those employees who have switched jobs at least twice.

The Oracle Server evaluates a correlated subquery as follows:

1. Select a row from the table specified in the outer query. This will be the current candidate row.
2. Store the value of the column referenced in the subquery from this candidate row. (In the example in the slide, the column referenced in the subquery is E.EMPLOYEE_ID.)
3. Perform the subquery with its condition referencing the value from the outer query’s candidate row. (In the example in the slide, group function COUNT(*) is evaluated based on the value of the E.EMPLOYEE_ID column obtained in step 2.)
4. Evaluate the WHERE clause of the outer query on the basis of results of the subquery performed in step 3. This is determines if the candidate row is selected for output. (In the example, the number of times an employee has switched jobs, evaluated by the subquery, is compared with 2 in the WHERE clause of the outer query. If the condition is satisfied, that employee record is displayed.)
5. Repeat the procedure for the next candidate row of the table, and so on until all the rows in the table have been processed.

The correlation is established by using an element from the outer query in the subquery. In this example, the correlation is established by the statement EMPLOYEE_ID = E.EMPLOYEE_ID in which you compare EMPLOYEE_ID from the table in the subquery with the EMPLOYEE_ID from the table in the outer query.
Using the EXISTS Operator

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

The EXISTS Operator

With nesting SELECT statements, all logical operators are valid. In addition, you can use the EXISTS operator. This operator is frequently used with correlated subqueries to test whether a value retrieved by the outer query exists in the results set of the values retrieved by the inner query. If the subquery returns at least one row, the operator returns TRUE. If the value does not exist, it returns FALSE. Accordingly, NOT EXISTS tests whether a value retrieved by the outer query is not a part of the results set of the values retrieved by the inner query.
Using the **EXISTS** Operator

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

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

<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>Koehler</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>

0 rows selected.

**Using the **EXISTS** Operator**

The **EXISTS** operator ensures that the search in the inner query does not continue when at least one match is found for the manager and employee number by the condition:

```
WHERE manager_id = outer.employee_id.
```

Note that the inner **SELECT** query does not need to return a specific value, so a constant can be selected. From a performance standpoint, it is faster to select a constant than a column.

**Note:** Having **EMPLOYEE_ID** in the **SELECT** clause of the inner query causes a table scan for that column. Replacing it with the literal **X**, or any constant, improves performance. This is more efficient than using the **IN** operator.

A **IN** construct can be used as an alternative for a **EXISTS** operator, as shown in the following example:

```
SELECT employee_id, last_name, job_id, department_id
FROM   employees
WHERE  employee_id IN (SELECT manager_id
                        FROM   employees
                        WHERE  manager_id IS NOT NULL);
```
Using the **NOT EXISTS** Operator

Find all departments that do not have any employees.

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

Using the **NOT EXISTS** Operator

**Alternative Solution**

A **NOT IN** construct can be used as an alternative for a **NOT EXISTS** operator, as shown in the following example.

```sql
SELECT department_id, department_name
FROM departments
WHERE department_id NOT IN (SELECT department_id
                               FROM employees);
```

However, **NOT IN** evaluates to **FALSE** if any member of the set is a **NULL** value. Therefore, your query will not return any rows even if there are rows in the **departments** table that satisfy the **WHERE** condition.
Correlated UPDATE

In the case of the UPDATE statement, you can use a correlated subquery to update rows in one table based on rows from another table.

```sql
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.

```
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 UPDATE (continued)

The example in the slide denormalizes the EMPLOYEES table by adding a column to store the department name and then populates the table by using a correlated update.

Here is another example for a correlated update.

Problem Statement

Use a correlated subquery to update rows in the EMPLOYEES table based on rows from the REWARDS table:

```
UPDATE employees
SET salary = (SELECT employees.salary + rewards.pay_raise
              FROM rewards
              WHERE employee_id = employees.employee_id
              AND payraise_date =
                  (SELECT MAX(payraise_date)
                   FROM rewards
                   WHERE employee_id = employees.employee_id))
WHERE employees.employee_id
IN (SELECT employee_id FROM rewards);
```
**Correlated UPDATE (continued)**

This example uses the REWARDS table. The REWARDS table has the columns EMPLOYEE_ID, PAY_RAISE, and PAYRAISE_DATE. Every time an employee gets a pay raise, a record with the details of the employee ID, the amount of the pay raise, and the date of receipt of the pay raise is inserted into the REWARDS table. The REWARDS table can contain more than one record for an employee. The PAYRAISE_DATE column is used to identify the most recent pay raise received by an employee.

In the example, the SALARY column in the EMPLOYEES table is updated to reflect the latest pay raise received by the employee. This is done by adding the current salary of the employee with the corresponding pay raise from the REWARDS table.
Correlated **DELETE**

In the case of a `DELETE` statement, you can use a correlated subquery to delete only those rows that also exist in another table. If you decide that you will maintain only the last four job history records in the `JOB_HISTORY` table, then when an employee transfers to a fifth job, you delete the oldest `JOB_HISTORY` row by looking up the `JOB_HISTORY` table for the `MIN(START_DATE)` for the employee. The following code illustrates how the preceding operation can be performed using a correlated `DELETE`:

```sql
DELETE FROM job_history JH
WHERE employee_id =
  (SELECT employee_id
   FROM employees E
   WHERE JH.employee_id = E.employee_id
   AND START_DATE =
     (SELECT MIN(start_date)
      FROM job_history JH
      WHERE JH.employee_id = E.employee_id)
   AND 5 > (SELECT COUNT(*)
     FROM job_history JH
     WHERE JH.employee_id = E.employee_id
     GROUP BY EMPLOYEE_ID
     HAVING COUNT(*) >= 4));
```
Correlated DELETE

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

```sql
DELETE FROM employees E
WHERE employee_id =
    (SELECT employee_id
     FROM   emp_history
     WHERE  employee_id = E.employee_id);
```

Correlated DELETE (continued)

Example

Two tables are used in this example. They are:

- The EMPLOYEES table, which gives details of all the current employees
- The EMP_HISTORY table, which gives details of previous employees

EMP_HISTORY contains data regarding previous employees, so it would be erroneous if the same employee’s record existed in both the EMPLOYEES and EMP_HISTORY tables. You can delete such erroneous records by using the correlated subquery shown in the slide.
**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.

---

**The WITH clause**

Using the **WITH** clause, you can define a query block before using it in a query. The **WITH** clause (formally known as `subquery_factoring_clause`) enables you to reuse the same query block in a **SELECT** statement when it occurs more than once within a complex query. This is particularly useful when a query has many references to the same query block and there are joins and aggregations.

Using the **WITH** clause, you can reuse the same query when it is high cost to evaluate the query block and it occurs more than once within a complex query. Using the **WITH** clause, the Oracle Server retrieves the results of a query block and stores it in the user’s temporary tablespace. This can improve performance.

**WITH Clause Benefits**

- Makes the query easy to read
- Evaluates a clause only once, even if it appears multiple times in the query, thereby enhancing performance
WITH Clause: Example

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

WITH Clause: Example

The problem in the slide would require the following intermediate calculations:

1. Calculate the total salary for every department, and store the result using a WITH clause.
2. Calculate the average salary across departments, and store the result using a WITH clause.
3. Compare the total salary calculated in the first step with the average salary calculated in the second step. If the total salary for a particular department is greater than the average salary across departments, display the department name and the total salary for that department.

The solution for the preceding problem is given in the next page.
WITH Clause: Example

WITH dept_costs AS (
    SELECT department_name, SUM(salary) AS dept_total
    FROM employees, departments
    WHERE employees.department_id =
        departments.department_id
    GROUP BY department_name),
    avg_cost AS
    (SELECT SUM(dept_total)/COUNT(*) AS dept_avg
     FROM dept_costs)
SELECT * FROM dept_costs
WHERE dept_total >
    (SELECT FROM dept_avg)
ORDER BY department_name;

WITH Clause: Example

The SQL code in the slide is an example of a situation in which you can improve performance and write SQL more simply by using the WITH clause. The query creates the query names DEPT_COSTS and AVG_COST and then uses them in the body of the main query. Internally, the WITH clause is resolved either as an in-line view or a temporary table. The optimizer chooses the appropriate resolution depending on the cost or benefit of temporarily storing the results of the WITH clause.

Note: A subquery in the FROM clause of a SELECT statement is also called an in-line view.

The output generated by the SQL code on the slide will be as follows:

<table>
<thead>
<tr>
<th>DEPARTMENT_NAME</th>
<th>DEPT_TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executive</td>
<td>58000</td>
</tr>
<tr>
<td>Sales</td>
<td>37100</td>
</tr>
</tbody>
</table>

The WITH Clause Usage Notes

- It is used only with SELECT statements.
- A query name is visible to all WITH element query blocks (including their subquery blocks) defined after it and the main query block itself (including its subquery blocks).
- When the query name is the same as an existing table name, the parser searches from the inside out, the query block name takes precedence over the table name.
- The WITH clause can hold more than one query. Each query is then separated by a comma.
Summary

In this lesson, you should have learned the following:

• A multiple-column subquery returns more than one column.
• Multiple-column comparisons can be pairwise or nonpairwise.
• A multiple-column subquery can also be used in the FROM clause of a SELECT statement.
• Scalar subqueries have been enhanced in Oracle 9i.

Summary

You can use multiple-column subqueries to combine multiple WHERE conditions into a single WHERE clause. Column comparisons in a multiple-column subquery can be pairwise comparisons or non-pairwise comparisons.

You can use a subquery to define a table to be operated on by a containing query.

Oracle 9i enhances the the uses of scalar subqueries. Scalar subqueries can now be used in:

• Condition and expression part of DECODE and CASE
• All clauses of SELECT except GROUP BY
• SET clause and WHERE clause of UPDATE statement
Summary

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

Summary

The Oracle Server performs a correlated subquery when the subquery references a column from a table referred to in the parent statement. A correlated subquery is evaluated once for each row processed by the parent statement. The parent statement can be a SELECT, UPDATE, or DELETE statement. Using the WITH clause, you can reuse the same query when it is costly to reevaluate the query block and it occurs more than once within a complex query.
Practice 18 Overview

This practice covers the following topics:
• Creating multiple-column subqueries
• Writing correlated subqueries
• Using the EXISTS operator
• Using scalar subqueries
• Using the WITH clause

Practice 18 Overview

In this practice, you write multiple-column subqueries, correlated and scalar subqueries. You also solve problems by writing the WITH clause.
Practice 18

1. Write a query to display the last name, department number, and salary of any employee whose department number and salary both match the department number and salary of any employee who earns a commission.

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>DEPARTMENT_ID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Taylor</td>
<td>80</td>
<td>8600</td>
</tr>
<tr>
<td>Zlotkey</td>
<td>80</td>
<td>10500</td>
</tr>
<tr>
<td>Abel</td>
<td>80</td>
<td>11000</td>
</tr>
</tbody>
</table>

2. Display the last name, department name, and salary of any employee whose salary and commission match the salary and commission of any employee located in location ID 1700.

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>DEPARTMENT_NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Whalen</td>
<td>Administration</td>
<td>4400</td>
</tr>
<tr>
<td>Gietz</td>
<td>Accounting</td>
<td>8300</td>
</tr>
<tr>
<td>Higgins</td>
<td>Accounting</td>
<td>12000</td>
</tr>
<tr>
<td>Kochhar</td>
<td>Executive</td>
<td>17000</td>
</tr>
<tr>
<td>De Haan</td>
<td>Executive</td>
<td>17000</td>
</tr>
<tr>
<td>King</td>
<td>Executive</td>
<td>24000</td>
</tr>
</tbody>
</table>

6 rows selected.

3. Create a query to display the last name, hire date, and salary for all employees who have the same salary and commission as Kochhar.

    Note: Do not display Kochhar in the result set.

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>HIRE_DATE</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>De Haan</td>
<td>13-JAN-93</td>
<td>17000</td>
</tr>
</tbody>
</table>

4. Create a query to display the employees who earn a salary that is higher than the salary of all of the sales managers (JOB_ID = 'SA_MAN'). Sort the results on salary from highest to lowest.

<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>Kochhar</td>
<td>AD_VP</td>
<td>17000</td>
</tr>
<tr>
<td>De Haan</td>
<td>AD_VP</td>
<td>17000</td>
</tr>
<tr>
<td>Hartstein</td>
<td>MK_MAN</td>
<td>13000</td>
</tr>
<tr>
<td>Higgins</td>
<td>AC_MGR</td>
<td>12000</td>
</tr>
<tr>
<td>Abel</td>
<td>SA_REP</td>
<td>11000</td>
</tr>
</tbody>
</table>

6 rows selected.
5. Display the details of the employee ID, last name, and department ID of those employees who live in cities whose name begins with T.

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>201</td>
<td>Hartstein</td>
<td>20</td>
</tr>
<tr>
<td>202</td>
<td>Fay</td>
<td>20</td>
</tr>
</tbody>
</table>

6. Write a query to find all employees who earn more than the average salary in their departments. Display last name, salary, department ID, and the average salary for the department. Sort by average salary. Use aliases for the columns retrieved by the query as shown in the sample output.

<table>
<thead>
<tr>
<th>ENAME</th>
<th>SALARY</th>
<th>DEPTNO</th>
<th>DEPT_AVG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mourgos</td>
<td>5800</td>
<td>50</td>
<td>3500</td>
</tr>
<tr>
<td>Hurdle</td>
<td>9000</td>
<td>60</td>
<td>6400</td>
</tr>
<tr>
<td>Hartstein</td>
<td>13000</td>
<td>20</td>
<td>9500</td>
</tr>
<tr>
<td>Abel</td>
<td>11000</td>
<td>80</td>
<td>10033.3333</td>
</tr>
<tr>
<td>Zlotkey</td>
<td>10500</td>
<td>00</td>
<td>10033.3333</td>
</tr>
<tr>
<td>Higgins</td>
<td>12000</td>
<td>110</td>
<td>10150</td>
</tr>
<tr>
<td>King</td>
<td>24000</td>
<td>90</td>
<td>19333.3333</td>
</tr>
</tbody>
</table>

7 rows selected.

7. Find all employees who are not supervisors.
   a. First do this using the NOT EXISTS operator.

<table>
<thead>
<tr>
<th>LAST_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ernst</td>
</tr>
<tr>
<td>Lorentz</td>
</tr>
<tr>
<td>Raja</td>
</tr>
<tr>
<td>Davies</td>
</tr>
<tr>
<td>Matos</td>
</tr>
<tr>
<td>Vargas</td>
</tr>
<tr>
<td>Abel</td>
</tr>
<tr>
<td>Taylor</td>
</tr>
<tr>
<td>Grant</td>
</tr>
<tr>
<td>Whalen</td>
</tr>
<tr>
<td>Fay</td>
</tr>
<tr>
<td>Gietz</td>
</tr>
</tbody>
</table>

12 rows selected.

b. Can this be done by using the NOT IN operator? How, or why not?
8. Write a query to display the last names of the employees who earn less than the average salary in their departments.

<table>
<thead>
<tr>
<th>LAST_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kochhar</td>
</tr>
<tr>
<td>De Haan</td>
</tr>
<tr>
<td>Ernst</td>
</tr>
<tr>
<td>Lorentz</td>
</tr>
<tr>
<td>Davies</td>
</tr>
<tr>
<td>Matos</td>
</tr>
<tr>
<td>Vargas</td>
</tr>
<tr>
<td>Taylor</td>
</tr>
<tr>
<td>Fay</td>
</tr>
<tr>
<td>Gietz</td>
</tr>
</tbody>
</table>

10 rows selected.

9. Write a query to display the last names of the employees who have one or more coworkers in their departments with later hire dates but higher salaries.

<table>
<thead>
<tr>
<th>LAST_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raja</td>
</tr>
<tr>
<td>Davies</td>
</tr>
<tr>
<td>Matos</td>
</tr>
<tr>
<td>Vargas</td>
</tr>
<tr>
<td>Taylor</td>
</tr>
</tbody>
</table>
10. Write a query to display the employee ID, last names, and department names of all employees. 
   **Note:** Use a scalar subquery to retrieve the department name in the SELECT statement.

```
<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>DEPARTMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>205</td>
<td>Higgins</td>
<td>Accounting</td>
</tr>
<tr>
<td>206</td>
<td>Gietz</td>
<td>Accounting</td>
</tr>
<tr>
<td>200</td>
<td>Whalen</td>
<td>Administration</td>
</tr>
<tr>
<td>100</td>
<td>King</td>
<td>Executive</td>
</tr>
<tr>
<td>101</td>
<td>Kochhar</td>
<td>Executive</td>
</tr>
<tr>
<td>102</td>
<td>De Haan</td>
<td>Executive</td>
</tr>
<tr>
<td>103</td>
<td>Hunold</td>
<td>IT</td>
</tr>
<tr>
<td>104</td>
<td>Ernst</td>
<td>IT</td>
</tr>
<tr>
<td>107</td>
<td>Lorentz</td>
<td>IT</td>
</tr>
<tr>
<td>201</td>
<td>Hartstein</td>
<td>Marketing</td>
</tr>
<tr>
<td>202</td>
<td>Fay</td>
<td>Marketing</td>
</tr>
<tr>
<td>149</td>
<td>Zlotkey</td>
<td>Sales</td>
</tr>
<tr>
<td>176</td>
<td>Taylor</td>
<td>Sales</td>
</tr>
<tr>
<td>174</td>
<td>Abel</td>
<td>Sales</td>
</tr>
<tr>
<td>124</td>
<td>Morgos</td>
<td>Shipping</td>
</tr>
<tr>
<td>141</td>
<td>Rejs</td>
<td>Shipping</td>
</tr>
<tr>
<td>142</td>
<td>Davies</td>
<td>Shipping</td>
</tr>
<tr>
<td>143</td>
<td>Matos</td>
<td>Shipping</td>
</tr>
<tr>
<td>144</td>
<td>Vargas</td>
<td>Shipping</td>
</tr>
<tr>
<td>178</td>
<td>Grant</td>
<td></td>
</tr>
</tbody>
</table>
```

20 rows selected.

11. Write a query to display the department names of those departments whose total salary cost is above one eighth (1/8) of the total salary cost of the whole company. Use the WITH clause to write this query. Name the query `SUMMARY`.

```
<table>
<thead>
<tr>
<th>DEPARTMENT_NAME</th>
<th>DEPT_TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executive</td>
<td>58000</td>
</tr>
<tr>
<td>Sales</td>
<td>37100</td>
</tr>
</tbody>
</table>
```
Hierarchical Retrieval
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

Lesson Aim

In this lesson, you learn how to use hierarchical queries to create tree-structured reports.
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>Hunkeld</td>
<td>IT_PROG</td>
<td>102</td>
</tr>
<tr>
<td>104</td>
<td>Ernst</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>Murgos</td>
<td>ST_MAN</td>
<td>100</td>
</tr>
<tr>
<td>141</td>
<td>Raje</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>Motes</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>145</td>
<td>Zrikey</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>178</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>Huttstein</td>
<td>MK_MAN</td>
<td>100</td>
</tr>
<tr>
<td>202</td>
<td>Fry</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>Getz</td>
<td>AC_ACCOUNT</td>
<td>205</td>
</tr>
</tbody>
</table>

Sample Data from the EMPLOYEES Table

Using hierarchical queries, you can retrieve data based on a natural hierarchical relationship between rows in a table. A relational database does not store records in a hierarchical way. However, where a hierarchical relationship exists between the rows of a single table, a process called tree walking enables the hierarchy to be constructed. A hierarchical query is a method of reporting, in order, the branches of a tree.

Imagine a family tree with the eldest members of the family found close to the base or trunk of the tree and the youngest members representing branches of the tree. Branches can have their own branches, and so on.

A hierarchical query is possible when a relationship exists between rows in a table. For example, in the slide, you see that employees with the job IDs of AD_VP, ST_MAN, SA_MAN, and MK_MAN report directly to the president of the company. We know this because the MANAGER_ID column of these records contain the employee ID 100, which belongs to the president (AD_PRES).

Note: Hierarchical trees are used in various fields such as human genealogy (family trees), livestock (breeding purposes), corporate management (management hierarchies), manufacturing (product assembly), evolutionary research (species development), and scientific research.
Natural Tree Structure

The EMPLOYEES table has a tree structure representing the management reporting line. The hierarchy can be created by looking at the relationship between equivalent values in the EMPLOYEE_ID and MANAGER_ID columns. This relationship can be exploited by joining the table to itself. The MANAGER_ID column contains the employee number of the employee’s manager.

The parent-child relationship of a tree structure enables you to control:

- The direction in which the hierarchy is walked
- The starting point inside the hierarchy

Note: The slide displays an inverted tree structure of the management hierarchy of the employees in the EMPLOYEES table.
Hierarchical Queries

Hierarchical queries can be identified by the presence of the `CONNECT BY` and `START WITH` clauses.

In the syntax:

```
SELECT [LEVEL], column, expr...
FROM table
[WHERE condition(s)]
[START WITH condition(s)]
[CONNECT BY PRIOR condition(s)];
```

**WHERE condition:**

```
expr comparison_operator expr
```

**Keywords and Clauses**

Hierarchical queries can be identified by the presence of the `CONNECT BY` and `START WITH` clauses.

In the syntax:

- **SELECT** is the standard `SELECT` clause.
- **LEVEL** For each row returned by a hierarchical query, the `LEVEL` pseudocolumn returns 1 for a root row, 2 for a child of a root, and so on.
- **FROM table** Specifies the table, view, or snapshot containing the columns. You can select from only one table.
- **WHERE** Restricts the rows returned by the query without affecting other rows of the hierarchy.
- **condition** Is a comparison with expressions.
- **START WITH** Specifies the root rows of the hierarchy (where to start). This clause is required for a true hierarchical query.
- **CONNECT BY PRIOR** Specifies the columns in which the relationship between parent and child rows exist. This clause is required for a hierarchical query.

The `SELECT` statement cannot contain a join or query from a view that contains a join.
Walking the Tree

Starting Point

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

\[
\text{START WITH } \text{column1} = \text{value}
\]

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

\[
\ldots \text{START WITH } \text{last}_\text{name} = \text{'Kochhar'}
\]

Walking the Tree

The row or rows to be used as the root of the tree are determined by the START WITH clause. The START WITH clause can be used in conjunction with any valid condition.

Examples

Using the EMPLOYEES table, start with King, the president of the company.

\[
\ldots \text{START WITH } \text{manager}_\text{id} \text{ IS NULL}
\]

Using the EMPLOYEES table, start with employee Kochhar. A START WITH condition can contain a subquery.

\[
\ldots \text{START WITH } \text{employee}_\text{id} = \text{(SELECT employee}_\text{id} \\
\text{FROM employees} \\
\text{WHERE } \text{last}_\text{name} = \text{'Kochhar'})
\]

If the START WITH clause is omitted, the tree walk is started with all of the rows in the table as root rows. If a WHERE clause is used, the walk is started with all the rows that satisfy the WHERE condition. This no longer reflects a true hierarchy.

Note: The clauses CONNECT BY PRIOR and START WITH are not ANSI SQL standard.
Walking the Tree (continued)

The direction of the query, whether it is from parent to child or from child to parent, is determined by the CONNECT BY PRIOR column placement. The PRIOR operator refers to the parent row. To find the children of a parent row, the Oracle Server evaluates the PRIOR expression for the parent row and the other expressions for each row in the table. Rows for which the condition is true are the children of the parent. The Oracle Server always selects children by evaluating the CONNECT BY condition with respect to a current parent row.

Examples

Walk from the top down using the EMPLOYEES table. Define a hierarchical relationship in which the EMPLOYEE_ID value of the parent row is equal to the MANAGER_ID value of the child row.

... CONNECT BY PRIOR employee_id = manager_id

Walk from the bottom up using the EMPLOYEES table.

... CONNECT BY PRIOR manager_id = employee_id

The PRIOR operator does not necessarily need to be coded immediately following the CONNECT BY. Thus, the following CONNECT BY PRIOR clause gives the same result as the one in the preceding example.

... CONNECT BY employee_id = PRIOR manager_id

Note: The CONNECT BY clause cannot contain a subquery.
Walking the Tree: From the Bottom Up

The example in the slide displays a list of managers starting with the employee whose employee ID is 101.

Example
In the following example, EMPLOYEE_ID values are evaluated for the parent row and MANAGER_ID, and SALARY values are evaluated for the child rows. The PRIOR operator applies only to the EMPLOYEE_ID value.

```
... CONNECT BY PRIOR employee_id = manager_id
    AND salary > 15000;
```

To qualify as a child row, a row must have a MANAGER_ID value equal to the EMPLOYEE_ID value of the parent row and must have a SALARY value greater than $15,000.
Walking the Tree: From the Top Down

Walking from the top down, display the names of the employees and their manager. Use employee King as the starting point. Print only one column.
Ranking Rows with the **LEVEL** Pseudocolumn

You can explicitly show the rank or level of a row in the hierarchy by using the **LEVEL** pseudocolumn. This will make your report more readable. The forks where one or more branches split away from a larger branch are called nodes, and the very end of a branch is called a leaf, or leaf node. The diagram in the slide shows the nodes of the inverted tree with their **LEVEL** values. For example, employee Higgens is a parent and a child, while employee Davies is a child and a leaf.

**The **LEVEL** Pseudocolumn**

<table>
<thead>
<tr>
<th>Value</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A root node</td>
</tr>
<tr>
<td>2</td>
<td>A child of a root node</td>
</tr>
<tr>
<td>3</td>
<td>A child of a child, and so on</td>
</tr>
</tbody>
</table>

**Note:** A **root node** is the highest node within an inverted tree. A **child node** is any nonroot node. A parent node is any node that has children. A leaf node is any node without children. The number of levels returned by a hierarchical query may be limited by available user memory.

In the slide, King is the root or parent (**LEVEL = 1**). Kochhar, De Hann, Mourgos, Zlotkey, Hartstein, Higgens, and Hunold are children and also parents (**LEVEL = 2**). Whalen, Rajs, Davies, Matos, Vargas, Gietz, Ernst, Lorentz, Abel, Taylor, Grant, and Goyal are children and leaves. (**LEVEL = 3** and **LEVEL = 4**).
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}

Formatting Hierarchical Reports Using \texttt{LEVEL}

The nodes in a tree are assigned level numbers from the root. Use the \texttt{LPAD} function in conjunction with the pseudocolumn \texttt{LEVEL} to display a hierarchical report as an indented tree.

In the example on the slide:

- \texttt{LPAD(ciår1,n [,char2])} returns \texttt{ciår1}, left-padded to length \texttt{n} with the sequence of characters in \texttt{char2}. The argument \texttt{n} is the total length of the return value as it is displayed on your terminal screen.
- \texttt{LPAD(last_name, LENGTH(last_name)+(LEVEL*2)-2,'_')} defines the display format.
- \texttt{char1} is the \texttt{LAST_NAME}, \texttt{n} the total length of the return value, is length of the \texttt{LAST_NAME} + (LEVEL*2)-2, and \texttt{char2} is '\_'.

In other words, this tells SQL to take the \texttt{LAST_NAME} and left-pad it with the '\_' character till the length of the resultant string is equal to the value determined by \texttt{LENGTH(last_name)+(LEVEL*2)-2}.

For King, \texttt{LEVEL = 1}. Hence, \((2 \times 1) - 2 = 2 - 2 = 0\). So King does not get padded with any '\_' character and is displayed in column 1.

For Kochhar, \texttt{LEVEL = 2}. Hence, \((2 \times 2) - 2 = 4 - 2 = 2\). So Kochhar gets padded with 2 '\_' characters and is displayed indented.

The rest of the records in the \texttt{EMPLOYEES} table are displayed similarly.
### Formatting Hierarchical Reports Using `LEVEL` (continued)

<table>
<thead>
<tr>
<th>ORG_CHART</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
</tr>
<tr>
<td>____Kochhar</td>
</tr>
<tr>
<td>____Whalen</td>
</tr>
<tr>
<td>____Higgins</td>
</tr>
<tr>
<td>____Gietz</td>
</tr>
<tr>
<td>____De Haan</td>
</tr>
<tr>
<td>____Hunold</td>
</tr>
<tr>
<td>_______Ernst</td>
</tr>
<tr>
<td>_______Lorentz</td>
</tr>
<tr>
<td>____Mourgos</td>
</tr>
<tr>
<td>____Rajs</td>
</tr>
<tr>
<td>____Davies</td>
</tr>
<tr>
<td>____Matos</td>
</tr>
<tr>
<td>____Vargas</td>
</tr>
<tr>
<td>____Zlotkey</td>
</tr>
<tr>
<td>____Abel</td>
</tr>
<tr>
<td>____Taylor</td>
</tr>
<tr>
<td>____Grant</td>
</tr>
<tr>
<td>____Hartstein</td>
</tr>
<tr>
<td>____Fay</td>
</tr>
</tbody>
</table>

20 rows selected.
Pruning Branches

You can use the WHERE and CONNECT BY clauses to prune the tree; that is, to control which nodes or rows are displayed. The predicate you use acts as a Boolean condition.

Examples
Starting at the root, walk from the top down, and eliminate employee Higgins in the result, but process the child rows.

```
SELECT department_id, employee_id, last_name, job_id, salary
FROM employees
WHERE last_name != 'Higgins'
START WITH manager_id IS NULL
CONNECT BY PRIOR employee_id = manager_id
```

Starting at the root, walk from the top down, and eliminate employee Higgins and all child rows.

```
SELECT department_id, employee_id, last_name, job_id, salary
FROM employees
START WITH manager_id IS NULL
CONNECT BY PRIOR employee_id = manager_id
AND last_name != 'Higgins';
```
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.

Summary

You can use hierarchical queries to retrieve data based on a natural hierarchical relationship between rows in a table. The LEVEL pseudocolumn counts how far down a hierarchical tree you have traveled. You can specify the direction of the query using the CONNECT BY PRIOR clause. You can specify the starting point using the START WITH clause. You can use the WHERE and CONNECT BY clauses to prune the tree branches.
Practice 19 Overview

This practice covers the following topics:

• Distinguishing hierarchical queries from nonhierarchical queries
• Performing tree walks
• Producing an indented report by using the LEVEL pseudocolumn
• Pruning the tree structure
• Sorting the output

Paper-Based Questions

Question 1 is a paper-based question.
Practice 19

1. Look at the following output. Is this output the result of a hierarchical query? Explain why or why not.

   a. Exhibit 1:

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>MANAGER_ID</th>
<th>SALARY</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>144</td>
<td>Vargas</td>
<td>124</td>
<td>2500</td>
<td>50</td>
</tr>
<tr>
<td>143</td>
<td>Matos</td>
<td>124</td>
<td>2600</td>
<td>50</td>
</tr>
<tr>
<td>142</td>
<td>Davies</td>
<td>124</td>
<td>3100</td>
<td>50</td>
</tr>
<tr>
<td>141</td>
<td>Rajs</td>
<td>124</td>
<td>3500</td>
<td>50</td>
</tr>
<tr>
<td>107</td>
<td>Lorentz</td>
<td>103</td>
<td>4200</td>
<td>60</td>
</tr>
<tr>
<td>200</td>
<td>Whalen</td>
<td>101</td>
<td>4400</td>
<td>10</td>
</tr>
<tr>
<td>124</td>
<td>Mourgos</td>
<td>100</td>
<td>5800</td>
<td>50</td>
</tr>
<tr>
<td>104</td>
<td>Ernst</td>
<td>103</td>
<td>6000</td>
<td>60</td>
</tr>
<tr>
<td>202</td>
<td>Fay</td>
<td>201</td>
<td>6000</td>
<td>20</td>
</tr>
</tbody>
</table>

   201 | Hartstein | 100 | 13000 | 90 |
   101 | Kochhar   | 100 | 17000 | 90 |
   102 | De Haan   | 100 | 17000 | 90 |
   100 | King      | 100 | 24000 | 90 |

   Exhibit 2:

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>DEPARTMENT_ID</th>
<th>DEPARTMENT_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>Whalen</td>
<td>10</td>
<td>Administration</td>
</tr>
<tr>
<td>201</td>
<td>Hartstein</td>
<td>20</td>
<td>Marketing</td>
</tr>
<tr>
<td>202</td>
<td>Fay</td>
<td>20</td>
<td>Marketing</td>
</tr>
<tr>
<td>124</td>
<td>Mourgos</td>
<td>50</td>
<td>Shipping</td>
</tr>
<tr>
<td>141</td>
<td>Rajs</td>
<td>50</td>
<td>Shipping</td>
</tr>
<tr>
<td>100</td>
<td>King</td>
<td>90</td>
<td>Executive</td>
</tr>
<tr>
<td>101</td>
<td>Kochhar</td>
<td>90</td>
<td>Executive</td>
</tr>
<tr>
<td>102</td>
<td>De Haan</td>
<td>90</td>
<td>Executive</td>
</tr>
<tr>
<td>205</td>
<td>Higgins</td>
<td>110</td>
<td>Accounting</td>
</tr>
<tr>
<td>206</td>
<td>Gietz</td>
<td>110</td>
<td>Accounting</td>
</tr>
</tbody>
</table>

Introduction to Oracle9i: SQL 19-16
Practice 19 (continued)

Exhibit 3:

<table>
<thead>
<tr>
<th>RANK</th>
<th>EMPLOYEE_ID</th>
<th>DEPARTMENT_ID</th>
<th>MANAGER_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>90</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>101</td>
<td>90</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>200</td>
<td>10</td>
<td>101</td>
</tr>
<tr>
<td>3</td>
<td>205</td>
<td>110</td>
<td>101</td>
</tr>
<tr>
<td>4</td>
<td>206</td>
<td>110</td>
<td>205</td>
</tr>
<tr>
<td>2</td>
<td>102</td>
<td>90</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>103</td>
<td>60</td>
<td>102</td>
</tr>
<tr>
<td>4</td>
<td>104</td>
<td>60</td>
<td>103</td>
</tr>
</tbody>
</table>

|     |            |              |            | 149 |
|-----|------------|--------------|------------|
| 3   | 174        |              |            |
| 3   | 176        | 80           | 149        |
| 3   | 178        |              | 149        |
| 2   | 201        | 20           | 100        |
| 3   | 202        | 20           | 201        |
Practice 19 (continued)

2. Produce a report showing an organization chart for Mourgos’s department. Print last names, salaries, and department IDs.

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>SALARY</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mourgos</td>
<td>5800</td>
<td>50</td>
</tr>
<tr>
<td>Rajs</td>
<td>3500</td>
<td>50</td>
</tr>
<tr>
<td>Davies</td>
<td>3100</td>
<td>50</td>
</tr>
<tr>
<td>Matos</td>
<td>2600</td>
<td>50</td>
</tr>
<tr>
<td>Vargas</td>
<td>2500</td>
<td>50</td>
</tr>
</tbody>
</table>

3. Create a report that shows the hierarchy of the managers for the employee Lorentz. Display his immediate manager first.

<table>
<thead>
<tr>
<th>LAST_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hunold</td>
</tr>
<tr>
<td>De Haan</td>
</tr>
<tr>
<td>King</td>
</tr>
</tbody>
</table>
Practice 19 (continued)

4. Create an indented report showing the management hierarchy starting from the employee whose LAST_NAME is Kochhar. Print the employee’s last name, manager ID, and department ID. Give alias names to the columns as shown in the sample output.

<table>
<thead>
<tr>
<th>NAME</th>
<th>MGR</th>
<th>DEPTNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kochhar</td>
<td>100</td>
<td>90</td>
</tr>
<tr>
<td>Whalen</td>
<td>101</td>
<td>10</td>
</tr>
<tr>
<td>Higgins</td>
<td>101</td>
<td>110</td>
</tr>
<tr>
<td>Gietz</td>
<td>205</td>
<td>110</td>
</tr>
</tbody>
</table>

If you have time, complete the following exercise:

5. Produce a company organization chart that shows the management hierarchy. Start with the person at the top level, exclude all people with a job ID of IT_PROG, and exclude De Haan and those employees who report to De Haan.

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>EMPLOYEE_ID</th>
<th>MANAGER_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Hartstein</td>
<td>201</td>
<td>100</td>
</tr>
<tr>
<td>Fay</td>
<td>202</td>
<td>201</td>
</tr>
<tr>
<td>Kochhar</td>
<td>101</td>
<td>100</td>
</tr>
<tr>
<td>Whalen</td>
<td>200</td>
<td>101</td>
</tr>
<tr>
<td>Higgins</td>
<td>205</td>
<td>101</td>
</tr>
<tr>
<td>Gietz</td>
<td>206</td>
<td>205</td>
</tr>
<tr>
<td>Mourgos</td>
<td>124</td>
<td>100</td>
</tr>
<tr>
<td>Raja</td>
<td>141</td>
<td>124</td>
</tr>
<tr>
<td>Davies</td>
<td>142</td>
<td>124</td>
</tr>
<tr>
<td>Matos</td>
<td>143</td>
<td>124</td>
</tr>
<tr>
<td>Vargas</td>
<td>144</td>
<td>124</td>
</tr>
<tr>
<td>Zlotkey</td>
<td>149</td>
<td>100</td>
</tr>
<tr>
<td>Abel</td>
<td>174</td>
<td>149</td>
</tr>
<tr>
<td>Taylor</td>
<td>176</td>
<td>149</td>
</tr>
<tr>
<td>Grant</td>
<td>178</td>
<td>149</td>
</tr>
</tbody>
</table>

16 rows selected.
Oracle 9i 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 INSERT
  – Pivoting INSERT
  – Conditional ALL INSERT
  – Conditional FIRST INSERT
• Create and use external tables
• Name the index at the time of creating a primary key constraint

Lesson Aim

This lesson addresses the Oracle9i extensions to DDL and DML statements. It focuses on multitable INSERT statements, types of multitable INSERT statements, external tables, and the provision to name the index at the time of creating a primary key constraint.
Review of the INSERT Statement

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

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

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

```
INSERT INTO departments(department_id, department_name, manager_id, location_id) 
VALUES (70, 'Public Relations', 100, 1700); 
1 row created.
```

Review of the INSERT Statement

You can add new rows to a table by issuing the INSERT statement. 
In the syntax:
- `table` is the name of the table 
- `column` is the name of the column in the table to populate 
- `value` is the corresponding value for the column 

**Note:** This statement with the VALUES clause adds only one row at a time to a table.
Review of the **UPDATE** Statement

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

```sql
UPDATE employees
SET department_id = 70
WHERE employee_id = 142;
1 row updated.
```

Review of the **UPDATE** Statement

You can modify existing rows by using the **UPDATE** statement.

In the syntax:
- `table` is the name of the table
- `column` is the name of the column in the table to populate
- `value` is the corresponding value or subquery for the column
- `condition` identifies the rows to be updated and is composed of column names, expressions, constants, subqueries, and comparison operators

Confirm the update operation by querying the table to display the updated rows.
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

Introduction to Oracle9i: SQL 20-5
Overview of Multitable Insert Statements

Multitable INSERTS statement offer the benefits of the INSERT ... SELECT statement when multiple tables are involved as targets. Using functionality prior to Oracle9i, you had to deal with \( n \) independent INSERT ... SELECT statements, thus processing the same source data \( n \) times and increasing the transformation workload \( n \) times.

As with the existing INSERT ... SELECT statement, the new statement can be parallelized and used with the direct-load mechanism for faster performance.

Each record from any input stream, such as a nonrelational database table, can now be converted into multiple records for more relational database table environment. To implement this functionality before Oracle9i, you had to write multiple INSERT statements.
Types of Multitable INSERT Statements

Oracle9i introduces the following types of multitable insert statements:

- Unconditional INSERT
- Conditional ALL INSERT
- Conditional FIRST INSERT
- Pivoting INSERT

You use different clauses to indicate the type of INSERT to be executed.
Multitable INSERT Statements

Syntax

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

```sql
conditional_insert_clause

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

Multitable INSERT Statements

The slide displays the generic format for multitable INSERT statements. There are four types of multitable insert statements.

- Unconditional INSERT
- Conditional ALL INSERT
- Conditional FIRST INSERT
- Pivoting INSERT

Unconditional INSERT: ALL into_clause

Specify ALL followed by multiple insert_into_clauses to perform an unconditional multitable insert. The Oracle Server executes each insert_into_clause once for each row returned by the subquery.

Conditional INSERT: conditional_insert_clause

Specify the conditional_insert_clause to perform a conditional multitable insert. The Oracle server filters each insert_into_clause through the corresponding WHEN condition, which determines whether that insert_into_clause is executed. A single multitable insert statement can contain up to 127 WHEN clauses.

Conditional INSERT: ALL

If you specify ALL, the Oracle server evaluates each WHEN clause regardless of the results of the evaluation of any other WHEN clause. For each WHEN clause whose condition evaluates to true, the Oracle server executes the corresponding INTO clause list.
Multitable INSERT Statements (continued)

Conditional FIRST: INSERT

If you specify FIRST, the Oracle Server evaluates each WHEN clause in the order in which it appears in the statement. If the first WHEN clause evaluates to true, the Oracle Server executes the corresponding INTO clause and skips subsequent WHEN clauses for the given row.

Conditional INSERT: ELSE Clause

For a given row, if no WHEN clause evaluates to true:

- If you have specified an ELSE clause, the Oracle Server executes the INTO clause list associated with the ELSE clause.
- If you did not specify an ELSE clause, the Oracle Server takes no action for that row.

Restrictions on Multitable INSERT Statements

- You can perform multitable inserts only on tables, not on views or materialized views.
- You cannot perform a multitable insert into a remote table.
- You cannot specify a table collection expression when performing a multitable insert.
- In a multitable insert, all of the insert_into_clauses cannot combine to specify more than 999 target columns.
Unconditional INSERT ALL

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

```
INSERT ALL
INTO sal_history VALUES(EMPID, HIREDATE, SAL)
INTO mgr_history VALUES(EMPID, MGR, SAL)
```

```
SELECT employee_id EMPI, hire_date HIREDATE, 
salary SAL, manager_id MGR
FROM employees
WHERE employee_id > 200;
```

8 rows created.

Unconditional INSERT ALL

The example in the slide inserts rows into both the SAL_HISTORY and the MGR_HISTORY tables. The SELECT statement retrieves the details of employee ID, hire date, salary, and manager ID of those employees whose employee ID is greater than 200 from the EMPLOYEES table. The details of the employee ID, hire date, and salary are inserted into the SAL_HISTORY table. The details of employee ID, manager ID and salary are inserted into the MGR_HISTORY table.

This INSERT statement is referred to as an unconditional INSERT, as no further restriction is applied to the rows that are retrieved by the SELECT statement. All the rows retrieved by the SELECT statement are inserted into the two tables, SAL_HISTORY and MGR_HISTORY. The VALUES clause in the INSERT statements specifies the columns from the SELECT statement that have to be inserted into each of the tables. Each row returned by the SELECT statement results in two inserts, one for the SAL_HISTORY table and one for the MGR_HISTORY table.

The feedback 8 rows created can be interpreted to mean that a total of eight inserts were performed on the base tables, SAL_HISTORY and MGR_HISTORY.
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_HISTROY 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.

The problem statement for a conditional INSERT ALL statement is specified in the slide. The solution to the preceding problem is shown in the next page.
Conditional INSERT ALL (continued)

The example on the slide is similar to the example on the previous slide as it inserts rows into both the SAL_HISTORY and the MGR_HISTORY tables. The SELECT statement retrieves the details of employee ID, hire date, salary, and manager ID of those employees whose employee ID is greater than 200 from the EMPLOYEES table. The details of employee ID, hire date, and salary are inserted into the SAL_HISTORY table. The details of employee ID, manager ID, and salary are inserted into the MGR_HISTORY table.

This INSERT statement is referred to as a conditional ALL INSERT, as a further restriction is applied to the rows that are retrieved by the SELECT statement. From the rows that are retrieved by the SELECT statement, only those rows in which the value of the SAL column is more than 10000 are inserted in the SAL_HISTORY table, and similarly only those rows where the value of the MGR column is more than 200 are inserted in the MGR_HISTORY table.

Observe that unlike the previous example, where eight rows were inserted into the tables, in this example only four rows are inserted.

The feedback 4 rows created can be interpreted to mean that a total of four inserts were performed on the base tables, SAL_HISTORY and MGR_HISTORY.
Conditional **FIRST INSERT**

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

---

**Conditional **FIRST INSERT****

The problem statement for a conditional **FIRST INSERT** statement is specified in the slide. The solution to the preceding problem is shown on the next page.
Conditional FIRST INSERT (continued)

The example in the slide inserts rows into more than one table, using one single INSERT statement. The SELECT statement retrieves the details of department ID, total salary, and maximum hire date for every department in the EMPLOYEES table.

This INSERT statement is referred to as a conditional FIRST INSERT, as an exception is made for the departments whose total salary is more than $25,000. The condition WHEN ALL > 25000 is evaluated first. If the total salary for a department is more than $25,000, then the record is inserted into the SPECIAL_SAL table irrespective of the hire date. If this first WHEN clause evaluates to true, the Oracle server executes the corresponding INTO clause and skips subsequent WHEN clauses for this row.

For the rows that do not satisfy the first WHEN condition (WHEN SAL > 25000), the rest of the conditions are evaluated just as a conditional INSERT statement, and the records retrieved by the SELECT statement are inserted into the HIREDATE_HISTORY_00, or HIREDATE_HISTORY_99, or HIREDATE_HISTORY tables, based on the value in the HIREDATE column.

The feedback 8 rows created can be interpreted to mean that a total of eight INSERT statements were performed on the base tables, SPECIAL_SAL, HIREDATE_HISTORY_00, HIREDATE_HISTORY_99, and HIREDATE_HISTORY.
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.

Pivoting INSERT

Pivoting is an operation in which you need to build a transformation such that each record from any input stream, such as, a nonrelational database table, must be converted into multiple records for a more relational database table environment.

In order to solve the problem mentioned in the slide, you need to build a transformation such that each record from the original nonrelational database table, `SALES_SOURCE_DATA`, is converted into five records for the data warehouse's `SALES_INFO` table. This operation is commonly referred to as pivoting.

The problem statement for a pivoting INSERT statement is specified in the slide. The solution to the preceding problem is shown in the next page.
Pivoting INSERT

INSERT ALL
INTO sales_info VALUES (employee_id, week_id, sales_MON)
INTO sales_info VALUES (employee_id, week_id, sales_TUE)
INTO sales_info VALUES (employee_id, week_id, sales_WED)
INTO sales_info VALUES (employee_id, week_id, sales_THUR)
INTO sales_info VALUES (employee_id, week_id, sales_FRI)
SELECT EMPLOYEE_ID, week_id, sales_MON, sales_TUE,
      sales_WED, sales_THUR, sales_FRI
FROM sales_source_data;

5 rows created.

Pivoting INSERT

In the example in the slide, the sales data is received from the nonrelational database table
SALES_SOURCE_DATA, which is the details of the sales performed by a sales representative on each day
of a week, for a week with a particular week ID.

```
DESC SALES_SOURCE_DATA
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Null?</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPLOYEE_ID</td>
<td></td>
<td>NUMBER(6)</td>
</tr>
<tr>
<td>WEEK_ID</td>
<td></td>
<td>NUMBER(2)</td>
</tr>
<tr>
<td>SALES_MON</td>
<td></td>
<td>NUMBER(8,2)</td>
</tr>
<tr>
<td>SALES_TUE</td>
<td></td>
<td>NUMBER(8,2)</td>
</tr>
<tr>
<td>SALES_WED</td>
<td></td>
<td>NUMBER(8,2)</td>
</tr>
<tr>
<td>SALES_THUR</td>
<td></td>
<td>NUMBER(8,2)</td>
</tr>
<tr>
<td>SALES_FRI</td>
<td></td>
<td>NUMBER(8,2)</td>
</tr>
</tbody>
</table>
Pivoting INSERT (continued)

```
SELECT * FROM SALES_SOURCE_DATA;
```

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>WEEK_ID</th>
<th>SALES_MON</th>
<th>SALES_TUE</th>
<th>SALES_WED</th>
<th>SALES_THUR</th>
<th>SALES_FRI</th>
</tr>
</thead>
<tbody>
<tr>
<td>176</td>
<td>6</td>
<td>2000</td>
<td>3000</td>
<td>4000</td>
<td>5000</td>
<td>6000</td>
</tr>
</tbody>
</table>

```
DESC SALES_INFO
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Null?</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPLOYEE_ID</td>
<td></td>
<td>NUMBER(6)</td>
</tr>
<tr>
<td>WEEK</td>
<td></td>
<td>NUMBER(2)</td>
</tr>
<tr>
<td>SALES</td>
<td></td>
<td>NUMBER(0,2)</td>
</tr>
</tbody>
</table>

```
SELECT * FROM sales_info;
```

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>WEEK</th>
<th>SALES</th>
</tr>
</thead>
<tbody>
<tr>
<td>176</td>
<td>6</td>
<td>2000</td>
</tr>
<tr>
<td>176</td>
<td>6</td>
<td>3000</td>
</tr>
<tr>
<td>176</td>
<td>6</td>
<td>4000</td>
</tr>
<tr>
<td>176</td>
<td>6</td>
<td>5000</td>
</tr>
<tr>
<td>176</td>
<td>6</td>
<td>6000</td>
</tr>
</tbody>
</table>

Observe in the preceding example that using a pivoting INSERT, one row from the SALES_SOURCE_DATA table is converted into five records for the relational table, SALES_INFO.
External Tables

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

Creating an External Table

You create external tables using the `ORGANIZATION EXTERNAL` clause of the `CREATE TABLE` statement. You are not in fact creating a table. Rather, you are creating metadata in the data dictionary that you can use to access external data. The `ORGANIZATION` clause lets you specify the order in which the data rows of the table are stored. By specifying `EXTERNAL` in the `ORGANIZATION` clause, you indicate that the table is a read-only table located outside the database.

`TYPE access_driver_type` indicates the access driver of the external table. The access driver is the Application Programming Interface (API) that interprets the external data for the database. If you do not specify `TYPE`, Oracle uses the default access driver, `ORACLE_LOADER`.

The `REJECT LIMIT` clause lets you specify how many conversion errors can occur during a query of the external data before an Oracle error is returned and the query is aborted. The default value is 0.

`DEFAULT DIRECTORY` lets you specify one or more default directory objects corresponding to directories on the file system where the external data sources may reside. Default directories can also be used by the access driver to store auxiliary files such as error logs. Multiple default directories are permitted to facilitate load balancing on multiple disk drives.

The optional `ACCESS PARAMETERS` clause lets you assign values to the parameters of the specific access driver for this external table. Oracle does not interpret anything in this clause. It is up to the access driver to interpret this information in the context of the external data.

The `LOCATION` clause lets you specify one external locator for each external data source. Usually the `location_specifier` is a file, but it need not be. Oracle does not interpret this clause. It is up to the access driver to interpret this information in the context of the external data.
Example of Creating an External Table

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

```sql
CREATE DIRECTORY emp_dir AS '/flat_files';
```

**Example of Creating an External Table**

Use the `CREATE DIRECTORY` statement to create a directory object. A directory object specifies an alias for a directory on the server's file system where an external data source resides. You can use directory names when referring to an external data source, rather than hard-code the operating system pathname, for greater file management flexibility.

You must have `CREATE ANY DIRECTORY` system privileges to create directories. When you create a directory, you are automatically granted the `READ` object privilege and can grant `READ` privileges to other users and roles. The DBA can also grant this privilege to other users and roles.

**Syntax**

```sql
CREATE [OR REPLACE] DIRECTORY AS 'path_name';
```

In the syntax:

- **OR REPLACE** Specify `OR REPLACE` to re-create the directory database object if it already exists. You can use this clause to change the definition of an existing directory without dropping, re-creating, and regranting database object privileges previously granted on the directory. Users who had previously been granted privileges on a redefined directory can still access the directory without being regranted the privileges.
- **directory** Specify the name of the directory object to be created. The maximum length of directory is 30 bytes. You cannot qualify a directory object with a schema name.
- **'path_name'** Specify the full pathname of the operating system directory on the server where the files are located. The single quotes are required, with the result that the path name is case sensitive.
Example of Creating an External Table (continued)

Assume that there is a flat file that has records in the following format:

10, jones, 11-Dec-1934
20, smith, 12-Jun-1972

Records are delimited by new lines, and the fields are all terminated by a ",". The name of the file is:

/flat_files/emp1.txt

To convert this file as the data source for an external table, whose metadata will reside in the database, you need to perform the following steps:

1. Create a directory object `emp_dir` as follows:
   ```sql
   CREATE DIRECTORY emp_dir AS '/flat_files' ;
   ```
2. Run the `CREATE TABLE` command shown in the slide.

The example in the slide illustrates the table specification to create an external table for the file:

```
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.

In the example, the `TYPE` specification is given only to illustrate its use. If not specified, `ORACLE_LOADER` is the default access driver. The `ACCESS PARAMETERS` provide values to parameters of the specific access driver and are interpreted by the access driver, not by the Oracle Server.

The `PARALLEL` clause enables five parallel execution servers to simultaneously scan the external data sources (files) when executing the `INSERT INTO TABLE` statement. For example, if `PARALLEL=5` were specified, then more that one parallel execution server could be working on a data source. Because external tables can be very large, for performance reasons it is advisable to specify the `PARALLEL` clause, or a parallel hint for the query.
Example of Defining External Tables (continued)

The `REJECT LIMIT` clause specifies that if more than 200 conversion errors occur during a query of the external data, the query is aborted and an error returned. These conversion errors can arise when the access driver tries to transform the data in the data file to match the external table definition.

Once the `CREATE TABLE` command executes successfully, the external table `OLDEMP` can be described, queried upon like a relational table.

```sql
DESC oldemp
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Null?</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPNO</td>
<td></td>
<td>NUMBER</td>
</tr>
<tr>
<td>EMPNAME</td>
<td></td>
<td>CHAR(20)</td>
</tr>
<tr>
<td>BIRTHDATE</td>
<td></td>
<td>DATE</td>
</tr>
</tbody>
</table>

In the following example, the `INSERT INTO TABLE` statement generates a dataflow from the external data source to the Oracle SQL engine where data is processed. As data is extracted from the external table, it is transparently converted by the `ORACLE_LOADER` access driver from its external representation into an equivalent Oracle native representation. The `INSERT` statement inserts data from the external table `OLDEMP` into the `BIRTHDAYS` table:

```sql
INSERT INTO birthdays(empno, empname, birthdate)
    SELECT empno, empname, birthdate FROM oldemp;

2 rows created.
```

We can now select from the `BIRTHDAYS` table.

```sql
SELECT * FROM birthdays;
```

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>EMPNAME</th>
<th>BIRTHDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>jones</td>
<td>11-DEC-1934 00:00:00</td>
</tr>
<tr>
<td>20</td>
<td>smith</td>
<td>12-JUN-1972 00:00:00</td>
</tr>
</tbody>
</table>

2 rows selected.
Querying External Table

An external table does not describe any data that is stored in the database. Nor does it describe how data is stored in the external source. Instead, it describes how the external table layer needs to present the data to the server. It is the responsibility of the access driver and the external table layer to do the necessary transformations required on the data in the data file so that it matches the external table definition.

When the database server needs to access data in an external source, it calls the appropriate access driver to get the data from an external source in a form that the database server expects.

It is important to remember that the description of the data in the data source is separate from the definition of the external table. The source file can contain more or fewer fields than there are columns in the table. Also, the data types for fields in the data source can be different from the columns in the table. The access driver takes care of ensuring the data from the data source is processed so that it matches the definition of the external table.
CREATE INDEX with CREATE TABLE Statement

In the example in the slide, the CREATE INDEX clause is used with the CREATE TABLE statement to create a primary key index explicitly. This is an enhancement provided with Oracle 9i. You can now name your indexes at the time of PRIMARY key creation, unlike before where the Oracle Server would create an index, but you did not have any control over the name of the index. The following example illustrates this:

```
CREATE TABLE 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>

CREATE INDEX with CREATE TABLE Statement

Observe that the Oracle Server gives a name to the Index that it creates for the PRIMARY KEY column. But this name is cryptic and not easily understood. With Oracle 9i, you can now name your PRIMARY KEY column indexes, as you create the table with the CREATE TABLE statement. However, prior to Oracle 9i, if you named your primary key constraint at the time of constraint creation, the index would also be created with the same name as the constraint name.
In this lesson, you should have learned how to use the following enhancements to DML and DDL statements:

- The `INSERT...SELECT` statement can be used to insert rows into multiple tables as part of a single DML statement.
- External tables can be created.
- Indexes can be named using the `CREATE INDEX` statement along with the `CREATE TABLE` statement.

Oracle 9i introduces the following types of multitable `INSERT` statements.

- Unconditional `INSERT`
- Conditional `ALL INSERT`
- Conditional `FIRST INSERT`
- Pivoting `INSERT`

Use the `external_table_clause` to create an external table, which is a read-only table whose metadata is stored in the database but whose data is stored outside the database. External tables let you query data without first loading it into the database.

With Oracle9i, you can name your `PRIMARY KEY` column indexes as you create the table with the `CREATE TABLE` statement.
Practice 20 Overview

This practice covers the following topics:

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

Practice 20 Overview

In this practice, you write multitable inserts and use the CREATE INDEX command at the time of table creation, along with the CREATE TABLE command.
Practice 20

1. Run the `cre_sal_history.sql` script in the Labs folder to create the `SAL_HISTORY` table.

2. Display the structure of the `SAL_HISTORY` table.

<table>
<thead>
<tr>
<th>Name</th>
<th>Null?</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPLOYEE_ID</td>
<td></td>
<td>NUMBER(6)</td>
</tr>
<tr>
<td>HIRE_DATE</td>
<td></td>
<td>DATE</td>
</tr>
<tr>
<td>SALARY</td>
<td></td>
<td>NUMBER(8,2)</td>
</tr>
</tbody>
</table>

3. Run the `cre_mgr_history.sql` script in the Labs folder to create the `MGR_HISTORY` table.

4. Display the structure of the `MGR_HISTORY` table.

<table>
<thead>
<tr>
<th>Name</th>
<th>Null?</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPLOYEE_ID</td>
<td></td>
<td>NUMBER(6)</td>
</tr>
<tr>
<td>MANAGER_ID</td>
<td></td>
<td>NUMBER(6)</td>
</tr>
<tr>
<td>SALARY</td>
<td></td>
<td>NUMBER(8,2)</td>
</tr>
</tbody>
</table>

5. Run the `cre_special_sal.sql` script in the Labs folder to create the `SPECIAL_SAL` table.

6. Display the structure of the `SPECIAL_SAL` table.

<table>
<thead>
<tr>
<th>Name</th>
<th>Null?</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPLOYEE_ID</td>
<td></td>
<td>NUMBER(6)</td>
</tr>
<tr>
<td>SALARY</td>
<td></td>
<td>NUMBER(8,2)</td>
</tr>
</tbody>
</table>

7. a. Write a query to do the following:
   - Retrieve the details of the employee ID, hire date, salary, and manager ID of those employees whose employee ID is less than 125 from the `EMPLOYEES` table.
   - If the salary is more than $20,000, insert the details of employee ID and salary into the `SPECIAL_SAL` table.
   - Insert the details of employee ID, hire date, salary into the `SAL_HISTORY` table.
   - Insert the details of the employee ID, manager ID, and salary into the `MGR_HISTORY` table.
b. Display the records from the `SPECIAL_SAL` table.

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>24000</td>
</tr>
</tbody>
</table>


c. Display the records from the `SAL_HISTORY` table.

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>HIRE_DATE</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>21-SEP-89</td>
<td>17000</td>
</tr>
<tr>
<td>102</td>
<td>13-JAN-93</td>
<td>17000</td>
</tr>
<tr>
<td>103</td>
<td>03-JAN-90</td>
<td>9000</td>
</tr>
<tr>
<td>104</td>
<td>21-MAY-91</td>
<td>6000</td>
</tr>
<tr>
<td>107</td>
<td>07-FEB-99</td>
<td>4200</td>
</tr>
<tr>
<td>124</td>
<td>16-NOV-99</td>
<td>5800</td>
</tr>
</tbody>
</table>

6 rows selected.

d. Display the records from the `MGR_HISTORY` table.

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>MANAGER_ID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>100</td>
<td>17000</td>
</tr>
<tr>
<td>102</td>
<td>100</td>
<td>17000</td>
</tr>
<tr>
<td>103</td>
<td>102</td>
<td>9000</td>
</tr>
<tr>
<td>104</td>
<td>103</td>
<td>6000</td>
</tr>
<tr>
<td>107</td>
<td>103</td>
<td>4200</td>
</tr>
<tr>
<td>124</td>
<td>100</td>
<td>5800</td>
</tr>
</tbody>
</table>

6 rows selected.
Practice 20 (continued)

8. a. Run the `cre_sales_source_data.sql` script in the Labs folder to create the `SALES_SOURCE_DATA` table.

   b. Run the `ins_sales_source_data.sql` script in the Labs folder to insert records into the `SALES_SOURCE_DATA` table.

   c. Display the structure of the `SALES_SOURCE_DATA` table.

<table>
<thead>
<tr>
<th>Name</th>
<th>Null?</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPLOYEE_ID</td>
<td></td>
<td>NUMBER(6)</td>
</tr>
<tr>
<td>WEEK_ID</td>
<td></td>
<td>NUMBER(2)</td>
</tr>
<tr>
<td>SALES_MON</td>
<td></td>
<td>NUMBER(6,2)</td>
</tr>
<tr>
<td>SALES_TUE</td>
<td></td>
<td>NUMBER(6,2)</td>
</tr>
<tr>
<td>SALES_WED</td>
<td></td>
<td>NUMBER(6,2)</td>
</tr>
<tr>
<td>SALES_THUR</td>
<td></td>
<td>NUMBER(6,2)</td>
</tr>
<tr>
<td>SALES_FRI</td>
<td></td>
<td>NUMBER(6,2)</td>
</tr>
</tbody>
</table>

   d. Display the records from the `SALES_SOURCE_DATA` table.

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>WEEK_ID</th>
<th>SALES_MON</th>
<th>SALES_TUE</th>
<th>SALES_WED</th>
<th>SALES_THUR</th>
<th>SALES_FRI</th>
</tr>
</thead>
<tbody>
<tr>
<td>178</td>
<td>6</td>
<td>1750</td>
<td>2200</td>
<td>1500</td>
<td>1500</td>
<td>3000</td>
</tr>
</tbody>
</table>

e. Run the `cre_sales_info.sql` script in the Labs folder to create the `SALES_INFO` table.

f. Display the structure of the `SALES_INFO` table.

<table>
<thead>
<tr>
<th>Name</th>
<th>Null?</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPLOYEE_ID</td>
<td></td>
<td>NUMBER(6)</td>
</tr>
<tr>
<td>WEEK</td>
<td></td>
<td>NUMBER(2)</td>
</tr>
<tr>
<td>SALES</td>
<td></td>
<td>NUMBER(6,2)</td>
</tr>
</tbody>
</table>
Practice 20 (continued)

g. Write a query to do the following:
Retrieve the details of employee ID, week ID, sales on Monday, sales on Tuesday, sales on Wednesday, sales on Thursday, and sales on Friday from the SALES_SOURCE_DATA table.
Build a transformation such that each record retrieved from the SALES_SOURCE_DATA table is converted into multiple records for the SALES_INFO table.
**Hint:** Use a pivoting `INSERT` statement.

h. Display the records from the SALES_INFO table.

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>WEEK</th>
<th>SALES</th>
</tr>
</thead>
<tbody>
<tr>
<td>178</td>
<td>6</td>
<td>1750</td>
</tr>
<tr>
<td>178</td>
<td>6</td>
<td>2200</td>
</tr>
<tr>
<td>178</td>
<td>6</td>
<td>1500</td>
</tr>
<tr>
<td>178</td>
<td>6</td>
<td>1500</td>
</tr>
<tr>
<td>178</td>
<td>6</td>
<td>3000</td>
</tr>
</tbody>
</table>

5 rows selected.

9. a. Create the DEPT_NAMED_INDEX table based on the following table instance chart. Name the index for the PRIMARY KEY column as DEPT_PK_IDX.

<table>
<thead>
<tr>
<th>COLUMN Name</th>
<th>Deptno</th>
<th>Dname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Key</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Datatype</td>
<td>Number</td>
<td>VARCHAR2</td>
</tr>
<tr>
<td>Length</td>
<td>4</td>
<td>30</td>
</tr>
</tbody>
</table>

b. Query the USER_INDEXES table to display the INDEX_NAME for the DEPT_NAMED_INDEX table.

<table>
<thead>
<tr>
<th>INDEX_NAME</th>
<th>TABLE_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEPT_PK_IDX</td>
<td>DEPT_NAMED_INDEX</td>
</tr>
</tbody>
</table>
Practice Solutions
Practice 1 Solutions

1. Initiate an iSQL*Plus session using the user ID and password provided by the instructor.
2. iSQL*Plus commands access the database.
   False
3. The following SELECT statement executes successfully:
   True
   
   ```sql
   SELECT last_name, job_id, salary AS Sal
   FROM   employees;
   ```

4. The following SELECT statement executes successfully:
   True
   
   ```sql
   SELECT *
   FROM   job_grades;
   ```

5. There are four coding errors in this statement. Can you identify them?
   ```sql
   SELECT    employee_id, last_name 
   sal x 12  ANNUAL SALARY
   FROM      employees;
   ```
   - The EMPLOYEES table does not contain a column called `sal`. The column is called `SALARY`.
   - The multiplication operator is `*`, not `x`, as shown in line 2.
   - The `ANNUAL SALARY` alias cannot include spaces. The alias should read `ANNUAL_SALARY` or be enclosed in double quotation marks.
   - A comma is missing after the column, `LAST_NAME`.

6. Show the structure of the DEPARTMENTS table. Select all data from the DEPARTMENTS table.
   ```sql
   DESCRIBE departments
   SELECT *
   FROM   departments;
   ```

7. Show the structure of the EMPLOYEES table. Create a query to display the last name, job code, hire date, and employee number for each employee, with employee number appearing first. Save your SQL statement to a file named `lab1_7.sql`.
   ```sql
   DESCRIBE employees
   SELECT employee_id, last_name, job_id, hire_date
   FROM   employees;
   ```
8. Run your query in the file lab1_7.sql.

   SELECT employee_id, last_name, job_id, hire_date
   FROM employees;

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

   SELECT DISTINCT job_id
   FROM employees;

If you have time, complete the following exercises:

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

   SELECT employee_id "Emp ", last_name "Employee",
   job_id "Job", hire_date "Hire Date"
   FROM employees;

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

   SELECT last_name||', '||job_id "Employee and Title"
   FROM employees;

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

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

   SELECT employee_id || ', ' || first_name || ', ' || last_name
   || ', ' || email || ', ' || phone_number || ', ' || job_id
   || ', ' || manager_id || ', ' || hire_date || ', '
   || salary || ', ' || commission_pct || ', ' || department_id
   THE_OUTPUT
   FROM employees;
Practice 2 Solutions

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

   ```sql
   SELECT last_name, salary
   FROM employees
   WHERE salary > 12000;
   ```

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

   ```sql
   SELECT last_name, department_id
   FROM employees
   WHERE employee_id = 176;
   ```

3. Modify lab2_1.sql to display the last name and salary for all employees whose salary is not in the range of $5,000 and $12,000. Place your SQL statement in a text file named lab2_3.sql.

   ```sql
   SELECT last_name, salary
   FROM employees
   WHERE salary NOT BETWEEN 5000 AND 12000;
   ```

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

   ```sql
   SELECT last_name, job_id, hire_date
   FROM employees
   WHERE hire_date BETWEEN '20-Feb-1998' AND '01-May-1998'
   ORDER BY hire_date;
   ```
Practice 2 Solutions (continued)

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

```sql
SELECT last_name, department_id
FROM employees
WHERE department_id IN (20, 50)
ORDER BY last_name;
```

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

```sql
SELECT last_name "Employee", salary "Monthly Salary"
FROM employees
WHERE salary BETWEEN 5000 AND 12000
AND department_id IN (20, 50);
```

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

```sql
SELECT last_name, hire_date
FROM employees
WHERE hire_date LIKE '%94';
```

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

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

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

```sql
SELECT last_name, salary, commission_pct
FROM employees
WHERE commission_pct IS NOT NULL
ORDER BY salary DESC, commission_pct DESC;
```
Practice 2 Solutions (continued)

If you have time, complete the following exercises.

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

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

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

   ```sql
   SELECT   last_name
   FROM     employees
   WHERE    last_name LIKE '%a%'
   AND    last_name LIKE '%e%';
   ```

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

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

   ```sql
   SELECT   last_name, job_id, salary
   FROM     employees
   WHERE    job_id IN ('SA_REP', 'ST_CLERK')
   AND    salary NOT IN (2500, 3500, 7000);
   ```

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

   ```sql
   SELECT   last_name "Employee", salary "Monthly Salary",
            commission_pct
   FROM     employees
   WHERE    commission_pct = .20;
   ```
Practice 3 Solutions

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

   ```sql
   SELECT   sysdate "Date"
   FROM     dual;
   ```

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

   ```sql
   SELECT   employee_id, last_name, salary,
            ROUND(salary * 1.15, 0) "New Salary"
   FROM     employees;
   ```

3. Run your query in the file lab3_2.sql.

   ```sql
   SELECT   employee_id, last_name, salary,
            ROUND(salary * 1.15, 0) "New Salary"
   FROM     employees;
   ```

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

   ```sql
   SELECT   employee_id, last_name, salary,
            ROUND(salary * 1.15, 0) "New Salary",
            ROUND(salary * 1.15, 0) - salary "Increase"
   FROM     employees;
   ```

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

   ```sql
   SELECT      INITCAP(last_name) "Name",
               LENGTH(last_name) "Length"
   FROM        employees
   WHERE       last_name LIKE 'J%'
               OR       last_name LIKE 'M%'
               OR       last_name LIKE 'A%'
   ORDER BY    last_name;
   ```
Practice 3 Solutions (continued)

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

Note: Your results will differ.

```
SELECT   last_name, ROUND(MONTHS_BETWEEN
                   (SYSDATE, hire_date)) MONTHS_WORKED
FROM     employees
ORDER BY MONTHS_BETWEEN(SYSDATE, hire_date);
```

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

```
SELECT   last_name || ' earns ' || TO_CHAR(salary, 'fm$99,999.00') || ' monthly but wants ' || TO_CHAR(salary * 3, 'fm$99,999.00') || '.' "Dream Salaries"
FROM     employees;
```

If you have time, complete the following exercises:

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

```
SELECT   last_name, LPAD(salary, 15, '$') SALARY
FROM     employees;
```

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

```
SELECT   last_name, hire_date,
          TO_CHAR(NEXT_DAY(ADD_MONTHS(hire_date, 6),'MONDAY'),
          'fmDay, "the" Ddspth "of" Month, YYYY') REVIEW
FROM     employees;
```

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

```
SELECT   last_name, hire_date,
          TO_CHAR(hire_date, 'DAY') DAY
FROM     employees
ORDER BY TO_CHAR(hire_date - 1, 'd');
```
If you want an extra challenge, complete the following exercises:

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

   \begin{verbatim}
   SELECT last_name, NVL(TO_CHAR(commission_pct), 'No Commission') COMM
   FROM employees;
   \end{verbatim}

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

   \begin{verbatim}
   SELECT rpad(last_name, 8) || ' ' || rpad(' ', salary/1000+1, '*') EMPLOYEES_AND_THEIR_SALARIES
   FROM employees
   ORDER BY salary DESC;
   \end{verbatim}

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

   \begin{center}
   \begin{tabular}{|c|c|}
   \hline
   JOB & GRADE \\
   \hline
   AD_PRES & A \\
   ST_MAN & B \\
   IT_PROG & C \\
   SA_REP & D \\
   ST_CLERK & E \\
   None of the above & 0 \\
   \hline
   \end{tabular}
   \end{center}

   \begin{verbatim}
   SELECT job_id, decode (job_id, 'ST_CLERK', 'E', 'SA_REP', 'D', 'IT_PROG', 'C', 'ST_MAN', 'B', 'AD_PRES', 'A', '0') GRADE
   FROM employees;
   \end{verbatim}
14. Rewrite the statement in the preceding question using the CASE syntax.

```sql
SELECT job_id, CASE job_id
    WHEN 'ST_CLERK' THEN 'E'
    WHEN 'SA_REP' THEN 'D'
    WHEN 'IT_PROG' THEN 'C'
    WHEN 'ST_MAN' THEN 'B'
    WHEN 'AD_PRES' THEN 'A'
    ELSE '0'
END AS GRADE
FROM employees;
```
Practice 4 Solutions

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

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

2. Create a unique listing of all jobs that are in department 30. Include the location of department 90 in the output.

   ```sql
   SELECT DISTINCT job_id, location_id
   FROM employees, departments
   WHERE employees.department_id = departments.department_id
   AND employees.department_id = 80;
   ```

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

   ```sql
   SELECT e.last_name, d.department_name, d.location_id, l.city
   FROM employees e, departments d, locations l
   WHERE e.department_id = d.department_id
   AND d.location_id = l.location_id
   AND e.commission_pct IS NOT NULL;
   ```

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

   ```sql
   SELECT last_name, department_name
   FROM employees, departments
   WHERE employees.department_id = departments.department_id
   AND last_name LIKE '%a%';
   ```
5. Write a query to display the last name, job, department number, and department name for all employees who work in Toronto.

```sql
SELECT e.last_name, e.job_id, e.department_id, d.department_name
FROM employees e JOIN departments d
JOIN locations l
ON (d.location_id = l.location_id)
WHERE LOWER(l.city) = 'toronto';
```

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

```sql
SELECT w.last_name "Employee", w.employee_id "EMP#",
       m.last_name "Manager", m.employee_id "Mgr#"
FROM employees w JOIN employees m
ON (w.manager_id = m.employee_id);
```
Practice 4 Solutions (continued)

7. Modify `lab4_6.sql` to display all employees including King, who has no manager.
   Place your SQL statement in a text file named `lab4_7.sql`. Run the query in `lab4_7.sql`
   ```sql
   SELECT w.last_name "Employee", w.employee_id "EMP#",
   m.last_name "Manager", m.employee_id "Mgr#"
   FROM employees w
   LEFT OUTER JOIN employees m
   ON (w.manager_id = m.employee_id);
   ```

If you have time, complete the following exercises.

8. Create a query that displays employee last names, department numbers, and all the
   employees who work in the same department as a given employee. Give each column an appropriate
   label.
   ```sql
   SELECT e.department_id department, e.last_name employee,
   c.last_name colleague
   FROM   employees e JOIN employees c
   ON     (e.department_id = c.department_id)
   WHERE   e.employee_id <> c.employee_id
   ORDER BY e.department_id, e.last_name, c.last_name;
   ```

9. Show the structure of the `JOB_GRADES` table. Create a query that displays the name, job,
   department name, salary, and grade for all employees.
   ```sql
   DESC JOB_GRADES
   SELECT e.last_name, e.job_id, d.department_name,
   e.salary, j.grade_level
   FROM   employees e, departments d, job_grades j
   WHERE  e.department_id = d.department_id
   AND    e.salary BETWEEN j.lowest_sal AND j.highest_sal;
   -- OR
   SELECT e.last_name, e.job_id, d.department_name,
   e.salary, j.grade_level
   FROM   employees e JOIN departments d
   ON     (e.department_id = d.department_id)
   JOIN   job_grades j
   ON     (e.salary BETWEEN j.lowest_sal AND j.highest_sal);
   ```
Practice 4 Solutions (continued)

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

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

```
SELECT e.last_name, e.hire_date
FROM   employees e, employees davies
WHERE davies.last_name = 'Davies'
AND    davies.hire_date < e.hire_date
-- OR
SELECT e.last_name, e.hire_date
FROM   employees e JOIN employees davies
ON     (davies.last_name = 'Davies')
WHERE    davies.hire_date < e.hire_date;
```

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

```
SELECT w.last_name, w.hire_date, m.last_name, m.hire_date
FROM   employees w, employees m
WHERE  w.manager_id = m.employee_id
AND    w.hire_date <  m.hire_date;
-- OR
SELECT w.last_name, w.hire_date, m.last_name, m.hire_date
FROM   employees w JOIN employees m
ON    (w.manager_id = m.employee_id)
WHERE    w.hire_date <  m.hire_date;
```
Practice 5 Solutions

Determine the validity of the following three statements. Circle either True or False.

1. Group functions work across many rows to produce one result.
   True

2. Group functions include nulls in calculations.
   False. Group functions ignore null values. If you want to include null values, use the NVL function.

3. The WHERE clause restricts rows prior to inclusion in a group calculation.
   True

4. Display the highest, lowest, sum, and average salary of all employees. Label the columns Maximum, Minimum, Sum, and Average, respectively. Round your results to the nearest whole number. Place your SQL statement in a text file named lab5_6.sql.

   ```sql
   SELECT   ROUND(MAX(salary),0) "Maximum",
            ROUND(MIN(salary),0) "Minimum",
            ROUND(SUM(salary),0) "Sum",
            ROUND(AVG(salary),0) "Average"
   FROM     employees;
   ```

5. Modify the query in lab5_4.sql to display the minimum, maximum, sum, and average salary for each job type. Resave lab5_4.sql to lab5_5.sql. Run the statement in lab5_5.sql.

   ```sql
   SELECT   job_id, ROUND(MAX(salary),0) "Maximum",
            ROUND(MIN(salary),0) "Minimum",
            ROUND(SUM(salary),0) "Sum",
            ROUND(AVG(salary),0) "Average"
   FROM     employees
   GROUP BY job_id;
   ```
Practice 5 Solutions (continued)

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

   ```sql
   SELECT   job_id, COUNT(*)
   FROM     employees
   GROUP BY job_id;
   ```

7. Determine the number of managers without listing them. Label the column Number of Managers. Hint: Use the MANAGER_ID column to determine the number of managers.

   ```sql
   SELECT   COUNT(DISTINCT manager_id) "Number of Managers"
   FROM     employees;
   ```

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

   ```sql
   SELECT   MAX(salary) - MIN(salary) DIFFERENCE
   FROM     employees;
   ```

If you have time, complete the following exercises.

9. Display the manager number and the salary of the lowest paid employee for that manager. Exclude anyone whose manager is not known. Exclude any groups where the minimum salary is less than $6,000. Sort the output in descending order of salary.

   ```sql
   SELECT   manager_id, MIN(salary)
   FROM     employees
   WHERE    manager_id IS NOT NULL
   GROUP BY manager_id
   HAVING   MIN(salary) > 6000
   ORDER BY MIN(salary) DESC;
   ```

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

    ```sql
    SELECT   d.department_name "Name", d.location_id "Location",
             COUNT(*) "Number of People",
             ROUND(AVG(salary),2) "Salary"
    FROM     employees e, departments d
    WHERE    e.department_id = d.department_id
    GROUP BY d.department_name, d.location_id;
    ```
Practice 5 Solutions (continued)

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

11. Create a query that will display the total number of employees and, of that total, the number of employees hired in 1995, 1996, 1997, and 1998. Create appropriate column headings.

   ```sql
   SELECT COUNT(*) total,
           SUM(DECODE(TO_CHAR(hire_date, 'YYYY'), '1995', 1, 0)) "1995",
           SUM(DECODE(TO_CHAR(hire_date, 'YYYY'), '1996', 1, 0)) "1996",
           SUM(DECODE(TO_CHAR(hire_date, 'YYYY'), '1997', 1, 0)) "1997",
           SUM(DECODE(TO_CHAR(hire_date, 'YYYY'), '1998', 1, 0)) "1998"
   FROM employees;
   ```

12. Create a matrix query to display the job, the salary for that job based on department number, and the total salary for that job, for departments 20, 50, 80, and 90, giving each column an appropriate heading.

   ```sql
   SELECT job_id "Job",
           SUM(DECODE(department_id, 20, salary)) "Dept 20",
           SUM(DECODE(department_id, 50, salary)) "Dept 50",
           SUM(DECODE(department_id, 80, salary)) "Dept 80",
           SUM(DECODE(department_id, 90, salary)) "Dept 90",
           SUM(salary) "Total"
   FROM employees
   GROUP BY job_id;
   ```
Practice 6 Solutions

1. Write a query to display the last name and hire date of any employee in the same department as Zlotkey. Exclude Zlotkey.

   ```sql
   SELECT last_name, hire_date
   FROM   employees
   WHERE  department_id = (SELECT department_id
                           FROM   employees
                           WHERE  last_name = 'Zlotkey')
   AND    last_nae <> 'Zlotkey';
   ```

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

   ```sql
   SELECT employee_id, last_name
   FROM   employees
   WHERE  salary > (SELECT AVG(salary)
                    FROM   employees);
   ```

3. Write a query that displays the employee numbers and last names of all employees who work in a department with any employee whose last name contains a u. Place your SQL statement in a text file named `lab6_3.sql`. Run your query.

   ```sql
   SELECT employee_id, last_name
   FROM   employees
   WHERE  department_id IN (SELECT department_id
                               FROM   employees
                               WHERE  last_name like '%u%');
   ```

4. Display the last name, department number, and job ID of all employees whose department location ID is 1700.

   ```sql
   SELECT last_name, department_id, job_id
   FROM   employees
   WHERE  department_id IN (SELECT department_id
                              FROM   departments
                              WHERE  location_id = 1700);
Practice 6 Solutions (continued)

5. Display the last name and salary of every employee who reports to King.

```
SELECT last_name, salary
FROM employees
WHERE manager_id = (SELECT employee_id
    FROM employees
    WHERE last_name = 'King');
```

6. Display the department number, last name, and job ID for every employee in the Executive department.

```
SELECT department_id, last_name, job_id
FROM employees
WHERE department_id IN (SELECT department_id
    FROM departments
    WHERE department_name = 'Executive');
```

If you have time, complete the following exercises:

7. Modify the query in lab6_3.sql to display the employee numbers, last names, and salaries of all employees who earn more than the average salary and who work in a department with any employee with a u in their name. Resave lab6_3.sql to lab6_7.sql. Run the statement in lab6_7.sql.

```
SELECT employee_id, last_name, salary
FROM employees
WHERE department_id IN (SELECT department_id
    FROM employees
    WHERE last_name like '%u%')
    AND salary > (SELECT AVG(salary)
        FROM employees);
```
Practice 7 Solutions

Determine whether the following statements are true or false:

1. The following statement is correct:

   `DEFINE & p_val = 100`

   False
   
   The correct use of `DEFINE` is `DEFINE p_val=100`. The `&` is used within the SQL code.

2. The `DEFINE` command is a SQL command.

   False
   
   The `DEFINE` command is an *iSQL* command.

3. Write a script file to display the employee last name, job, and hire date for all employees who started between a given range. Concatenate the name and job together, separated by a space and comma, and label the column Employees. Use the `DEFINE` command to provide the two ranges. Use the format MM/DD/YYYY. Save the script file as `lab7_3.sql`.

   ```sql
   SET ECHO OFF
   SET VERIFY OFF
   DEFINE low_date = 01/01/1998
   DEFINE high_date = 01/01/1999
   SELECT  last_name ||', '|| job_id EMPLOYEES, hire_date
   FROM    employees
   WHERE   hire_date BETWEEN TO_DATE('&low_date', 'MM/DD/YYYY')
           AND TO_DATE('&high_date', 'MM/DD/YYYY')
   /
   UNDEFINE low_date
   UNDEFINE high_date
   SET VERIFY ON
   SET ECHO ON
   ```
4. Write a script to display the employee last name, job, and department name for a given location. The search condition should allow for case-insensitive searches of the department location. Save the script file as lab7_4.sql.

```sql
SET ECHO OFF
SET VERIFY OFF
COLUMN last_name HEADING "EMPLOYEE NAME"
COLUMN department_name HEADING "DEPARTMENT NAME"
SELECT e.last_name, e.job_id, d.department_name
FROM employees e, departments d, locations l
WHERE e.department_id = d.department_id
AND l.location_id = d.location_id
AND l.city = INITCAP('&p_location')
/
COLUMN last_name CLEAR
COLUMN department_name CLEAR
SET VERIFY ON
SET ECHO ON
```
Practice 7 Solutions (continued)

5. Modify the code in lab7_4.sql to create a report containing the department name, employee last name, hire date, salary, and each employee’s annual salary for all employees in a given location. Label the columns DEPARTMENT NAME, EMPLOYEE NAME, START DATE, SALARY, and ANNUAL SALARY, placing the labels on multiple lines. Resave the script as lab7_5.sql and execute the commands in the script.

```sql
SET ECHO OFF
SET FEEDBACK OFF
SET VERIFY OFF
BREAK ON department_name
COLUMN department_name HEADING "DEPARTMENT NAME"
COLUMN last_name HEADING "EMPLOYEE NAME"
COLUMN hire_date HEADING "START DATE"
COLUMN salary HEADING "SALARY" FORMAT $99,990.00
COLUMN asal HEADING "ANNUAL SALARY" FORMAT $99,990.00
SELECT d.department_name, e.last_name, e.hire_date,
    e.salary, e.salary*12 asal
FROM   departments d, employees e, locations l
WHERE  e.department_id = d.department_id
AND    d.location_id = l.location_id
AND    l.city = '&p_location'
ORDER BY d.department_name
/
COLUMN department_name CLEAR
COLUMN last_name CLEAR
COLUMN hire_date CLEAR
COLUMN salary CLEAR
COLUMN asal CLEAR
CLEAR BREAK
SET VERIFY ON
SET FEEDBACK ON
SET ECHO ON
```
Practice 8 Solutions

Insert data into the MY_EMPLOYEE table.

1. Run the statement in the lab8_1.sql script to build the MY_EMPLOYEE table that will be used for the lab.

   ```sql
   CREATE TABLE my_employee
   (id    NUMBER(4) CONSTRAINT my_employee_id_nn NOT NULL,
    last_name VARCHAR2(25),
    first_name VARCHAR2(25),
    userid  VARCHAR2(8),
    salary  NUMBER(9,2));
   ```

2. Describe the structure of the MY_EMPLOYEE table to identify the column names.

   ```sql
   DESCRIBE my_employee
   ```

3. Add the first row of data to the MY_EMPLOYEE table from the following sample data. Do not list the columns in the INSERT clause.

   ```sql
   INSERT INTO my_employee
   VALUES (1, 'Patel', 'Ralph', 'rpatel', 895);
   ```

4. Populate the MY_EMPLOYEE table with the second row of sample data from the preceding list. This time, list the columns explicitly in the INSERT clause.

   ```sql
   INSERT INTO my_employee (id, last_name, first_name, userid, salary)
   VALUES (2, 'Dancs', 'Betty', 'bdancs', 860);
   ```

5. Confirm your addition to the table.

   ```sql
   SELECT * FROM my_employee;
   ```
6. Write an insert statement in a text file named `loademp.sql` to load rows into the `MY_EMPLOYEE` table. Concatenate the first letter of the first name and the first seven characters of the last name to produce the userid.

```sql
SET ECHO OFF
SET VERIFY OFF
INSERT INTO my_employee
VALUES (&p_id, '&p_last_name', '&p_first_name',
        lower(substr('&p_first_name', 1, 1) ||
           substr('&p_last_name', 1, 7)), &p_salary);

SET VERIFY ON
SET ECHO ON
```

7. Populate the table with the next two rows of sample data by running the insert statement in the script that you created.

```sql
SET ECHO OFF
SET VERIFY OFF
INSERT INTO my_employee
VALUES (&p_id, '&p_last_name', '&p_first_name',
        lower(substr('&p_first_name', 1, 1) ||
           substr('&p_last_name', 1, 7)), &p_salary);

SET VERIFY ON
SET ECHO ON
```

8. Confirm your additions to the table.

```sql
SELECT *
FROM my_employee;
```

9. Make the data additions permanent.

```sql
COMMIT;
```
Practice 8 Solutions (continued)

Update and delete data in the MY_EMPLOYEE table.

10. Change the last name of employee 3 to Drexler.

```sql
UPDATE my_employee
SET last_name = 'Drexler'
WHERE id = 3;
```

11. Change the salary to 1000 for all employees with a salary less than 900.

```sql
UPDATE my_employee
SET salary = 1000
WHERE salary < 900;
```

12. Verify your changes to the table.

```sql
SELECT last_name, salary
FROM my_employee;
```

13. Delete Betty Dancs from the MY_EMPLOYEE table.

```sql
DELETE
FROM my_employee
WHERE last_name = 'Dancs';
```

14. Confirm your changes to the table.

```sql
SELECT *
FROM my_employee;
```

15. Commit all pending changes.

```sql
COMMIT;
```

Control data transaction to the MY_EMPLOYEE table.

16. Populate the table with the last row of sample data by modifying the statements in the script that you created in step 6. Run the statements in the script.

```sql
SET ECHO OFF
SET VERIFY OFF
INSERT INTO my_employee
VALUES (&p_id, '&p_last_name', '&p_first_name',
        lower(substr('&p_first_name', 1, 1) ||
              substr('&p_last_name', 1, 7)), &p_salary);
SET VERIFY ON
SET ECHO ON
```
Practice 8 Solutions (continued)

17. Confirm your addition to the table.

    SELECT *  
    FROM my_employee;

18. Mark an intermediate point in the processing of the transaction.

    SAVEPOINT step_18;

19. Empty the entire table.

    DELETE  
    FROM   my_employee;

20. Confirm that the table is empty.

    SELECT *  
    FROM   my_employee;

21. Discard the most recent DELETE operation without discarding the earlier INSERT operation.

    ROLLBACK TO step_18;

22. Confirm that the new row is still intact.

    SELECT *  
    FROM   my_employee;

23. Make the data addition permanent.

    COMMIT;
Practice 9 Solutions

1. Create the **DEPT** table based on the following table instance chart. Place the syntax in a script called `lab9_1.sql`, then execute the statement in the script to create the table. Confirm that the table is created.

   ```sql
   CREATE TABLE dept
   (id NUMBER(7),
   name VARCHAR2(25));
   DESCRIBE dept
   ```

2. Populate the **DEPT** table with data from the **DEPARTMENTS** table. Include only columns that you need.

   ```sql
   INSERT INTO dept
   SELECT department_id, department_name
   FROM departments;
   ```

3. Create the **EMP** table based on the following table instance chart. Place the syntax in a script called `lab9_3.sql`, and then execute the statement in the script to create the table. Confirm that the table is created.

   ```sql
   Column Name | ID | LAST_NAME | FIRST_NAME | DEPT_ID
   Key Type    |    |          |           |      
   Nulls/Unique|    |          |           |      
   FK Table    |    |          |           |      
   FK Column   |    |          |           |      
   Data type   | Number | VARCHAR2 | VARCHAR2 | Number 
   Length      | 7  | 25       | 25        | 7     
   ```
Practice 9 Solutions (continued)

```sql
CREATE TABLE emp
(id NUMBER(7),
 last_name VARCHAR2(25),
 first_name VARCHAR2(25),
 dept_id NUMBER(7));

DESCRIBE emp

4. Modify the EMP table to allow for longer employee last names. Confirm your modification.

   ALTER TABLE emp
   MODIFY (last_name VARCHAR2(50));

   DESCRIBE emp

5. Confirm that both the DEPT and EMP tables are stored in the data dictionary. (Hint: USER_TABLES)

   SELECT table_name
   FROM user_tables
   WHERE table_name IN ('DEPT', 'EMP');

6. Create the EMPLOYEES2 table based on the structure of the EMPLOYEES table. Include only the EMPLOYEE_ID, FIRST_NAME, LAST_NAME, SALARY, and DEPARTMENT_ID columns. Name the columns in your new table ID, FIRST_NAME, LAST_NAME, SALARY, and DEPT_ID, respectively.

   CREATE TABLE employees2 AS
   SELECT employee_id id, first_name, last_name, salary,
   department_id dept_id
   FROM employees;

7. Drop the EMP table.

   DROP TABLE emp;

8. Rename the EMPLOYEES2 table to EMP.

   RENAME employees2 TO emp;
```
Practice 9 Solutions (continued)

9. Add a comment to the DEPT and EMP table definitions describing the tables. Confirm your additions in the data dictionary.

   COMMENT ON TABLE emp IS 'Employee Information';
   COMMENT ON TABLE dept IS 'Department Information';
   SELECT * FROM user_tab_comments
   WHERE table_name = 'DEPT'
   OR table_name = 'EMP';

10. Drop the FIRST_NAME column from the EMP table. Confirm your modification by checking the description of the table.

    ALTER TABLE emp
    DROP COLUMN FIRST_NAME;

    DESCRIBE emp

11. In the EMP table, mark the DEPT_ID column in the EMP table as UNUSED. Confirm your modification by checking the description of the table.

    ALTER TABLE emp
    SET UNUSED (dept_id);

    DESCRIBE emp

12. Drop all the UNUSED columns from the EMP table. Confirm your modification by checking the description of the table.

    ALTER TABLE emp
    DROP UNUSED COLUMNS;

    DESCRIBE emp
Practice 10 Solutions

1. Add a table-level PRIMARY KEY constraint to the EMP table on the ID column. The constraint should be named at creation. Name the constraint my_emp_id_pk.

   ```sql
   ALTER TABLE emp
   ADD CONSTRAINT my_emp_id_pk PRIMARY KEY (id);
   ```

2. Create a PRIMARY KEY constraint to the DEPT table using the ID column. The constraint should be named at creation. Name the constraint my_deptid_pk.

   ```sql
   ALTER TABLE dept
   ADD CONSTRAINT my_deptid_pk PRIMARY KEY(id);
   ```

3. Add a column DEPT_ID to the EMP table. Add a foreign key reference on the EMP table that ensures that the employee is not assigned to a nonexistent department. Name the constraint my_emp_dept_id_fk.

   ```sql
   ALTER TABLE emp
   ADD (dept_id NUMBER(7));
   ```

   ```sql
   ALTER TABLE emp
   ADD CONSTRAINT my_emp_dept_id_fk
   FOREIGN KEY (dept_id) REFERENCES dept(id);
   ```

4. Confirm that the constraints were added by querying the USER_CONSTRAINTS view. Note the types and names of the constraints. Save your statement text in a file called lab10_4.sql.

   ```sql
   SELECT constraint_name, constraint_type
   FROM user_constraints
   WHERE table_name IN ('EMP', 'DEPT');
   ```

5. Display the object names and types from the USER_OBJECTS data dictionary view for the EMP and DEPT tables. Notice that the new tables and a new index were created.

   ```sql
   SELECT object_name, object_type
   FROM user_objects
   WHERE object_name LIKE 'EMP%'
   OR object_name LIKE 'DEPT%';
   ```

If you have time, complete the following exercise:

6. Modify the EMP table. Add a COMMISSION column of NUMBER data type, precision 2, scale 2. Add a constraint to the commission column that ensures that a commission value is greater than zero.

   ```sql
   ALTER TABLE EMP
   ADD commission NUMBER(2,2)
   CONSTRAINT my_emp_comm_ck CHECK (commission >= 0);
   ```
Practice 11 Solutions

1. Create a view called EMPLOYEES_VU based on the employee numbers, employee names, and department numbers from the EMPLOYEES table. Change the heading for the employee name to EMPLOYEE.
   
   CREATE OR REPLACE VIEW employees_vu AS
   SELECT employee_id, last_name employee, department_id
   FROM employees;

2. Display the contents of the EMPLOYEES_VU view.
   
   SELECT *
   FROM employees_vu;

3. Select the view name and text from the USER_VIEWS data dictionary view.
   
   Note: Another view already exists. The EMP_DETAILS_VIEW was created as part of your schema.
   
   Note: To see more contents of a LONG column, use the iSQL*Plus command SET LONG n, where n is the value of the number of characters of the LONG column that you want to see.
   
   SET LONG 600
   SELECT view_name, text
   FROM user_views;

4. Using your EMPLOYEES_VU view, enter a query to display all employee names and department numbers.
   
   SELECT employee, department_id
   FROM employees_vu;

5. Create a view named DEPT50 that contains the employee numbers, employee last names, and department numbers for all employees in department 50. Label the view columns EMPNO, EMPLOYEE, and DEPTNO. Do not allow an employee to be reassigned to another department through the view.
   
   CREATE VIEW dept50 AS
   SELECT employee_id empno, last_name employee, department_id deptno
   FROM employees
   WHERE department_id = 50
   WITH CHECK OPTION CONSTRAINT emp_dept_50;
Practice 11 Solutions (continued)

6. Display the structure and contents of the DEPT50 view.

   DESCRIPT  dept50
   SELECT   *
   FROM     dept50;

7. Attempt to reassign Matos to department 80.

   UPDATE   dept50
   SET      deptno = 80
   WHERE    employee = 'Matos';

If you have time, complete the following exercise:

8. Create a view called SALARY_VU based on the employee last names, department names, salaries, and salary grades for all employees. Use the EMPLOYEES, DEPARTMENTS, and JOB_GRADES tables. Label the columns Employee, Department, Salary, and Grade, respectively.

   CREATE OR REPLACE VIEW salary_vu
   AS
   SELECT e.last_name "Employee",
       d.department_name "Department",
       e.salary "Salary",
       j.grade_level "Grades"
   FROM   employees e,
       departments d,
       job_grades j
   WHERE  e.department_id = d.department_id
   AND    e.salary BETWEEN j.lowest_sal and j.highest_sal;
Practice 12 Solutions

1. Create a sequence to be used with the primary key column of the DEPT table. The sequence should start at 200 and have a maximum value of 1000. Have your sequence increment by ten numbers. Name the sequence DEPT_ID_SEQ.

   CREATE SEQUENCE dept_id_seq
   START WITH 200
   INCREMENT BY 10
   MAXVALUE 1000;

2. Write a query in a script to display the following information about your sequences: sequence name, maximum value, increment size, and last number. Name the script lab12_2.sql. Run the statement in your script.

   SELECT   sequence_name, max_value, increment_by, last_number
            FROM     user_sequences;

3. Write a script to insert two rows into the DEPT table. Name your script lab12_3.sql. Be sure to use the sequence that you created for the ID column. Add two departments named Education and Administration. Confirm your additions. Run the commands in your script.

   INSERT INTO dept
   VALUES (dept_id_seq.nextval, 'Education');

   INSERT INTO dept
   VALUES (dept_id_seq.nextval, 'Administration');

4. Create a nonunique index on the foreign key column (DEPT_ID) in the EMP table.

   CREATE INDEX emp_dept_id_idx ON emp (dept_id);

5. Display the indexes and uniqueness that exist in the data dictionary for the EMP table. Save the statement into a script named lab12_5.sql.

   SELECT   index_name, table_name, uniqueness
            FROM     user_indexes
            WHERE    table_name = 'EMP';
Practice 13 Solutions

1. What privilege should a user be given to log on to the Oracle Server? Is this a system or an object privilege?

   The CREATE SESSION system privilege

2. What privilege should a user be given to create tables?

   The CREATE TABLE privilege

3. If you create a table, who can pass along privileges to other users on your table?

   You can, or anyone you have given those privileges to by using the WITH GRANT OPTION.

4. You are the DBA. You are creating many users who require the same system privileges. What should you use to make your job easier?

   Create a role containing the system privileges and grant the role to the users

5. What command do you use to change your password?

   The ALTER USER statement

6. Grant another user access to your DEPARTMENTS table. Have the user grant you query access to his or her DEPARTMENTS table.

   Team 2 executes the GRANT statement.

   ```sql
   GRANT select
   ON departments
   TO <user1>;
   ```

   Team 1 executes the GRANT statement.

   ```sql
   GRANT select
   ON departments
   TO <user2>;
   ```

   WHERE user1 is the name of team 1 and user2 is the name of team 2.

7. Query all the rows in your DEPARTMENTS table.

   ```sql
   SELECT *
   FROM departments;
   ```
Practice 13 Solutions (continued)

8. Add a new row to your DEPARTMENTS table. Team 1 should add Education as department number 500. Team 2 should add Human Resources department number 510. Query the other team’s table.

Team 1 executes this INSERT statement.
INSERT INTO departments(department_id, department_name)
VALUES (200, 'Education');
COMMIT;

Team 2 executes this INSERT statement.
INSERT INTO departments(department_id, department_name)
VALUES (210, 'Administration');
COMMIT;

9. Create a synonym for the other team’s DEPARTMENTS table.

Team 1 creates a synonym named team2.
CREATE SYNONYM team2
FOR <user2>.DEPARTMENTS;

Team 2 creates a synonym named team1.
CREATE SYNONYM team1
FOR <user1>. DEPARTMENTS;

10. Query all the rows in the other team’s DEPARTMENTS table by using your synonym.

Team 1 executes this SELECT statement.
SELECT *
FROM team2;

Team 2 executes this SELECT statement.
SELECT *
FROM team1;
11. Query the `USER_TABLES` data dictionary to see information about the tables that you own.

```sql
SELECT  table_name
FROM    user_tables;
```

12. Query the `ALL_TABLES` data dictionary view to see information about all the tables that you can access. Exclude tables that you own.

```sql
SELECT  table_name, owner
FROM    all_tables
WHERE   owner <> '<your account>';
```

13. Revoke the `SELECT` privilege from the other team.

```
Team 1 revokes the privilege.
    REVOKE select
    ON     departments
    FROM   user2;

Team 2 revokes the privilege.
    REVOKE select
    ON     departments
    FROM   user1;
```
Practice 14 Solutions

1. Create the tables based on the following table instance charts. Choose the appropriate data types and be sure to add integrity constraints.

   a. Table name: MEMBER

```
CREATE TABLE member
  (member_id NUMBER(10) CONSTRAINT member_member_id_pk PRIMARY KEY,
   last_name VARCHAR2(25) CONSTRAINT member_last_name_nn NOT NULL,
   first_name VARCHAR2(25),
   address VARCHAR2(100),
   city VARCHAR2(30),
   phone VARCHAR2(15),
   join_date DATE DEFAULT SYSDATE CONSTRAINT member_join_date_nn NOT NULL);
```

<table>
<thead>
<tr>
<th>Column Name</th>
<th>MEMBER_ID</th>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>ADDRESS</th>
<th>CITY</th>
<th>PHONE</th>
<th>JOIN_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Type</td>
<td>PK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Null/Unique</td>
<td>NN, U</td>
<td>NN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NN</td>
</tr>
<tr>
<td>Default Value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>System Date</td>
</tr>
<tr>
<td>Data Type</td>
<td>NUMBER</td>
<td>VARCHAR2</td>
<td>VARCHAR2</td>
<td>VARCHAR2</td>
<td>VARCHAR2</td>
<td>VARCHAR2</td>
<td>DATE</td>
</tr>
<tr>
<td>Length</td>
<td>10</td>
<td>25</td>
<td>25</td>
<td>100</td>
<td>30</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>
b. Table name: TITLE

<table>
<thead>
<tr>
<th>Column_Name</th>
<th>TITLE_ID</th>
<th>TITLE</th>
<th>DESCRIPTION</th>
<th>RATING</th>
<th>CATEGORY</th>
<th>RELEASE_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Type</td>
<td>PK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Null/Unique</td>
<td>NN,U</td>
<td>NN</td>
<td>NN</td>
<td>G, PG, R, NC17, NR</td>
<td>DRAMA, COMEDY, ACTION, CHILD, SCIFI, DOCUMENTARY</td>
<td></td>
</tr>
<tr>
<td>Check</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Type</td>
<td>NUMBER</td>
<td>VARCHAR2</td>
<td>VARCHAR2</td>
<td>VARCHAR2</td>
<td>VARCHAR2</td>
<td>DATE</td>
</tr>
<tr>
<td>Length</td>
<td>10</td>
<td>60</td>
<td>400</td>
<td>4</td>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>

CREATE TABLE title
(title_id NUMBER(10)
    CONSTRAINT title_title_id_pk PRIMARY KEY,
    title VARCHAR2(60)
    CONSTRAINT title_title_nn NOT NULL,
    description VARCHAR2(400)
    CONSTRAINT title_description_nn NOT NULL,
    rating VARCHAR2(4)
    CONSTRAINT title_rating_ck CHECK (rating IN ('G', 'PG', 'R', 'NC17', 'NR')),
    category VARCHAR2(20),
    CONSTRAINT title_category_ck CHECK (category IN ('DRAMA', 'COMEDY', 'ACTION', 'CHILD', 'SCIFI', 'DOCUMENTARY')),
    release_date DATE);
Practice 14 Solutions (continued)

c. Table name: `TITLE_COPY`

<table>
<thead>
<tr>
<th>Column Name</th>
<th>COPY_ID</th>
<th>TITLE_ID</th>
<th>STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Type</td>
<td>PK</td>
<td>PK,FK</td>
<td></td>
</tr>
<tr>
<td>Null/Unique</td>
<td>NN,U</td>
<td>NN,U</td>
<td>NN</td>
</tr>
<tr>
<td>Check</td>
<td></td>
<td></td>
<td>AVAILABLE, DESTROYED, RENTED, RESERVED</td>
</tr>
<tr>
<td>FK Ref Table</td>
<td>TITLE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FK Ref Column</td>
<td>TITLE_ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Type</td>
<td>NUMBER</td>
<td>NUMBER</td>
<td>VARCHAR2</td>
</tr>
<tr>
<td>Length</td>
<td>10</td>
<td>10</td>
<td>15</td>
</tr>
</tbody>
</table>

```
CREATE TABLE title_copy
     (copy_id NUMBER(10),
      title_id NUMBER(10)
 CONSTRAINT title_copy_title_if_fk REFERENCES title(title_id),
      status VARCHAR2(15) NOT NULL
 CONSTRAINT title_copy_status_nn
 CONSTRAINT title_copy_status_ck CHECK (status IN
     ('AVAILABLE', 'DESTROYED','RENTED', 'RESERVED')),
 CONSTRAINT title_copy_copy_id_title_id_pk
     PRIMARY KEY (copy_id, title_id));
```
d. Table name: RENTAL

<table>
<thead>
<tr>
<th>Column Name</th>
<th>BOOK_DATE</th>
<th>MEMBER_ID</th>
<th>COPY_ID</th>
<th>ACT_RET_DATE</th>
<th>EXP_RET_DATE</th>
<th>TITLE_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Type</td>
<td>PK</td>
<td>PK,FK1</td>
<td>PK,FK2</td>
<td></td>
<td></td>
<td>PK,FK2</td>
</tr>
<tr>
<td>Default Value</td>
<td>System Date</td>
<td></td>
<td></td>
<td></td>
<td>System Date + 2 days</td>
<td></td>
</tr>
<tr>
<td>FK Ref Table</td>
<td>MEMBER</td>
<td>TITLE_COPY</td>
<td>COPY_ID</td>
<td></td>
<td></td>
<td>TITLE_COPY</td>
</tr>
<tr>
<td>FK Ref Column</td>
<td>MEMBER_ID</td>
<td>COPY_ID</td>
<td></td>
<td></td>
<td></td>
<td>TITLE_ID</td>
</tr>
<tr>
<td>Data Type</td>
<td>DATE</td>
<td>NUMBER</td>
<td>NUMBER</td>
<td>DATE</td>
<td>DATE</td>
<td>NUMBER</td>
</tr>
<tr>
<td>Length</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CREATE TABLE rental
(book_date DATE DEFAULT SYSDATE,
member_id NUMBER(10)
CONSTRAINT rental_member_id_fk
REFERENCES member(member_id),
copy_id NUMBER(10),
act_ret_date DATE,
exp_ret_date DATE DEFAULT SYSDATE + 2,
title_id NUMBER(10),
CONSTRAINT rental_book_date_copy_title_pk
PRIMARY KEY (book_date, member_id,
copy_id,title_id),
CONSTRAINT rental_copy_id_title_id_fk
FOREIGN KEY (copy_id, title_id)
REFERENCES title_copy(copy_id, title_id));
e. Table name: RESERVATION

```
CREATE TABLE reservation
(res_date DATE,
 member_id NUMBER(10)
    CONSTRAINT reservation_member_id
       REFERENCES member(member_id),
 title_id NUMBER(10)
    CONSTRAINT reservation_title_id
       REFERENCES title(title_id),
 CONSTRAINT reservation_resdate_mem_tit_pk PRIMARY KEY
   (res_date, member_id, title_id));
```
Practice 14 Solutions (continued)

2. Verify that the tables and constraints were created properly by checking the data dictionary.

```sql
SELECT table_name
FROM user_tables
WHERE table_name IN ('MEMBER', 'TITLE', 'TITLE_COPY',
                      'RENTAL', 'RESERVATION');

SELECT constraint_name, constraint_type, table_name
FROM user_constraints
WHERE table_name IN ('MEMBER', 'TITLE', 'TITLE_COPY',
                      'RENTAL', 'RESERVATION');
```

3. Create sequences to uniquely identify each row in the MEMBER table and the TITLE table.
   a. Member number for the MEMBER table: start with 101; do not allow caching of the values. Name the sequence MEMBER_ID_SEQ.

   ```sql
   CREATE SEQUENCE member_id_seq
   START WITH 101
   NOCACHE;
   ```

   b. Title number for the TITLE table: start with 92; no caching. Name the sequence TITLE_ID_SEQ.

   ```sql
   CREATE SEQUENCE title_id_seq
   START WITH 92
   NOCACHE;
   ```

   c. Verify the existence of the sequences in the data dictionary.

   ```sql
   SELECT sequence_name, increment_by, last_number
   FROM user_sequences
   WHERE sequence_name IN ('MEMBER_ID_SEQ', 'TITLE_ID_SEQ');
   ```
4. Add data to the tables. Create a script for each set of data to add.
   a. Add movie titles to the TITLE table. Write a script to enter the movie information. Save the statements in a script named lab14_4a.sql. Use the sequences to uniquely identify each title. Enter the release dates in the DD-MON-YYYY format. Remember that single quotation marks in a character field must be specially handled. Verify your additions.

   ```sql
   SET ECHO OFF
   INSERT INTO title(title_id, title, description, rating, category, release_date)
   VALUES (title_id_seq.NEXTVAL, 'Willie and Christmas Too', 'All of Willie''s friends make a Christmas list for Santa, but Willie has yet to add his own wish list.', 'G', 'CHILD', TO_DATE('05-OCT-1995','DD-MON-YYYY'))
 /
   INSERT INTO title(title_id, title, description, rating, category, release_date)
   VALUES (title_id_seq.NEXTVAL, 'Alien Again', 'Yet another installment of science fiction history. Can the heroine save the planet from the alien life form?', 'R', 'SCIFI', TO_DATE('19-MAY-1995','DD-MON-YYYY'))
 /
   INSERT INTO title(title_id, title, description, rating, category, release_date)
   VALUES (title_id_seq.NEXTVAL, 'The Glob', 'A meteor crashes near a small American town and unleashes carnivorous goo in this classic.', 'NR', 'SCIFI', TO_DATE('12-AUG-1995','DD-MON-YYYY'))
 /
   INSERT INTO title(title_id, title, description, rating, category, release_date)
   VALUES (title_id_seq.NEXTVAL, 'My Day Off', 'With a little luck and a lot ingenuity, a teenager skips school for a day in New York.', 'PG', 'COMEDY', TO_DATE('12-JUL-1995','DD-MON-YYYY'))
 /
   ...
   COMMIT
   /
   SET ECHO ON
   
   SELECT  title
   FROM    title;
   ```
### Practice 14 Solutions (continued)

<table>
<thead>
<tr>
<th>Title</th>
<th>Description</th>
<th>Rating</th>
<th>Category</th>
<th>Release Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Willie and Christmas Too</td>
<td>All of Willie’s friends make a Christmas list for Santa, but Willie has yet to add his own wish list.</td>
<td>G</td>
<td>CHILD</td>
<td>05-OCT-1995</td>
</tr>
<tr>
<td>Alien Again</td>
<td>Yet another installation of science fiction history. Can the heroine save the planet from the alien life form?</td>
<td>R</td>
<td>SCIFI</td>
<td>19-MAY-1995</td>
</tr>
<tr>
<td>The Glob</td>
<td>A meteor crashes near a small American town and unleashes carnivorous goo in this classic.</td>
<td>NR</td>
<td>SCIFI</td>
<td>12-AUG-1995</td>
</tr>
<tr>
<td>My Day Off</td>
<td>With a little luck and a lot of ingenuity, a teenager skips school for a day in New York</td>
<td>PG</td>
<td>COMEDY</td>
<td>12-JUL-1995</td>
</tr>
<tr>
<td>Miracles on Ice</td>
<td>A six-year-old has doubts about Santa Claus, but she discovers that miracles really do exist.</td>
<td>PG</td>
<td>DRAMA</td>
<td>12-SEP-1995</td>
</tr>
<tr>
<td>Soda Gang</td>
<td>After discovering a cache of drugs, a young couple find themselves pitted against a vicious gang.</td>
<td>NR</td>
<td>ACTION</td>
<td>01-JUN-1995</td>
</tr>
</tbody>
</table>
b. Add data to the MEMBER table. Place the insert statements in a script named `lab14_4b.sql`. Execute commands in the script. Be sure to use the sequence to add the member numbers.

<table>
<thead>
<tr>
<th>First_Name</th>
<th>Last_Name</th>
<th>Address</th>
<th>City</th>
<th>Phone</th>
<th>Join_Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carmen</td>
<td>Velasquez</td>
<td>283 King Street</td>
<td>Seattle</td>
<td>206-899-6666</td>
<td>08-MAR-1990</td>
</tr>
<tr>
<td>LaDoris</td>
<td>Ngao</td>
<td>5 Modrany</td>
<td>Bratislava</td>
<td>586-355-8882</td>
<td>08-MAR-1990</td>
</tr>
<tr>
<td>Midori</td>
<td>Nagayama</td>
<td>68 Via Centrale</td>
<td>Sao Paolo</td>
<td>254-852-5764</td>
<td>17-JUN-1991</td>
</tr>
<tr>
<td>Mark</td>
<td>Quick-to-See</td>
<td>6921 King Way</td>
<td>Lagos</td>
<td>63-559-7777</td>
<td>07-APR-1990</td>
</tr>
<tr>
<td>Audry</td>
<td>Ropeburn</td>
<td>86 Chu Street</td>
<td>Hong Kong</td>
<td>41-559-87</td>
<td>18-JAN-1991</td>
</tr>
<tr>
<td>Molly</td>
<td>Urguhart</td>
<td>3035 Laurier</td>
<td>Quebec</td>
<td>418-542-9988</td>
<td>18-JAN-1991</td>
</tr>
</tbody>
</table>

```
SET ECHO OFF
SET VERIFY OFF
INSERT INTO member(member_id, first_name, last_name, address, city, phone, join_date)
VALUES (member_id_seq.NEXTVAL, '&first_name', '&last_name', '&address', '&city', '&phone', TO_DATE('&join_date', 'DD-MM-YYYY'));
COMMIT;
SET VERIFY ON
SET ECHO ON
```
c. Add the following movie copies in the `TITLE_COPY` table:

*Note:* Have the `TITLE_ID` numbers available for this exercise.

<table>
<thead>
<tr>
<th>Title</th>
<th>Copy_Id</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Willie and Christmas Too</td>
<td>1</td>
<td>AVAILABLE</td>
</tr>
<tr>
<td>Alien Again</td>
<td>1</td>
<td>AVAILABLE</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>RENTED</td>
</tr>
<tr>
<td>The Glob</td>
<td>1</td>
<td>AVAILABLE</td>
</tr>
<tr>
<td>My Day Off</td>
<td>1</td>
<td>AVAILABLE</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>RENTED</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>RENTED</td>
</tr>
<tr>
<td>Miracles on Ice</td>
<td>1</td>
<td>AVAILABLE</td>
</tr>
<tr>
<td>Soda Gang</td>
<td>1</td>
<td>AVAILABLE</td>
</tr>
</tbody>
</table>

```sql
INSERT INTO title_copy(copy_id, title_id, status)
VALUES (1, 92, 'AVAILABLE');
INSERT INTO title_copy(copy_id, title_id, status)
VALUES (1, 93, 'AVAILABLE');
INSERT INTO title_copy(copy_id, title_id, status)
VALUES (2, 93, 'RENTED');
INSERT INTO title_copy(copy_id, title_id, status)
VALUES (1, 94, 'AVAILABLE');
INSERT INTO title_copy(copy_id, title_id, status)
VALUES (1, 95, 'AVAILABLE');
INSERT INTO title_copy(copy_id, title_id, status)
VALUES (2, 95, 'RENTED');
INSERT INTO title_copy(copy_id, title_id, status)
VALUES (3, 95, 'RENTED');
INSERT INTO title_copy(copy_id, title_id, status)
VALUES (1, 96, 'AVAILABLE');
INSERT INTO title_copy(copy_id, title_id, status)
VALUES (1, 97, 'AVAILABLE');
```
Practice 14 Solutions (continued)

d. Add the following rentals to the RENTAL table:

Note: Title number may be different depending on sequence number.

<table>
<thead>
<tr>
<th>Title_Id</th>
<th>Copy_Id</th>
<th>Member_Id</th>
<th>Book_date</th>
<th>Exp_Ret_Date</th>
<th>Act_Ret_Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>92</td>
<td>1</td>
<td>101</td>
<td>3 days ago</td>
<td>1 day ago</td>
<td>2 days ago</td>
</tr>
<tr>
<td>93</td>
<td>2</td>
<td>101</td>
<td>1 day ago</td>
<td>1 day from now</td>
<td></td>
</tr>
<tr>
<td>95</td>
<td>3</td>
<td>102</td>
<td>2 days ago</td>
<td>Today</td>
<td></td>
</tr>
<tr>
<td>97</td>
<td>1</td>
<td>106</td>
<td>4 days ago</td>
<td>2 days ago</td>
<td>2 days ago</td>
</tr>
</tbody>
</table>

```
INSERT INTO rental(title_id, copy_id, member_id, book_date, exp_ret_date, act_ret_date)
VALUES (92, 1, 101, sysdate-3, sysdate-1, sysdate-2);
INSERT INTO rental(title_id, copy_id, member_id, book_date, exp_ret_date, act_ret_date)
VALUES (93, 2, 101, sysdate-1, sysdate-1, NULL);
INSERT INTO rental(title_id, copy_id, member_id, book_date, exp_ret_date, act_ret_date)
VALUES (95, 3, 102, sysdate-2, sysdate, NULL);
INSERT INTO rental(title_id, copy_id, member_id, book_date, exp_ret_date, act_ret_date)
VALUES (97, 1, 106, sysdate-4, sysdate-2, sysdate-2);
COMMIT;
```
5. Create a view named TITLE_AVAIL to show the movie titles and the availability of each copy and its expected return date if rented. Query all rows from the view. Order the results by title.

```sql
CREATE VIEW title_avail AS
SELECT   t.title, c.copy_id, c.status, r.exp_ret_date
FROM     title t, title_copy c, rental r
WHERE    t.title_id = c.title_id
AND      c.copy_id = r.copy_id(+)
AND      c.title_id = r.title_id(+);
SELECT   *
FROM     title_avail
ORDER BY title, copy_id;
```

6. Make changes to data in the tables.
   a. Add a new title. The movie is “Interstellar Wars,” which is rated PG and classified as a scifi movie. The release date is 07-JUL-77. The description is “Futuristic interstellar action movie. Can the rebels save the humans from the evil empire?” Be sure to add a title copy record for two copies.

```sql
INSERT INTO title(title_id, title, description, rating, category, release_date)
VALUES (title_id_seq.NEXTVAL, 'Interstellar Wars', 'Futuristic interstellar action movie. Can the rebels save the humans from the evil Empire?', 'PG', 'SCIFI', '07-JUL-77');
INSERT INTO title_copy (copy_id, title_id, status)
VALUES (1, 98, 'AVAILABLE');
INSERT INTO title_copy (copy_id, title_id, status)
VALUES (2, 98, 'AVAILABLE');
```

   b. Enter two reservations. One reservation is for Carmen Velasquez, who wants to rent “Interstellar Wars.” The other is for Mark Quick-to-See, who wants to rent “Soda Gang.”

```sql
INSERT INTO reservation (res_date, member_id, title_id)
VALUES (SYSDATE, 101, 98);
INSERT INTO reservation (res_date, member_id, title_id)
VALUES (SYSDATE, 104, 97);
```
Practice 14 Solutions (continued)

c. Customer Carmen Velasquez rents the movie “Interstellar Wars,” copy 1. Remove her reservation for the movie. Record the information about the rental. Allow the default value for the expected return date to be used. Verify that the rental was recorded by using the view you created.

```
INSERT INTO rental(title_id, copy_id, member_id)
VALUES (98,1,101);
UPDATE title_copy
SET status= 'RENTED'
WHERE title_id = 98
AND copy_id = 1;
DELETE
FROM reservation
WHERE member_id = 101;
SELECT *
FROM title_avail
ORDER BY title, copy_id;
```

7. Make a modification to one of the tables.

a. Add a `PRICE` column to the `TITLE` table to record the purchase price of the video. The column should have a total length of eight digits and two decimal places. Verify your modifications.

```
ALTER TABLE title
ADD (price NUMBER(8,2));
DESCRIBE title
```
Practice 14 Solutions (continued)

b. Create a script named `lab14_7b.sql` that contains update statements that update each video with a price according to the following list. Run the commands in the script.

**Note:** Have the `TITLE_ID` numbers available for this exercise.

<table>
<thead>
<tr>
<th>Title</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Willie and Christmas Too</td>
<td>25</td>
</tr>
<tr>
<td>Alien Again</td>
<td>35</td>
</tr>
<tr>
<td>The Glob</td>
<td>35</td>
</tr>
<tr>
<td>My Day Off</td>
<td>35</td>
</tr>
<tr>
<td>Miracles on Ice</td>
<td>30</td>
</tr>
<tr>
<td>Soda Gang</td>
<td>35</td>
</tr>
<tr>
<td>Interstellar Wars</td>
<td>29</td>
</tr>
</tbody>
</table>

```sql
SET ECHO OFF
SET VERIFY OFF
DEFINE price=
DEFINE title_id=
UPDATE title
SET price = &price
WHERE title_id = &title_id;
SET VERIFY OFF
SET ECHO OFF
```

c. Ensure that in the future all titles contain a price value. Verify the constraint.

```sql
ALTER TABLE title
MODIFY (price CONSTRAINT title_price_nn NOT NULL);
SELECT constraint_name, constraint_type,
     search_condition
FROM user_constraints
WHERE table_name = 'TITLE';
```
Practice 14 Solutions (continued)

8. Create a report titled Customer History Report. This report contains each customer's history of renting videos. Be sure to include the customer name, movie rented, dates of the rental, and duration of rentals. Total the number of rentals for all customers for the reporting period. Save the commands that generate the report in a script file named lab14_8.sql.

```
SET ECHO OFF
SET VERIFY OFF
TITITLE 'Customer History Report'
BREAK ON member  SKIP 1 ON REPORT
SELECT m.first_name||' '||m.last_name MEMBER, t.title,
     r.book_date, r.act_ret_date - r.book_date DURATION
FROM member m, title t, rental r
WHERE r.member_id = m.member_id
     AND r.title_id = t.title_id
ORDER BY member;

CLEAR BREAK
TITITLE OFF
SET VERIFY ON
SET ECHO ON
```
Practice 15 Solutions

1. List the department IDs for departments that do not contain the job ID ST_CLERK, using SET operators.

```sql
SELECT department_id
FROM departments
MINUS
SELECT department_id
FROM employees
WHERE job_id = 'ST_CLERK';
```

2. Display the country ID and the name of the countries that have no departments located in them, using SET operators.

```sql
SELECT country_id, country_name
FROM countries
MINUS
SELECT l.country_id, c.country_name
FROM locations l, countries c
WHERE l.country_id = c.country_id;
```

3. Produce a list of jobs for departments 10, 50, and 20, in that order. Display job ID and department ID, using SET operators.

```sql
COLUMN dummy PRINT
SELECT job_id, department_id, 'x' dummy
FROM employees
WHERE department_id = 10
UNION
SELECT job_id, department_id, 'y'
FROM employees
WHERE department_id = 50
UNION
SELECT job_id, department_id, 'z'
FROM employees
WHERE department_id = 20
ORDER BY 3;
COLUMN dummy NOPRINT
```
Practice 15 Solutions (continued)

4. List the employee IDs and job IDs of those employees, who are currently in the job title that they have held once before during their tenure with the company.

   SELECT employee_id, job_id
   FROM employees
   INTERSECT
   SELECT employee_id, job_id
   FROM job_history;

5. Write a compound query that lists the following:
   a. Last names and department ID of all the employees from the EMPLOYEES table, irrespective of whether they belong to any department
   b. Department ID and department name of all the departments from the DEPARTMENTS table, irrespective of whether they have employees working in them

   SELECT last_name, department_id, TO_CHAR(null)
   FROM employees
   UNION
   SELECT TO_CHAR(null), department_id, department_name
   FROM departments;
Practice 16 Solutions

1. Alter the session to set the NLS_DATE_FORMAT to DD-MON-YYYY HH24:MI:SS.

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

2. a. Write queries to display the time zone offsets (TZ_OFFSET) for the following time zones.

   US/Pacific-New
   `SELECT TZ_OFFSET ('US/Pacific-New') from dual;`

   Singapore
   `SELECT TZ_OFFSET ('Singapore') from dual;`

   Egypt
   `SELECT TZ_OFFSET ('Egypt') from dual;`

   b. Alter the session to set the TIME_ZONE parameter value to the time zone offset of US/Pacific-New.

   `ALTER SESSION SET TIME_ZONE = '-8:00';`

   c. Display the CURRENT_DATE, CURRENT_TIMESTAMP, and LOCALTIMESTAMP for this session.

      Note: The output might be different based on the date when the command is executed.

      `SELECT CURRENT_DATE, CURRENT_TIMESTAMP, LOCALTIMESTAMP FROM DUAL;`

   d. Alter the session to set the TIME_ZONE parameter value to the time zone offset of Singapore.

      `ALTER SESSION SET TIME_ZONE = '+8:00';`

   e. Display the CURRENT_DATE, CURRENT_TIMESTAMP, LOCALTIMESTAMP for this session.

      Note: The output might be different based on the date when the command is executed.

      `SELECT CURRENT_DATE, CURRENT_TIMESTAMP, LOCALTIMESTAMP FROM DUAL;`

3. Write a query to display the DBTIMEZONE and SESSIONTIMEZONE.

   `SELECT DBTIMEZONE, SESSIONTIMEZONE FROM DUAL;`
4. Write a query to extract the YEAR from HIRE_DATE column of the EMPLOYEES table for those employees who work in department 80.

    SELECT last_name, EXTRACT (YEAR FROM HIRE_DATE)
    FROM employees
    WHERE department_id = 80;
Practice 17 Solutions

1. Write a query to display the following for those employees whose manager ID is less than 120:
   - Manager ID
   - Job ID and total salary for every job ID for employees who report to the same manager
   - Total salary of those managers
   - Total salary of those managers, irrespective of the job IDs

   ```sql
   SELECT manager_id, job_id, sum(salary)
   FROM employees
   WHERE manager_id < 120
   GROUP BY ROLLUP(manager_id, job_id);
   ```

2. Observe the output from question 1. Write a query using the GROUPING function to determine whether the NULL values in the columns corresponding to the GROUP BY expressions are caused by the ROLLUP operation.

   ```sql
   SELECT manager_id MGR, job_id JOB, sum(salary), GROUPING(manager_id), GROUPING(job_id)
   FROM employees
   WHERE manager_id < 120
   GROUP BY ROLLUP(manager_id, job_id);
   ```

3. Write a query to display the following for those employees whose manager ID is less than 120:
   - Manager ID
   - Job and total salaries for every job for employees who report to the same manager
   - Total salary of those managers
   - Cross-tabulation values to display the total salary for every job, irrespective of the manager
   - Total salary irrespective of all job titles

   ```sql
   SELECT manager_id, job_id, sum(salary)
   FROM employees
   WHERE manager_id < 120
   GROUP BY CUBE(manager_id, job_id);
   ```
Practice 17 Solutions (continued)

4. Observe the output from question 3. Write a query using the `GROUPING` function to determine whether the NULL values in the columns corresponding to the `GROUP BY` expressions are caused by the `CUBE` operation.

   ```sql
   SELECT manager_id MGR, job_id JOB, 
         sum(salary), GROUPING(manager_id), GROUPING(job_id) 
   FROM   employees 
   WHERE manager_id < 120 
   GROUP BY CUBE(manager_id, job_id);
   ```

5. Using `GROUPING SETS`, write a query to display the following groupings:
   - department_id, manager_id, job_id
   - department_id, job_id
   - Manager_id,  job_id

   The query should calculate the sum of the salaries for each of these groups.

   ```sql
   SELECT department_id, manager_id, job_id, SUM(salary) 
   FROM employees 
   GROUP BY 
   GROUPING SETS ((department_id, manager_id, job_id), 
                  (department_id, job_id), (manager_id, job_id));
   ```
Practice 18 Solutions

1. Write a query to display the last name, department number, and salary of any employee whose department number and salary both match the department number and salary of any employee who earns a commission.

   ```sql
   SELECT last_name, department_id, salary
   FROM   employees
   WHERE  (salary, department_id) IN
           (SELECT salary, department_id
            FROM   employees
            WHERE  commission_pct IS NOT NULL);
   ```

2. Display the last name, department name, and salary of any employee whose salary and commission match the salary and commission of any employee located in location ID1700.

   ```sql
   SELECT last_name, department_name, salary
   FROM   employees e, departments d
   WHERE  e.department_id = d.department_id
   AND    (salary, NVL(commission_pct,0)) IN
           (SELECT salary, NVL(commission_pct,0)
            FROM   employees e, departments d
            WHERE  e.department_id = d.department_id
            AND    d.location_id = 1700);
   ```

3. Create a query to display the last name, hire date, and salary for all employees who have the same salary and commission as Kochhar. Note: Do not display Kochhar in the result set.

   ```sql
   SELECT last_name, hire_date, salary
   FROM   employees
   WHERE  (salary, NVL(commission_pct,0)) IN
           (SELECT salary, NVL(commission_pct,0)
            FROM   employees
            WHERE  last_name = 'Kochhar')
   AND     last_name != 'Kochhar';
   ```

4. Create a query to display the employees who earn a salary that is higher than the salary of all of the sales managers (JOB_ID = 'SA_MAN'). Sort the results on salary from highest to lowest.

   ```sql
   SELECT last_name, job_id, salary
   FROM   employees
   WHERE  salary > ALL
           (SELECT salary
            FROM   employees
            WHERE  job_id = 'SA_MAN')
   ORDER BY salary DESC;
   ```

Introduction to Oracle9i: SQL A-59
Practice 18 Solutions (continued)

5. Display the details of the employee ID, last name, and department ID of those employees who live in cities whose name begins with T.

```
SELECT employee_id, last_name, department_id
FROM employees
WHERE department_id IN (SELECT department_id
                          FROM departments
                          WHERE location_id IN
                          (SELECT location_id
                           FROM locations
                           WHERE city LIKE 'T%'));
```

6. Write a query to find all employees who earn more than the average salary in their departments. Display last name, salary, department ID, and the average salary for the department. Sort by average salary. Use aliases for the columns retrieved by the query as shown in the sample output.

```
SELECT e.last_name ename, e.salary salary, e.department_id deptno, AVG(a.salary) dept_avg
FROM employees e, employees a
WHERE e.department_id = a.department_id
AND e.salary > (SELECT AVG(salary)
                FROM employees
                WHERE department_id = e.department_id )
GROUP BY e.last_name, e.salary, e.department_id
ORDER BY AVG(a.salary);
```

7. Find all employees who are not supervisors.
   a. First do this by using the NOT EXISTS operator.

```
SELECT outer.last_name
FROM employees outer
WHERE NOT EXISTS (SELECT 'X'
                  FROM employees inner
                  WHERE inner.manager_id =
                  outer.employee_id);
```
b. Can this be done by using the NOT IN operator? How, or why not?

```sql
SELECT outer.last_name
FROM employees outer
WHERE outer.employee_id
NOT IN (SELECT inner.manager_id
FROM employees inner);
```

This alternative solution is not a good one. The subquery picks up a NULL value, so the entire query returns no rows. The reason is that all conditions that compare a NULL value result in NULL. Whenever NULL values are likely to be part of the value set, do not use NOT IN as a substitute for NOT EXISTS.

8. Write a query to display the last names of the employees who earn less than the average salary in their departments.

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

9. Write a query to display the last names who have one or more coworkers in their departments with later hire dates but higher salaries.

```sql
SELECT last_name
FROM employees outer
WHERE EXISTS (SELECT 'X'
FROM employees inner
WHERE inner.department_id = outer.department_id
AND inner.hire_date > outer.hire_date
AND inner.salary > outer.salary);
```

10. Write a query to display the employee ID, last names of the employees, and department names of all employees.

    Note: Use a scalar subquery to retrieve the department name in the SELECT statement.

```sql
SELECT employee_id, last_name,
(SELECT department_name
FROM departments d
WHERE e.department_id =
    d.department_id ) department
FROM employees e
ORDER BY department;
```
11. Write a query to display the department names of those departments whose total salary cost is above one-eighth (1/8) of the total salary cost of the whole company. Use the WITH clause to write this query. Name the query SUMMARY.

WITH
    summary AS (  
        SELECT department_name, SUM(salary) AS dept_total  
        FROM employees, departments  
        WHERE employees.department_id =  
            departments.department_id  
        GROUP BY department_name)  
    SELECT department_name, dept_total  
    FROM summary  
    WHERE dept_total > (  
        SELECT SUM(dept_total) * 1/8  
        FROM summary)  
    ORDER BY dept_total DESC;
Practice 19 Solutions

1. Look at the following output. Is this output the result of a hierarchical query? Explain why or why not.
   a. Exhibit 1: This is not a hierarchical query; the report simply has a descending sort on SALARY.

Exhibit 2: This is not a hierarchical query; there are two tables involved.

Exhibit 3: Yes, this is most definitely a hierarchical query as it displays the tree structure representing the management reporting line from the EMPLOYEES table.

2. Produce a report showing an organization chart for Mourgos’s department. Print last names, salaries, and department IDs.
   ```sql
   SELECT last_name, salary, department_id
   FROM employees
   START WITH last_name = 'Mourgos'
   CONNECT BY PRIOR employee_id = manager_id;
   ```

3. Create a report that shows the hierarchy of the managers for the employee Lorentz. Display his immediate manager first.
   ```sql
   SELECT last_name
   FROM employees
   WHERE last_name != 'Lorentz'
   START WITH last_name = 'Lorentz'
   CONNECT BY PRIOR manager_id = employee_id;
   ```

4. Create an indented report showing the management hierarchy starting from the employee whose LAST_NAME is Kochhar. Print the employee’s last name, manager ID, and department ID. Give alias names to the columns as shown in the sample output.
   ```sql
   COLUMN name FORMAT A20
   SELECT LPAD(last_name, LENGTH(last_name)+(LEVEL*2)-2,'_')
        name, manager_id mgr, department_id deptno
   FROM employees
   START WITH last_name = 'Kochhar'
   CONNECT BY PRIOR employee_id = manager_id
   /
   COLUMN name CLEAR
Practice 19 Solutions (continued)

If you have time, complete the following exercises:

5. Produce a company organization chart that shows the management hierarchy. Start with the person at the top level, exclude all people with a job ID of IT_PROG, and exclude De Haan and those employees who report to De Hann.

```sql
SELECT last_name, employee_id, manager_id
FROM employees
WHERE job_id != 'IT_PROG'
START WITH manager_id IS NULL
CONNECT BY PRIOR employee_id = manager_id
AND last_name != 'De Haan';
```
Practice 20 Solutions

1. Run the `cre_sal_history.sql` script in the Labs folder to create the SAL_HISTORY table.
   
   ```
   @ \Labs\cre_sal_history.sql
   ```

2. Display the structure of the SAL_HISTORY table.
   
   ```
   DESC sal_history
   ```

3. Run the `cre_mgr_history.sql` script in the Labs folder to create the MGR_HISTORY table.
   
   ```
   @ \Labs\cre_mgr_history.sql
   ```

4. Display the structure of the MGR_HISTORY table.
   
   ```
   DESC mgr_history
   ```

5. Run the `cre_special_sal.sql` script in the Labs folder to create the SPECIAL_SAL table.
   
   ```
   @ \Labs\cre_special_sal.sql
   ```

6. Display the structure of the SPECIAL_SAL table.
   
   ```
   DESC special_sal
   ```

7. a. Write a query to do the following:
   
   - Retrieve the details of the employee ID, hire date, salary, and manager ID of those employees whose employee ID is less than 125 from the EMPLOYEES table.
   
   - If the salary is more than $20,000, insert the details of employee ID and salary into the SPECIAL_SAL table.
   
   - Insert the details of the employee ID, hire date, and salary into the SAL_HISTORY table.
   
   - Insert the details of the employee ID, manager ID, and SYSDATE into the MGR_HISTORY table.

   ```
   INSERT ALL
   WHEN SAL > 20000 THEN
   INTO special_empsal VALUES (EMPID, SAL)
   ELSE
   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 < 125;
   ```
b. Display the records from the SPECIAL_SAL table.
   
   ```sql
   SELECT * FROM special_sal;
   ```

c. Display the records from the SAL_HISTORY table.
   
   ```sql
   SELECT * FROM sal_history;
   ```

d. Display the records from the MGR_HISTORY table.
   
   ```sql
   SELECT * FROM mgr_history;
   ```

8. a. Run the `cre_sales_source_data.sql` script in the Labs folder to create the SALES_SOURCE_DATA table.

   ```sql
   @ Labs\cre_sales_source_data.sql
   ```

b. Run the `ins_sales_source_data.sql` script in the Labs folder to insert records into the SALES_SOURCE_DATA table.

   ```sql
   @ Labs\ins_sales_source_data.sql
   ```

c. Display the structure of the SALES_SOURCE_DATA table.

   ```sql
   DESC sales_source_data
   ```

d. Display the records from the SALES_SOURCE_DATA table.

   ```sql
   SELECT * FROM SALES_SOURCE_DATA;
   ```

e. Run the `cre_sales_info.sql` script in the Labs folder to create the SALES_INFO table.

   ```sql
   @ Labs\cre_sales_info.sql
   ```

f. Display the structure of the SALES_INFO table.

   ```sql
   DESC sales_info
   ```

g. Write a query to do the following:

   - Retrieve the details of the employee ID, week ID, sales on Monday, sales on Tuesday, sales on Wednesday, sales on Thursday, and sales on Friday from the SALES_SOURCE_DATA table.

   - Build a transformation such that each record retrieved from the SALES_SOURCE_DATA table is converted into multiple records for the SALES_INFO table.

   **Hint:** Use a pivoting `INSERT` statement.
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;

h. Display the records from the SALES_INFO table.
   SELECT * FROM sales_info;

9. a. Create the DEPT_NAMED_INDEX table based on the following table instance chart. Name the index for the PRIMARY KEY column as DEPT_PK_IDX.

<table>
<thead>
<tr>
<th>COLUMN</th>
<th>Name</th>
<th>Deptno</th>
<th>Dname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Key</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data type</td>
<td>Number</td>
<td></td>
<td>VARCHAR2</td>
</tr>
<tr>
<td>Length</td>
<td>4</td>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>

CREATE TABLE DEPT_NAMED_INDEX
(deptno NUMBER(4)
PRIMARY KEY USING INDEX
(CREATE INDEX dept_pk_idx ON
DEPT_NAMED_INDEX(deptno),
dname VARCHAR2(30));

b. Query the USER_INDEXES table to display the INDEX_NAME for the DEPT_NAMED_INDEX table.

   SELECT INDEX_NAME, TABLE_NAME
   FROM USER_INDEXES
   WHERE TABLE_NAME = 'DEPT_NAMED_INDEX';
Practice D Solutions

1. Write a script to describe and select the data from your tables. Use `CHR(10)` in the select list with the concatenation operator (||) to generate a line feed in your report. Save the output of the script into `my_file1.sql`. To save the file, select the SAVE option for the output, and execute the code. Remember to save the file with a .sql extension. To execute the `my_file1.sql`, browse to locate the script, load the script, and execute the script.

```sql
SET PAGESIZE 0

SELECT 'DESC ' || table_name || CHR(10) || 'SELECT * FROM ' || table_name || ' ;' FROM user_tables
/
SET PAGESIZE 24
SET LINESIZE 100
```

2. Use SQL to generate SQL statements that revoke user privileges. Use the data dictionary views `USER_TAB_PRIVS_MADE` and `USER_COL_PRIVS_MADE`.

a. Execute the script `\Labs\privs.sql` to grant privileges to the user SYSTEM.

b. Query the data dictionary views to check the privileges. In the sample output shown, note that the data in the GRANTOR column can vary depending on who the GRANTOR is. Also the last column that has been truncated is the GRANTABLE column.

```sql
COLUMN grantee FORMAT A10
COLUMN table_name FORMAT A10
COLUMN column_name FORMAT A10
COLUMN grantor FORMAT A10
COLUMN privilege FORMAT A10
SELECT * FROM user_tab_privs_made
WHERE grantee = 'SYSTEM';

SELECT * FROM user_col_privs_made
WHERE grantee = 'SYSTEM';
```
c. Produce a script to revoke the privileges. Save the output of the script into my_file2.sql. To save the file, select the SAVE option for the output, and execute the code. Remember to save the file with a .sql extension. To execute the my_file2.sql, browse to locate the script, load the script, and execute the script.

```
SET VERIFY OFF
SET PAGESIZE 0

SELECT    'REVOKE ' || privilege || ' ON ' ||
table_name || '  FROM system;' FROM  user_tab_privs_made
WHERE grantee = 'SYSTEM'
/
SELECT DISTINCT 'REVOKE ' || privilege || ' ON ' ||
table_name || '  FROM system;' FROM user_col_privs_made
WHERE grantee = 'SYSTEM'
/

SET VERIFY ON
SET PAGESIZE 24
```
Table Descriptions
and Data
**COUNTRIES Table**

DESCRIBE countries

<table>
<thead>
<tr>
<th>Name</th>
<th>Null?</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNTRY_ID</td>
<td>NOT NULL</td>
<td>CHAR(2)</td>
</tr>
<tr>
<td>COUNTRY_NAME</td>
<td></td>
<td>VARCHAR2(40)</td>
</tr>
<tr>
<td>REGION_ID</td>
<td></td>
<td>NUMBER</td>
</tr>
</tbody>
</table>

SELECT * FROM countries;

<table>
<thead>
<tr>
<th>CO</th>
<th>COUNTRY_NAME</th>
<th>REGION_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA</td>
<td>Canada</td>
<td>2</td>
</tr>
<tr>
<td>DE</td>
<td>Germany</td>
<td>1</td>
</tr>
<tr>
<td>UK</td>
<td>United Kingdom</td>
<td>1</td>
</tr>
<tr>
<td>US</td>
<td>United States of America</td>
<td>2</td>
</tr>
</tbody>
</table>
### DEPARTMENTS Table

#### DESCRIBE departments

<table>
<thead>
<tr>
<th>Name</th>
<th>Null?</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEPARTMENT_ID</td>
<td>NOT NULL</td>
<td>NUMBER(4)</td>
</tr>
<tr>
<td>DEPARTMENT_NAME</td>
<td>NOT NULL</td>
<td>VARCHAR2(30)</td>
</tr>
<tr>
<td>MANAGER_ID</td>
<td></td>
<td>NUMBER(6)</td>
</tr>
<tr>
<td>LOCATION_ID</td>
<td></td>
<td>NUMBER(4)</td>
</tr>
</tbody>
</table>

#### 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.
## EMPLOYEES Table

<table>
<thead>
<tr>
<th>Name</th>
<th>Null?</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPLOYEE_ID_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>

```sql
DESCRIBE employees
```

```sql
SELECT * FROM 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>S</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></td>
</tr>
<tr>
<td>101</td>
<td>Neena</td>
<td>Kochhar</td>
<td>NKOCCHAR</td>
<td>515.123.4568</td>
<td>21-SEP-98</td>
<td>AD_VP</td>
<td></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></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></td>
</tr>
<tr>
<td>104</td>
<td>Bruce</td>
<td>Ernst</td>
<td>BERNST</td>
<td>590.423.4568</td>
<td>21-MAY-91</td>
<td>IT_PROG</td>
<td></td>
</tr>
<tr>
<td>107</td>
<td>Diana</td>
<td>Lorentz</td>
<td>DLORENTZ</td>
<td>590.423.5667</td>
<td>07-FEB-99</td>
<td>IT_PROG</td>
<td></td>
</tr>
<tr>
<td>124</td>
<td>Kevin</td>
<td>Mourgos</td>
<td>KMOURGOS</td>
<td>590.123.5234</td>
<td>16-NOV-99</td>
<td>ST_MAN</td>
<td></td>
</tr>
<tr>
<td>141</td>
<td>Trenna</td>
<td>Rajs</td>
<td>TRAJRS</td>
<td>650.121.8009</td>
<td>17-OCT-95</td>
<td>ST_CLERK</td>
<td></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></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></td>
</tr>
<tr>
<td>144</td>
<td>Peter</td>
<td>Vargas</td>
<td>PVARGAS</td>
<td>650.121.2004</td>
<td>09-JUL-98</td>
<td>ST_CLERK</td>
<td></td>
</tr>
<tr>
<td>149</td>
<td>Eleni</td>
<td>Zlotkey</td>
<td>EZLOTKEY</td>
<td>011.44.1344.429018</td>
<td>29-JAN-00</td>
<td>SA_MAN</td>
<td></td>
</tr>
<tr>
<td>174</td>
<td>Ellen</td>
<td>Abel</td>
<td>EABEL</td>
<td>011.44.1644.429367</td>
<td>11-MAY-96</td>
<td>SA_REP</td>
<td></td>
</tr>
<tr>
<td>176</td>
<td>Jonathon</td>
<td>Taylor</td>
<td>JTAYLOR</td>
<td>011.44.1644.429265</td>
<td>24-MAR-98</td>
<td>SA_REP</td>
<td></td>
</tr>
<tr>
<td>178</td>
<td>Kimberely</td>
<td>Grant</td>
<td>KGRANT</td>
<td>011.44.1644.429263</td>
<td>24-MAY-99</td>
<td>SA_REP</td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>Jennifer</td>
<td>Whalen</td>
<td>JWHALEN</td>
<td>515.123.4444</td>
<td>17-SEP-87</td>
<td>AD_ASST</td>
<td></td>
</tr>
<tr>
<td>201</td>
<td>Michael</td>
<td>Hartstein</td>
<td>MHARTSTE</td>
<td>515.123.5555</td>
<td>17-FEB-96</td>
<td>MK_MAN</td>
<td></td>
</tr>
<tr>
<td>202</td>
<td>Pat</td>
<td>Fay</td>
<td>PFAY</td>
<td>603.123.6666</td>
<td>17-AUG-97</td>
<td>MK_REP</td>
<td></td>
</tr>
<tr>
<td>205</td>
<td>Shelley</td>
<td>Higgins</td>
<td>SHIGGINS</td>
<td>515.123.8080</td>
<td>07-JUN-94</td>
<td>AC_MGR</td>
<td></td>
</tr>
<tr>
<td>206</td>
<td>William</td>
<td>Gietz</td>
<td>WGIETZ</td>
<td>515.123.8181</td>
<td>07-JUN-94</td>
<td>AC_ACCOUNT</td>
<td></td>
</tr>
</tbody>
</table>

20 rows selected.
### EMPLOYEES Table (continued)

<table>
<thead>
<tr>
<th>SALARY</th>
<th>COMMISSION_PCT</th>
<th>MANAGER_ID</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>24000</td>
<td></td>
<td></td>
<td>90</td>
</tr>
<tr>
<td>17000</td>
<td></td>
<td>100</td>
<td>90</td>
</tr>
<tr>
<td>17000</td>
<td></td>
<td>100</td>
<td>90</td>
</tr>
<tr>
<td>9000</td>
<td></td>
<td>102</td>
<td>60</td>
</tr>
<tr>
<td>6000</td>
<td></td>
<td>103</td>
<td>60</td>
</tr>
<tr>
<td>4200</td>
<td></td>
<td>103</td>
<td>60</td>
</tr>
<tr>
<td>5800</td>
<td></td>
<td>100</td>
<td>50</td>
</tr>
<tr>
<td>3500</td>
<td></td>
<td>124</td>
<td>50</td>
</tr>
<tr>
<td>3100</td>
<td></td>
<td>124</td>
<td>50</td>
</tr>
<tr>
<td>2800</td>
<td></td>
<td>124</td>
<td>50</td>
</tr>
<tr>
<td>2500</td>
<td></td>
<td>124</td>
<td>50</td>
</tr>
<tr>
<td>10500</td>
<td>.2</td>
<td>100</td>
<td>80</td>
</tr>
<tr>
<td>11000</td>
<td>.3</td>
<td>149</td>
<td>80</td>
</tr>
<tr>
<td>8600</td>
<td>.2</td>
<td>149</td>
<td>80</td>
</tr>
<tr>
<td>7000</td>
<td>.15</td>
<td>149</td>
<td></td>
</tr>
<tr>
<td>4400</td>
<td></td>
<td>101</td>
<td>10</td>
</tr>
<tr>
<td>13000</td>
<td></td>
<td>100</td>
<td>20</td>
</tr>
<tr>
<td>6000</td>
<td></td>
<td>201</td>
<td>20</td>
</tr>
<tr>
<td>12000</td>
<td></td>
<td>101</td>
<td>110</td>
</tr>
<tr>
<td>T 8300</td>
<td></td>
<td>205</td>
<td>110</td>
</tr>
</tbody>
</table>
### JOBS Table

**DESCRIBE jobs**

<table>
<thead>
<tr>
<th>Name</th>
<th>Null?</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOB_ID</td>
<td>NOT NULL</td>
<td>VARCHAR2(10)</td>
</tr>
<tr>
<td>JOB_TITLE</td>
<td>NOT NULL</td>
<td>VARCHAR2(35)</td>
</tr>
<tr>
<td>MIN_SALARY</td>
<td></td>
<td>NUMBER(6)</td>
</tr>
<tr>
<td>MAX_SALARY</td>
<td></td>
<td>NUMBER(6)</td>
</tr>
</tbody>
</table>

**SELECT * FROM jobs;**

<table>
<thead>
<tr>
<th>JOB_ID</th>
<th>JOB_TITLE</th>
<th>MIN_SALARY</th>
<th>MAX_SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>AD_PRES</td>
<td>President</td>
<td>20000</td>
<td>40000</td>
</tr>
<tr>
<td>AD_VP</td>
<td>Administration Vice President</td>
<td>15000</td>
<td>30000</td>
</tr>
<tr>
<td>AD_ASST</td>
<td>Administration Assistant</td>
<td>3000</td>
<td>6000</td>
</tr>
<tr>
<td>AC_MGR</td>
<td>Accounting Manager</td>
<td>8200</td>
<td>16000</td>
</tr>
<tr>
<td>AC_ACCOUNT</td>
<td>Public Accountant</td>
<td>4200</td>
<td>9000</td>
</tr>
<tr>
<td>SA_MAN</td>
<td>Sales Manager</td>
<td>10000</td>
<td>20000</td>
</tr>
<tr>
<td>SA_REP</td>
<td>Sales Representative</td>
<td>5000</td>
<td>12000</td>
</tr>
<tr>
<td>ST_MAN</td>
<td>Stock Manager</td>
<td>5500</td>
<td>8500</td>
</tr>
<tr>
<td>ST_CLERK</td>
<td>Stock Clerk</td>
<td>2000</td>
<td>5000</td>
</tr>
<tr>
<td>IT_PROG</td>
<td>Programmer</td>
<td>4000</td>
<td>10000</td>
</tr>
<tr>
<td>MK_MAN</td>
<td>Marketing Manager</td>
<td>9000</td>
<td>15000</td>
</tr>
<tr>
<td>MK_REP</td>
<td>Marketing Representative</td>
<td>4000</td>
<td>9000</td>
</tr>
</tbody>
</table>

12 rows selected.
**JOB_GRADES Table**

```sql
DESCRIBE job_grades
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Null?</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRADE_LEVEL</td>
<td></td>
<td>VARCHAR2(3)</td>
</tr>
<tr>
<td>LOWEST_SAL</td>
<td></td>
<td>NUMBER</td>
</tr>
<tr>
<td>HIGHEST_SAL</td>
<td></td>
<td>NUMBER</td>
</tr>
</tbody>
</table>

```sql
SELECT * FROM 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>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>

6 rows selected.
JOB_HISTORY Table

DESCRIBE job_history

<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>START_DATE</td>
<td>NOT NULL</td>
<td>DATE</td>
</tr>
<tr>
<td>END_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>DEPARTMENT_ID</td>
<td></td>
<td>NUMBER(4)</td>
</tr>
</tbody>
</table>

SELECT * FROM job_history;

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>START_DATE</th>
<th>END_DATE</th>
<th>JOB_ID</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>102</td>
<td>13-JAN-93</td>
<td>24-JUL-98</td>
<td>IT_PROG</td>
<td>60</td>
</tr>
<tr>
<td>101</td>
<td>21-SEP-89</td>
<td>27-OCT-93</td>
<td>AC_ACCOUNT</td>
<td>110</td>
</tr>
<tr>
<td>101</td>
<td>26-OCT-93</td>
<td>15-MAR-97</td>
<td>AC_MGR</td>
<td>110</td>
</tr>
<tr>
<td>201</td>
<td>17-FEB-96</td>
<td>19-DEC-99</td>
<td>MK_REP</td>
<td>20</td>
</tr>
<tr>
<td>114</td>
<td>24-MAR-98</td>
<td>31-DEC-99</td>
<td>ST_CLERK</td>
<td>50</td>
</tr>
<tr>
<td>122</td>
<td>01-JAN-99</td>
<td>31-DEC-99</td>
<td>ST_CLERK</td>
<td>50</td>
</tr>
<tr>
<td>200</td>
<td>17-SEP-87</td>
<td>17-JUN-93</td>
<td>AD_ASST</td>
<td>90</td>
</tr>
<tr>
<td>176</td>
<td>24-MAR-98</td>
<td>31-DEC-98</td>
<td>SA_REP</td>
<td>80</td>
</tr>
<tr>
<td>176</td>
<td>01-JAN-99</td>
<td>31-DEC-99</td>
<td>SA_MAN</td>
<td>80</td>
</tr>
<tr>
<td>200</td>
<td>01-JUL-94</td>
<td>31-DEC-98</td>
<td>AC_ACCOUNT</td>
<td>90</td>
</tr>
</tbody>
</table>

10 rows selected.
### LOCATIONS Table

**DESCRIBE locations**

<table>
<thead>
<tr>
<th>Name</th>
<th>Null?</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCATION_ID</td>
<td>NOT NULL</td>
<td>NUMBER(4)</td>
</tr>
<tr>
<td>STREET_ADDRESS</td>
<td></td>
<td>VARCHAR2(40)</td>
</tr>
<tr>
<td>POSTAL_CODE</td>
<td></td>
<td>VARCHAR2(12)</td>
</tr>
<tr>
<td>CITY</td>
<td>NOT NULL</td>
<td>VARCHAR2(30)</td>
</tr>
<tr>
<td>STATE_PROVINCE</td>
<td></td>
<td>VARCHAR2(25)</td>
</tr>
<tr>
<td>COUNTRY_ID</td>
<td></td>
<td>CHAR(2)</td>
</tr>
</tbody>
</table>

**SELECT * FROM locations;**

<table>
<thead>
<tr>
<th>LOCATION_ID</th>
<th>STREET_ADDRESS</th>
<th>POSTAL_CODE</th>
<th>CITY</th>
<th>STATE_PROVINCE</th>
<th>CO</th>
</tr>
</thead>
<tbody>
<tr>
<td>1400</td>
<td>2014 Jabberwocky Rd</td>
<td>26192</td>
<td>Southlake</td>
<td>Texas</td>
<td>US</td>
</tr>
<tr>
<td>1500</td>
<td>2011 Interiors Blvd</td>
<td>99236</td>
<td>South San Francisco</td>
<td>California</td>
<td>US</td>
</tr>
<tr>
<td>1700</td>
<td>2004 Charade Rd</td>
<td>98199</td>
<td>Seattle</td>
<td>Washington</td>
<td>US</td>
</tr>
<tr>
<td>1800</td>
<td>460 Bloor St. W.</td>
<td>ON M5S 1XB</td>
<td>Toronto</td>
<td>Ontario</td>
<td>CA</td>
</tr>
<tr>
<td>2500</td>
<td>Magdalen Centre, The Oxford Science Park</td>
<td>OX9 9ZB</td>
<td>Oxford</td>
<td>Oxford</td>
<td>UK</td>
</tr>
</tbody>
</table>
**REGIONS Table**

**DESCRIBE regions**

<table>
<thead>
<tr>
<th>Name</th>
<th>Null?</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>REGION_ID</td>
<td>NOT NULL</td>
<td>NUMBER</td>
</tr>
<tr>
<td>REGION_NAME</td>
<td></td>
<td>VARCHAR2(25)</td>
</tr>
</tbody>
</table>

**SELECT * FROM regions;**

<table>
<thead>
<tr>
<th>REGION_ID</th>
<th>REGION_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Europe</td>
</tr>
<tr>
<td>2</td>
<td>Americas</td>
</tr>
<tr>
<td>3</td>
<td>Asia</td>
</tr>
<tr>
<td>4</td>
<td>Middle East and Africa</td>
</tr>
</tbody>
</table>
Objectives

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

• Log in to SQL*Plus
• Edit SQL commands
• Format output using SQL*Plus commands
• Interact with script files

Lesson Aim

You may want to create SELECT statements that can be used again and again. This lesson also covers the use of SQL*Plus commands to execute SQL statements. You learn how to format output using SQL*Plus commands, edit SQL commands, and save scripts in SQL*Plus.
SQL and SQL*Plus

SQL is a command language for communication with the Oracle9i Server from any tool or application. Oracle SQL contains many extensions. When you enter a SQL statement, it is stored in a part of memory called the SQL buffer and remains there until you enter a new SQL statement.

SQL*Plus is an Oracle tool that recognizes and submits SQL statements to the Oracle9i Server for execution. It contains its own command language.

Features of SQL

- SQL can be used by a range of users, including those with little or no programming experience.
- It is a nonprocedural language.
- It reduces the amount of time required for creating and maintaining systems.
- It is an English-like language.

Features of SQL*Plus

- SQL*Plus accepts ad hoc entry of statements.
- It accepts SQL input from files.
- It provides a line editor for modifying SQL statements.
- It controls environmental settings.
- It formats query results into basic reports.
- It accesses local and remote databases.
### SQL Statements versus SQL*Plus Commands

<table>
<thead>
<tr>
<th>SQL</th>
<th>SQL*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>Keywords cannot be</td>
<td>Keywords can be abbreviated</td>
</tr>
<tr>
<td>abbreviated</td>
<td></td>
</tr>
<tr>
<td>Statements manipulate</td>
<td>Commands do not allow manipulation of values in the database data and table definitions in the database</td>
</tr>
</tbody>
</table>

#### SQL and SQL*Plus (continued)

The following table compares SQL and SQL*Plus:

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

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

SQL*Plus

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

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

<table>
<thead>
<tr>
<th>Category</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Environment</td>
<td>Affect the general behavior of SQL statements for the session</td>
</tr>
<tr>
<td>Format</td>
<td>Format query results</td>
</tr>
<tr>
<td>File manipulation</td>
<td>Save, load, and run script files</td>
</tr>
<tr>
<td>Execution</td>
<td>Send SQL statements from SQL buffer to the Oracle Server</td>
</tr>
<tr>
<td>Edit</td>
<td>Modify SQL statements in the buffer</td>
</tr>
<tr>
<td>Interaction</td>
<td>Create and pass variables to SQL statements, print variable values, and print messages to the screen</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>Connect to the database, manipulate the SQL*Plus environment, and display column definitions</td>
</tr>
</tbody>
</table>
Logging In to SQL*Plus

- From a Windows environment:

```
User Name: scott
Password: ****
Host String: 
```

- From a command line:

```
sqlplus [username[/password[@database]]]
```

Logging In to SQL*Plus

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

To log in through a Windows environment:
1. Select Start > Programs > Oracle for Windows NT > SQL*Plus.
2. Enter the username, password, and database name.

To log in through a command line environment:
1. Log on to your machine.
2. Enter the SQL*Plus command shown in the slide.

In the syntax:
```
username your database username.
password your database password (if you enter your password here, it is visible.)
@databases the database connect string.
```

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

After you log in to SQL*Plus, you see the following message (if you are using SQL*Plus version 9i):

```
SQL*Plus: Release 9.0.1.0.0 - Development on Tue Jan 9 08:44:28 2001
(c) Copyright 2000 Oracle Corporation. All rights reserved.
```
Displaying Table Structure

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

DESC[RI]BE tablename

Displaying Table Structure

In SQL*Plus you can display the structure of a table using the DESCRIBE command. The result of the command is a display of column names and data types as well as an indication if a column must contain data.

In the syntax:

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

To describe the JOB_GRADES table, use this command:

    SQL> DESCRIBE job_grades

<table>
<thead>
<tr>
<th>Name</th>
<th>Null?</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRADE_LEVEL</td>
<td></td>
<td>VARCHAR2 (3)</td>
</tr>
<tr>
<td>LOWEST_SAL</td>
<td></td>
<td>NUMBER</td>
</tr>
<tr>
<td>HIGHEST_SAL</td>
<td></td>
<td>NUMBER</td>
</tr>
</tbody>
</table>
Displaying Table Structure

SQL> DESCRIBE departments

<table>
<thead>
<tr>
<th>Name</th>
<th>Null?</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEPARTMENT_ID</td>
<td>NOT NULL</td>
<td>NUMBER(4)</td>
</tr>
<tr>
<td>DEPARTMENT_NAME</td>
<td>NOT NULL</td>
<td>VARCHAR2(30)</td>
</tr>
<tr>
<td>MANAGER_ID</td>
<td></td>
<td>NUMBER(6)</td>
</tr>
<tr>
<td>LOCATION_ID</td>
<td></td>
<td>NUMBER(4)</td>
</tr>
</tbody>
</table>

Displaying Table Structure (continued)

The example in the slide displays the information about the structure of the DEPARTMENTS table. In the result:

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

Type displays the data type for a column.

The following table describes the data types:

<table>
<thead>
<tr>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMBER(p,s)</td>
<td>Number value that has a maximum number of digits p, the number of digits to the right of the decimal point s</td>
</tr>
<tr>
<td>VARCHAR2(s)</td>
<td>Variable-length character value of maximum size s</td>
</tr>
<tr>
<td>DATE</td>
<td>Date and time value between January 1, 4712 B.C., and A.D. December 31, 9999</td>
</tr>
<tr>
<td>CHAR(s)</td>
<td>Fixed-length character value of size s</td>
</tr>
</tbody>
</table>
SQL*Plus Editing Commands

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

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[PPEND] text</td>
<td>Adds text to the end of the current line</td>
</tr>
<tr>
<td>C[HANGE] / old / new</td>
<td>Changes old text to new in the current line</td>
</tr>
<tr>
<td>C[HANGE] / text /</td>
<td>Deletes text from the current line</td>
</tr>
<tr>
<td>CL[EAR] BUFF[ER]</td>
<td>Deletes all lines from the SQL buffer</td>
</tr>
<tr>
<td>DEL</td>
<td>Deletes current line</td>
</tr>
<tr>
<td>DEL n</td>
<td>Deletes line n</td>
</tr>
<tr>
<td>DEL m n</td>
<td>Deletes lines m to n inclusive</td>
</tr>
</tbody>
</table>

Guidelines

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

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

**Note:** You can enter only one SQL*Plus command per SQL prompt. SQL*Plus commands are not stored in the buffer. To continue a SQL*Plus command on the next line, end the first line with a hyphen (−).
Using **LIST**, **n**, and **APPEND**

- Use the `L[IST]` command to display the contents of the SQL buffer. The * beside line 2 in the buffer indicates that line 2 is the current line. Any edits that you made apply to the current line.
- Change the number of the current line by entering the number of the line you want to edit. The new current line is displayed.
- Use the `A[PPEND]` command to add text to the current line. The newly edited line is displayed. Verify the new contents of the buffer by using the `LIST` command.

**Note:** Many SQL*Plus commands including **LIST** and **APPEND** can be abbreviated to just their first letter. **LIST** can be abbreviated to **L**, **APPEND** can be abbreviated to **A**.

```sql
SQL> LIST
1  SELECT last_name
2  FROM employees

SQL> 1
1* SELECT last_name

SQL> A , job_id
1* SELECT last_name, job_id

SQL> L
1  SELECT last_name, job_id
2  FROM employees
```
Using the CHANGE Command

- Use L[IST] to display the contents of the buffer
- Use the C[HANGE] command to alter the contents of the current line in the SQL buffer. In this case, replace the employees table with the departments table. The new current line is displayed.
- Use the L[IST] command to verify the new contents of the buffer.
SQL*Plus File Commands

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

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAVE filename [.ext] [REP[ACE]APP[END]]</td>
<td>Saves current contents of SQL buffer to a file. Use APPEND to add to an existing file; use REPLACE to overwrite an existing file. The default extension is .sql.</td>
</tr>
<tr>
<td>GET filename [.ext]</td>
<td>Writes the contents of a previously saved file to the SQL buffer. The default extension for the filename is .sql.</td>
</tr>
<tr>
<td>START filename [.ext]</td>
<td>Runs a previously saved command file.</td>
</tr>
<tr>
<td>@ filename</td>
<td>Runs a previously saved command file (same as START).</td>
</tr>
<tr>
<td>ED[IT]</td>
<td>Invokes the editor and saves the buffer contents to a file named afiedt.buf.</td>
</tr>
<tr>
<td>ED[IT] [filename[.ext]]</td>
<td>Invokes the editor to edit contents of a saved file.</td>
</tr>
<tr>
<td>SPOOL [filename</td>
<td>.ext]</td>
</tr>
<tr>
<td>EXIT</td>
<td>Leaves SQL*Plus.</td>
</tr>
</tbody>
</table>
Using the \texttt{SAVE} and \texttt{START} Commands

\begin{verbatim}
SQL> L
  1  SELECT last_name, manager_id, department_id
  2* FROM employees
SQL> SAVE my_query

Created file my_query

SQL> START my_query

LAST_NAME       MANAGER_ID DEPARTMENT_ID
------------------------- ---------- -------------
                 King                        90
                 Kochhar                    100  90
... 20 rows selected.
\end{verbatim}

\textbf{SAVE}

Use the \texttt{SAVE} command to store the current contents of the buffer in a file. In this way, you can store frequently used scripts for use in the future.

\textbf{START}

Use the \texttt{START} command to run a script in SQL*Plus.

\textbf{EDIT}

Use the \texttt{EDIT} command to edit an existing script. This opens an editor with the script file in it. When you have made the changes, exit the editor to return to the SQL*Plus command line.
Summary

Use SQL*Plus as an environment to:
• Execute SQL statements
• Edit SQL statements
• Format output
• Interact with script files

Summary

SQL*Plus is an execution environment that you can use to send SQL commands to the database server and to edit and save SQL commands. You can execute commands from the SQL prompt or from a script file.
Writing Advanced Scripts
Objectives

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

- Describe the types of problems that are solved by using SQL to generate SQL
- Write a script that generates a script of `DROP TABLE` statements
- Write a script that generates a script of `INSERT INTO` statements

Lesson Aim

In this appendix, you learn how to write a SQL script to generates a SQL script.
Using SQL to Generate SQL

SQL can be used to generate scripts in SQL.

The data dictionary

- Is a collection of tables and views that contain database information
- Is created and maintained by the Oracle server

The examples used in this lesson involve selecting information from the data dictionary. The data dictionary is a collection of tables and views that contain information about the database. This collection is created and maintained by the Oracle Server. All data dictionary tables are owned by the SYS user. Information stored in the data dictionary includes names of the Oracle Server users, privileges granted to users, database object names, table constraints, and audition information. There are four categories of data dictionary views. Each category has a distinct prefix that reflects its intended use.

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USER_</td>
<td>Contains details of objects owned by the user</td>
</tr>
<tr>
<td>ALL_</td>
<td>Contains details of objects to which the user has been granted access rights, in addition to objects owned by the user</td>
</tr>
<tr>
<td>DBA_</td>
<td>Contains details of users with DBA privileges to access any object in the database</td>
</tr>
<tr>
<td>V$___</td>
<td>Stored information about database server performance and locking; available only to the DBA</td>
</tr>
</tbody>
</table>
Creating a Basic Script

The example in the slide produces a report with `CREATE TABLE` statements from every table you own. Each `CREATE TABLE` statement produced in the report includes the syntax to create a table using the table name with a suffix of `_test` and having only the structure of the corresponding existing table. The old table name is obtained from the `TABLE_NAME` column of the data dictionary view `USER_TABLES`.

The next step is to enhance the report to automate the process.

**Note:** You can query the data dictionary tables to view various database objects that you own. The data dictionary views frequently used include:

- **USER_TABLES:** Displays description of the user’s own tables
- **USER_OBJECTS:** Displays all the objects owned by the user
- **USER_TAB_PRIVS_MADE:** Displays all grants on objects owned by the user
- **USER_COL_PRIVS_MADE:** Displays all grants on columns of objects owned by the user
Controlling the Environment

In order to execute the SQL statements that are generated, you must capture them in a spool file that can then be run. You must also plan to clean up the output that is generated and make sure that you suppress elements such as headings, feedback messages, top titles, and so on. You can accomplish all of this by using `SQL*Plus` commands.

```
SET ECHO OFF
SET FEEDBACK OFF
SET PAGESIZE 0

SPOOL dropem.sql

SQL STATEMENT
SPOOL OFF
SET FEEDBACK ON
SET PAGESIZE 24
SET ECHO ON

Set system variables to appropriate values.

Set system variables back to the default value.
```
The output of the command on the slide is saved into a file called `dropem.sql` using the Save Output option in `iSQL*Plus`. This file contains the following data. This file can now be started from the `iSQL*Plus` by locating the script file, loading it, and executing it.

```
SET ECHO OFF
SET FEEDBACK OFF
SET PAGESIZE 0

SELECT 'DROP TABLE ' || object_name || ';'
FROM user_objects
WHERE object_type = 'TABLE'
/

SET FEEDBACK ON
SET PAGESIZE 24
SET ECHO ON
```

**Note:** By default, files are spooled into the `ORACLE_HOME\ORANT\BIN` folder in Windows NT.
Dumping Table Contents to a File

Sometimes it is useful to have the values for the rows of a table in a text file in the format of an INSERT INTO VALUES statement. This script can be run to populate the table, in case the table has been dropped accidentally.

The example in the slide produces INSERT statements for the DEPARTMENTS_TEST table, captured in the data.sql file using the Save Output option in iSQL*Plus.

The contents of the data.sql script file are as follows:

```sql
INSERT INTO departments_test VALUES (10, 'Administration', 1700);
INSERT INTO departments_test VALUES (20, 'Marketing', 1800);
INSERT INTO departments_test VALUES (50, 'Shipping', 1500);
INSERT INTO departments_test VALUES (60, 'IT', 1400);
...```
Dumping Table Contents to a File (continued)

You may have noticed the large number of single quotes in the slide on the previous page. A set of four single quotes produces one single quote in the final statement. Also remember that character and date values must be surrounded by quotes.

Within a string, to display one single quote, you need to prefix it with another single quote. For example, in the fifth example in the slide, the surrounding quotes are for the entire string. The second quote acts as a prefix to display the third quote. Thus the result is one single quote followed by the parenthesis followed by the semicolon.
Generating a Dynamic Predicate

The example in the slide generates a SELECT statement that retrieves data of all employees in a department who were hired on a specific day. The script generates the WHERE clause dynamically.

Note: Once the user variable is in place, you need to use the UNDEFINE command to delete it.

The first SELECT statement prompts you to enter the department number. If you do not enter any department number, the department number is treated as null by the DECODE function, and the user is then prompted for the hire date. If you do not enter any hire date, the hire date is treated as null by the DECODE function and the dynamic WHERE clause that is generated is also a null, which causes the second SELECT statement to retrieve all rows from the EMPLOYEES table.

Note: The NEW_VALUE variable specifies a variable to hold a column value. You can reference the variable in TTITLE commands. Use NEW_VALUE to display column values or the date in the top title. You must include the column in a BREAK command with the SKIP PAGE action. The variable name cannot contain a pound sign (#). NEW_VALUE is useful for master/detail reports in which there is a new master record for each page.
Generating a Dynamic Predicate (continued)

Note: Here, the hire date must be entered in **DD-MON-YYYY** format.

The `SELECT` statement in the previous slide can be interpreted as follows:

```sql
IF (<<deptno>> is not entered) THEN
  IF (<<hiredate>> is not entered) THEN
    return empty string
  ELSE
    return the string ‘WHERE hire_date = TO_DATE(<<hiredate>>', 'DD-MON-YYYY')’
  END IF
ELSE
  IF (<<hiredate>> is not entered) THEN
    return the string ‘WHERE department_id = <<deptno>> entered’
  ELSE
    return the string ‘WHERE department_id = <<deptno>> entered
                   AND hire_date = TO_DATE(<<hiredate>>', 'DD-MON-YYYY')’
  END IF
END IF
```

The returned string becomes the value of the variable `DYN_WHERE_CLAUSE`, that will be used in the second `SELECT` statement.
Summary

In this appendix, you should have learned the following:

- You can write a SQL script to generate another SQL script.
- Script files often use the data dictionary.
- You can capture the output in a file.

SQL can be used to generate SQL scripts. These scripts can be used to avoid repetitive coding, drop or re-create objects, get help from the data dictionary, and generate dynamic predicates that contain run-time parameters.

*SQL*Plus commands can be used to capture the reports generated by the SQL statements and clean up the output that is generated, such as suppressing headings, feedback messages, and so on.
Practice D Overview

This practice covers the following topics:

• Writing a script to describe and select the data from your tables
• Writing a script to revoke user privileges
Practice D

1. Write a script to describe and select the data from your tables. Use `CHR(10)` in the select list with the concatenation operator (||) to generate a line feed in your report. Save the output of the script into `my_file1.sql`. To save the file, select SAVE option for the output and execute the code. Remember to save the file with a .sql extension. To execute the `my_file1.sql`, browse to locate the script, load the script, and execute the script.

2. Use SQL to generate SQL statements that revoke user privileges. Use the data dictionary views `USER_TAB_PRIVS_MADE` and `USER_COL_PRIVS_MADE`.
   a. Execute the script `\Lab\privs.sql` to grant privileges to the user `SYSTEM`.
   b. Query the data dictionary views to check the privileges. In the sample output shown, note that the data in the `GRANTOR` column can vary depending on who the `GRANTOR` is. Also the last column that has been truncated is the `GRANTABLE` column.

<table>
<thead>
<tr>
<th>GRANTEE</th>
<th>TABLE_NAME</th>
<th>GRANTOR</th>
<th>PRIVILEGE</th>
<th>GRA</th>
<th>HIE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSTEM</td>
<td>DEPARTMENT S</td>
<td>TRNG4</td>
<td>ALTER</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>SYSTEM</td>
<td>DEPARTMENT S</td>
<td>TRNG4</td>
<td>DELETE</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>SYSTEM</td>
<td>DEPARTMENT S</td>
<td>TRNG4</td>
<td>INDEX</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>SYSTEM</td>
<td>DEPARTMENT S</td>
<td>TRNG4</td>
<td>INSERT</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>SYSTEM</td>
<td>DEPARTMENT S</td>
<td>TRNG4</td>
<td>SELECT</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>SYSTEM</td>
<td>DEPARTMENT S</td>
<td>TRNG4</td>
<td>UPDATE</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>SYSTEM</td>
<td>DEPARTMENT S</td>
<td>TRNG4</td>
<td>REFERENCES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>SYSTEM</td>
<td>DEPARTMENT S</td>
<td>TRNG4</td>
<td>ON COMMIT REFRESH</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>SYSTEM</td>
<td>DEPARTMENT S</td>
<td>TRNG4</td>
<td>QUERY WRITE</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>SYSTEM</td>
<td>DEPARTMENT S</td>
<td>TRNG4</td>
<td>DEBUG</td>
<td>NO</td>
<td>NO</td>
</tr>
</tbody>
</table>

10 rows selected.

<table>
<thead>
<tr>
<th>GRANTEE</th>
<th>TABLE_NAME</th>
<th>COLUMN_NAME</th>
<th>GRANTOR</th>
<th>PRIVILEGE</th>
<th>GRA</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSTEM</td>
<td>EMPLOYEES</td>
<td>JOB_ID</td>
<td>TRNG4</td>
<td>UPDATE</td>
<td>NO</td>
</tr>
<tr>
<td>SYSTEM</td>
<td>EMPLOYEES</td>
<td>SALARY</td>
<td>TRNG4</td>
<td>UPDATE</td>
<td>NO</td>
</tr>
</tbody>
</table>

2 rows selected.
c. Produce a script to revoke the privileges. Save the output of the script into my_file2.sql. To save the file, select the SAVE option for the output, and execute the code. Remember to save the file with a .sql extension. To execute the my_file2.sql, browse to locate the script, load the script, and execute the script.

| 'REVOKE'|PRIVILEGE||'ON'||TABLE_NAME||'FROM'|SYSTEM; |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| REVOKE ALTER ON DEPARTMENTS FROM system; |
| REVOKE DELETE ON DEPARTMENTS FROM system; |
| REVOKE INDEX ON DEPARTMENTS FROM system; |
| REVOKE INSERT ON DEPARTMENTS FROM system; |
| REVOKE SELECT ON DEPARTMENTS FROM system; |
| REVOKE UPDATE ON DEPARTMENTS FROM system; |
| REVOKE REFERENCES ON DEPARTMENTS FROM system; |
| REVOKE ON COMMIT REFRESH ON DEPARTMENTS FROM system; |
| REVOKE QUERY REWRITE ON DEPARTMENTS FROM system; |
| REVOKE DEBUG ON DEPARTMENTS FROM system; |

| 'REVOKE'|PRIVILEGE||'ON'||TABLE_NAME||'FROM'|SYSTEM; |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| REVOKE UPDATE ON EMPLOYEES FROM system; |
Objectives

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

• Describe the Oracle Server architecture and its main components
• List the structures involved in connecting a user to an Oracle instance
• List the stages in processing:
  – Queries
  – DML statements
  – Commits

Objectives

This appendix introduces Oracle Server architecture by describing the files, processes, and memory structures involved in establishing a database connection and executing a SQL command.
Overview

The Oracle Server is an object relational database management system that provides an open, comprehensive, integrated approach to information management.

Primary Components

There are several processes, memory structures, and files in an Oracle Server; however, not all of them are used when processing a SQL statement. Some are used to improve the performance of the database, ensure that the database can be recovered in the event of a software or hardware error, or perform other tasks necessary to maintain the database. The Oracle Server consists of an Oracle instance and an Oracle database.

Oracle Instance

An Oracle instance is the combination of the background processes and memory structures. The instance must be started to access the data in the database. Every time an instance is started, a system global area (SGA) is allocated and Oracle background processes are started. The SGA is a memory area used to store database information that is shared by database processes.
Primary Components (continued)

Oracle Instance (continued)
Background processes perform functions on behalf of the invoking process. They consolidate functions that would otherwise be handled by multiple Oracle programs running for each user. The background processes perform I/O and monitor other Oracle processes to provide increased parallelism for better performance and reliability.

Other Processes
The user process is the application program that originates SQL statements. The server process executes the SQL statements sent from the user process.

Database Files
Database files are operating system files that provide the actual physical storage for database information. The database files are used to ensure that the data is kept consistent and can be recovered in the event of a failure of the instance.

Other Files
Nondatabase files are used to configure the instance, authenticate privileged users, and recover the database in the event of a disk failure.

SQL Statement Processing
The user and server processes are the primary processes involved when a SQL statement is executed; however, other processes may help the server complete the processing of the SQL statement.

Oracle Database Administrators
Database administrators are responsible for maintaining the Oracle Server so that the server can process user requests. An understanding of the Oracle architecture is necessary to maintain it effectively.
An Oracle database is a collection of data that is treated as a unit. The general purpose of a database is to store and retrieve related information. The database has a logical structure and a physical structure. The physical structure of the database is the set of operating system files in the database. An Oracle database consists of three file types:

- **Data files** contain the actual data in the database. The data is stored in user-defined tables, but data files also contain the data dictionary, before-images of modified data, indexes, and other types of structures. A database has at least one data file. The characteristics of data files are:
  - A data file can be associated with only one database. Data files can have certain characteristics set so they can automatically extend when the database runs out of space. One or more data files form a logical unit of database storage called a tablespace. Redo logs contain a record of changes made to the database to enable recovery of the data in case of failures. A database requires at least two redo log files.
  - Control files contain information necessary to maintain and verify database integrity. For example, a control file is used to identify the data files and redo log files. A database needs at least one control file.
Other Key Physical Structures

The Oracle Server also uses other files that are not part of the database:

- The parameter file defines the characteristics of an Oracle instance. For example, it contains parameters that size some of the memory structures in the SGA.
- The password file authenticates which users are permitted to start up and shut down an Oracle instance.
- Archived redo log files are offline copies of the redo log files that may be necessary to recover from media failures.
Oracle Instance

An Oracle instance:
- Is a means to access an Oracle database
- Always opens one and only one database

Oracle Instance consists of the SGA memory structure and the background processes used to manage a database. An instance is identified by using methods specific to each operating system. The instance can open and use only one database at a time.

System Global Area
The SGA is a memory area used to store database information that is shared by database processes. It contains data and control information for the Oracle Server. It is allocated in the virtual memory of the computer where the Oracle server resides. The SGA consists of several memory structures:
- The shared pool is used to store the most recently executed SQL statements and the most recently used data from the data dictionary. These SQL statements may be submitted by a user process or, in the case of stored procedures, read from the data dictionary.
- The database buffer cache is used to store the most recently used data. The data is read from, and written to, the data files.
- The redo log buffer is used to track changes made to the database by the server and background processes.
System Global Area (continued)

The purpose of these structures is discussed in detail in later sections of this lesson.

There are also two optional memory structures in the SGA:

- Java pool: Used to store Java code
- Large pool: Used to store large memory structures not directly related to SQL statement processing; for example, data blocks copied during backup and restore operations

Background Processes

The background processes in an instance perform common functions that are needed to service requests from concurrent users without compromising the integrity and performance of the system. They consolidate functions that would otherwise be handled by multiple Oracle programs running for each user. The background processes perform I/O and monitor other Oracle processes to provide increased parallelism for better performance and reliability.

Depending on its configuration, an Oracle instance may include several background processes, but every instance includes these five required background processes:

- Database Writer (DBW0) is responsible for writing changed data from the database buffer cache to the data files.
- Log Writer (LGWR) writes changes registered in the redo log buffer to the redo log files.
- System Monitor (SMON) checks for consistency of the database and, if necessary, initiates recovery of the database when the database is opened.
- Process Monitor (PMON) cleans up resources if one of the Oracle processes fails.
- The Checkpoint Process (CKPT) is responsible for updating database status information in the control files and data files whenever changes in the buffer cache are permanently recorded in the database.

The following sections of this lesson explain how a server process uses some of the components of the Oracle instance and database to process SQL statements submitted by a user process.
Processing a SQL Statement

- Connect to an instance using:
  - The user process
  - The server process
- The Oracle Server components that are used depend on the type of SQL statement:
  - Queries return rows
  - DML statements log changes
  - Commit ensures transaction recovery
- Some Oracle Server components do not participate in SQL statement processing.

Components Used to Process SQL

Not all of the components of an Oracle instance are used to process SQL statements. The user and server processes are used to connect a user to an Oracle instance. These processes are not part of the Oracle instance, but are required to process a SQL statement.

Some of the background processes, SGA structures, and database files are used to process SQL statements. Depending on the type of SQL statement, different components are used:

- Queries require additional processing to return rows to the user.
- Data manipulation language (DML) statements require additional processing to log the changes made to the data.
- Commit processing ensures that the modified data in a transaction can be recovered.

Some required background processes do not directly participate in processing a SQL statement but are used to improve performance and to recover the database.

The optional background process, ARC0, is used to ensure that a production database can be recovered.
Connecting to an Instance

Processes Used to Connect to an Instance
Before users can submit SQL statements to the Oracle Server, they must connect to an instance. The user starts a tool such as iSQL*Plus or runs an application developed using a tool such as Oracle Forms. This application or tool is executed in a user process.

In the most basic configuration, when a user logs on to the Oracle Server, a process is created on the computer running the Oracle server. This process is called a server process. The server process communicates with the Oracle instance on behalf of the user process that runs on the client. The server process executes SQL statements on behalf of the user.

Connection
A connection is a communication pathway between a user process and an Oracle Server. A database user can connect to an Oracle Server in one of three ways:

- The user logs on to the operating system running the Oracle instance and starts an application or tool that accesses the database on that system. The communication pathway is established using the interprocess communication mechanisms available on the host operating system.
Connection (continued)

- The user starts the application or tool on a local computer and connects over a network to the computer running the Oracle instance. In this configuration, called client-server, network software is used to communicate between the user and the Oracle Server.

- In a three-tiered connection, the user’s computer communicates over the network to an application or a network server, which is connected through a network to the machine running the Oracle instance. For example, the user runs a browser on a network computer to use an application residing on an NT server that retrieves data from an Oracle database running on a UNIX host.

Sessions

A session is a specific connection of a user to an Oracle Server. The session starts when the user is validated by the Oracle Server, and it ends when the user logs out or when there is an abnormal termination. For a given database user, many concurrent sessions are possible if the user logs on from many tools, applications, or terminals at the same time. Except for some specialized database administration tools, starting a database session requires that the Oracle Server be available for use.

Note: The type of connection explained here, where there is a one-to-one correspondence between a user and server process, is called a dedicated server connection.
Processing a Query

• Parse:
  – Search for identical statement
  – Check syntax, object names, and privileges
  – Lock objects used during parse
  – Create and store execution plan
• Execute: Identify rows selected
• Fetch: Return rows to user process

Query Processing Steps

Queries are different from other types of SQL statements because, if successful, they return data as results. Whereas other statements simply return success or failure, a query can return one row or thousands of rows.

There are three main stages in the processing of a query:

• Parse
• Execute
• Fetch

Parsing a SQL Statement

During the parse stage, the SQL statement is passed from the user process to the server process, and a parsed representation of the SQL statement is loaded into a shared SQL area.

During the parse, the server process performs the following functions:

• Searches for an existing copy of the SQL statement in the shared pool
• Validates the SQL statement by checking its syntax
• Performs data dictionary lookups to validate table and column definitions
The Shared Pool Components

During the parse stage, the server process uses the area in the SGA known as the shared pool to compile the SQL statement. The shared pool has two primary components:

- Library cache
- Data dictionary cache

Library Cache

The library cache stores information about the most recently used SQL statements in a memory structure called a shared SQL area. The shared SQL area contains:

- The text of the SQL statement
- The parse tree: A compiled version of the statement
- The execution plan: The steps to be taken when executing the statement

The optimizer is the function in the Oracle Server that determines the optimal execution plan.
Shared Pool Components (continued)

Library Cache (continued)

If a SQL statement is reexecuted and a shared SQL area already contains the execution plan for the statement, the server process does not need to parse the statement. The library cache improves the performance of applications that reuse SQL statements by reducing parse time and memory requirements. If the SQL statement is not reused, it is eventually aged out of the library cache.

Data Dictionary Cache

The data dictionary cache, also known as the dictionary cache or row cache, is a collection of the most recently used definitions in the database. It includes information about database files, tables, indexes, columns, users, privileges, and other database objects.

During the parse phase, the server process looks for the information in the dictionary cache to resolve the object names specified in the SQL statement and to validate the access privileges. If necessary, the server process initiates the loading of this information from the data files.

Sizing the Shared Pool

The size of the shared pool is specified by the initialization parameter SHARED_POOL_SIZE.
Database Buffer Cache

- Stores the most recently used blocks
- Size of a buffer based on `DB_BLOCK_SIZE`
- Number of buffers defined by `DB_BLOCK BUFFERS`

Function of the Database Buffer Cache

When a query is processed, the server process looks in the database buffer cache for any blocks it needs. If the block is not found in the database buffer cache, the server process reads the block from the data file and places a copy in the buffer cache. Because subsequent requests for the same block may find the block in memory, the requests may not require physical reads. The Oracle Server uses a least recently used algorithm to age out buffers that have not been accessed recently to make room for new blocks in the buffer cache.

Sizing the Database Buffer Cache

The size of each buffer in the buffer cache is equal to the size of an Oracle block, and it is specified by the `DB_BLOCK_SIZE` parameter. The number of buffers is equal to the value of the `DB_BLOCK BUFFERS` parameter.
Program Global Area (PGA)

- Not shared
- Writable only by the server process
- Contains:
  - Sort area
  - Session information
  - Cursor state
  - Stack space

Program Global Area Components

A program global area (PGA) is a memory region that contains data and control information for a server process. It is a nonshared memory created by Oracle when a server process is started. Access to it is exclusive to that server process and is read and written only by the Oracle Server code acting on behalf of it. The PGA memory allocated by each server process attached to an Oracle instance is referred to as the aggregated PGA memory allocated by the instance.

In a dedicated server configuration, the PGA of the server includes these components:

- Sort area: Used for any sorts that may be required to process the SQL statement
- Session information: Includes user privileges and performance statistics for the session
- Cursor state: Indicates the stage in the processing of the SQL statements that are currently used by the session
- Stack space: Contains other session variables

The PGA is allocated when a process is created and deallocated when the process is terminated.
DML Processing Steps

A data manipulation language (DML) statement requires only two phases of processing:

- Parse is the same as the parse phase used for processing a query
- Execute requires additional processing to make data changes

DML Execute Phase

To execute a DML statement:

- If the data and rollback blocks are not already in the buffer cache, the server process reads them from the data files into the buffer cache.
- The server process places locks on the rows that are to be modified.
- In the redo log buffer, the server process records the changes to be made to the rollback and data.
- The rollback block changes record the values of the data before it is modified. The rollback block is used to store the before image of the data, so that the DML statements can be rolled back if necessary.
- The data blocks changes record the new values of the data.
DML Processing Steps (continued)

DML Execute Phase (continued)

The server process records the before image to the rollback block and updates the data block. Both of these changes are done in the database buffer cache. Any changed blocks in the buffer cache are marked as dirty buffers: that is, buffers that are not the same as the corresponding blocks on the disk.

The processing of a **DELETE** or **INSERT** command uses similar steps. The before image for a **DELETE** contains the column values in the deleted row, and the before image of an **INSERT** contains the row location information.

Because the changes made to the blocks are only recorded in memory structures and are not written immediately to disk, a computer failure that causes the loss of the SGA can also lose these changes.
Redo Log Buffer Characteristics

The server process records most of the changes made to data file blocks in the redo log buffer, which is a part of the SGA. The redo log buffer has the following characteristics:

- Has its size defined by the `LOG_BUFFER` parameter.
- Records changes made through the instance.
- Is used sequentially.
- Is a circular buffer.

- It records the block that is changed, the location of the change, and the new value in a redo entry. A redo entry makes no distinction between the type of block that is changed; it simply records which bytes are changed in the block.
- The redo log buffer is used sequentially, and changes made by one transaction may be interleaved with changes made by other transactions.
- It is a circular buffer that is reused after it is filled, but only after all the old redo entries are recorded in the redo log files.
Rollback Segment

Before making a change, the server process saves the old data value into a rollback segment. This before image is used to:

- Undo the changes if the transaction is rolled back
- Provide read consistency by ensuring that other transactions do not see uncommitted changes made by the DML statement
- Recover the database to a consistent state in case of failures

Rollback segments, like tables and indexes, exist in data files, and rollback blocks are brought into the database buffer cache as required. Rollback segments are created by the DBA.

Changes to rollback segments are recorded in the redo log buffer.
Fast COMMIT

The Oracle Server uses a fast commit mechanism that guarantees that the committed changes can be recovered in case of instance failure.

System Change Number

Whenever a transaction commits, the Oracle Server assigns a commit system change number (SCN) to the transaction. The SCN is monotonically incremented and is unique within the database. It is used by the Oracle Server as an internal time stamp to synchronize data and to provide read consistency when data is retrieved from the data files. Using the SCN enables the Oracle Server to perform consistency checks without depending on the date and time of the operating system.

Steps in Processing COMMITs

When a COMMIT is issued, the following steps are performed:

- The server process places a commit record, along with the SCN, in the redo log buffer.
- LGWR performs a contiguous write of all the redo log buffer entries up to and including the commit record to the redo log files. After this point, the Oracle Server can guarantee that the changes will not be lost even if there is an instance failure.
Steps in Processing **COMMITs** (continued)

- The user is informed that the **COMMIT** is complete.
- The server process records information to indicate that the transaction is complete and that resource locks can be released.

Flush of the dirty buffers to the data file is performed independently by DBW0 and can occur either before or after the commit.

**Advantages of the Fast COMMIT**

The fast commit mechanism ensures data recovery by writing changes to the redo log buffer instead of the data files. It has the following advantages:

- Sequential writes to the log files are faster than writing to different blocks in the data file.
- Only the minimal information that is necessary to record changes is written to the log files, whereas writing to the data files would require whole blocks of data to be written.
- If multiple transactions request to commit at the same time, the instance piggybacks redo log records into a single write.
- Unless the redo log buffer is particularly full, only one synchronous write is required per transaction. If piggybacking occurs, there can be less than one synchronous write per transaction.
- Because the redo log buffer may be flushed before the **COMMIT**, the size of the transaction does not affect the amount of time needed for an actual **COMMIT** operation.

**Note:** Rolling back a transaction does not trigger LGWR to write to disk. The Oracle Server always rolls back uncommitted changes when recovering from failures. If there is a failure after a rollback, before the rollback entries are recorded on disk, the absence of a commit record is sufficient to ensure that the changes made by the transaction are rolled back.
**Log Writer (LGWR)**

LGWR writes when:
- There is a commit
- The redo buffer log is one-third full
- There is more than 1 MB of redo
- Before DBW0 writes

LOG Writer
LGWR performs sequential writes from the redo log buffer to the redo log file under the following situations:
- When a transaction commits
- When the redo log buffer is one-third full
- When there is more than a megabyte of changes recorded in the redo log buffer
- Before DBW0 writes modified blocks in the database buffer cache to the data files

Because the redo is needed for recovery, LGWR confirms the COMMIT only after the redo is written to disk.
Other Instance Processes

• Other required processes:
  – Database Writer (DBW0)
  – Process Monitor (PMON)
  – System Monitor (SMON)
  – Checkpoint (CKPT)
• The archive process (ARC0) is usually created in a production database

Other Required Processes

Four other required processes do not participate directly in processing SQL statements:
  • Database Writer (DBW0)
  • Process Monitor (PMON)
  • System Monitor (SMON)
  • Checkpoint (CKPT)

The checkpoint process is used to synchronize database files.

The Archiver Process

All other background processes are optional, depending on the configuration of the database; however, one of them, ARC0, is crucial to recovering a database after the loss of a disk. The ARC0 process is usually created in a production database.
**Database Writer (DBW0)**

The server process records changes to rollback and data blocks in the buffer cache. The Database Writer (DBW0) writes the dirty buffers from the database buffer cache to the data files. It ensures that a sufficient number of free buffers (buffers that can be overwritten when server processes need to read in blocks from the data files) are available in the database buffer cache. Database performance is improved because server processes make changes only in the buffer cache, and the DBW0 defers writing to the data files until one of the following events occurs:

- The number of dirty buffers reaches a threshold value
- A process scans a specified number of blocks when scanning for free buffers and cannot find any
- A timeout occurs (every three seconds)
- A checkpoint occurs (A checkpoint is a means of synchronizing the database buffer cache with the data file.)
**SMON: System Monitor**

- **Automatically recovers the instance:**
  - Rolls forward changes in the redo logs
  - Opens the database for user access
  - Rolls back uncommitted transactions
- **Coalesces free space**
- **Deallocates temporary segments**

**SMON: System Monitor**

If the Oracle instance fails, any information in the SGA that has not been written to disk is lost. For example, the failure of the operating system causes an instance failure. After the loss of the instance, the background process SMON automatically performs instance recovery when the database is reopened. Instance recovery consists of the following steps:

- Rolling forward to recover data that has not been recorded in the data files but that has been recorded in the online redo log. This data has not been written to disk because of the loss of the SGA during instance failure. During this process, SMON reads the redo log files and applies the changes recorded in the redo log to the data blocks. Because all committed transactions have been written to the redo logs, this process completely recovers these transactions.
- Opening the database so users can log on. Any data that is not locked by unrecovered transactions is immediately available.
- Rolling back uncommitted transactions. They are rolled back by SMON or by the individual server processes as they access locked data.

SMON also performs some space maintenance functions:

- It combines, or coalesces, adjacent areas of free space in the data files.
- It deallocates temporary segments to return them as free space in data files. Temporary segments are used to store data during SQL statement processing.
**PMON**: Process Monitor

Cleans up after failed processes by:

- Rolling back the transaction
- Releasing locks
- Releasing other resources

**PMON Functionality**

The background process **PMON** cleans up after failed processes by:

- Rolling back the user’s current transaction
- Releasing all currently held table or row locks
- Freeing other resources currently reserved by the user
Summary

In this appendix, you should have learned how to:

• Identify database files: data files, control files, online redo logs
• Describe SGA memory structures: DB buffer cache, shared SQL pool, and redo log buffer
• Explain primary background processes: DBW0, LGWR, CKPT, PMON, SMON, and ARC0
• List SQL processing steps: parse, execute, fetch

Summary

The Oracle database includes these files:

• Control files: Contain information required to verify the integrity of the database, including the names of the other files in the database (The control files are usually mirrored.)
• Data files: Contain the data in the database, including tables, indexes, rollback segments, and temporary segments
• Online redo logs: Contain the changes made to the data files (Online redo logs are used for recovery and are usually mirrored.)

Other files commonly used with the database include:

• Parameter file: Defines the characteristics of an Oracle instance
• Password file: Authenticates privileged database users
• Archived redo logs: Are backups of the online redo logs
SGA Memory Structures

The System Global Area (SGA) has three primary structures:

- Shared pool: Stores the most recently executed SQL statements and the most recently used data from the data dictionary
- Database buffer cache: Stores the most recently used data
- Redo log buffer: Records changes made to the database using the instance

Background Processes

A production Oracle instance includes these processes:

- Database Writer (DBW0): Writes changed data to the data files
- Log Writer (LGWR): Records changes to the data files in the online redo log files
- System Monitor (SMON): Checks for consistency and initiates recovery of the database when the database is opened
- Process Monitor (PMON): Cleans up the resources if one of the processes fails
- Checkpoint Process (CKPT): Updates the database status information after a checkpoint
- Archiver (ARC0): Backs up the online redo log to ensure recovery after a media failure (This process is optional, but is usually included in a production instance.)

Depending on its configuration, the instance may also include other processes.

SQL Statement Processing Steps

The steps used to process a SQL statement include:

- Parse: Compiles the SQL statement
- Execute: Identifies selected rows or applies DML changes to the data
- Fetch: Returns the rows queried by a SELECT statement
Index

Note: A bolded number or letter refers to an entire lesson or appendix.

A

APPEND Command  C-11
ACCESS PARAMETER  20-19
Adding Data through a View  11-16
ADD_MONTHS Function  3-21
ALL Operator  6-16
Alias  1-4, 1-17, 1-16, 2-7, 2-24, 11-9
  Table Aliases  4-12
ALL_INSERT (Conditional)  20-7
ALL_COL_COMMENT Data Dictionary View  9-30
ALL_TAB_COMMENT Data Dictionary View  9-30
ALTER SEQUENCE Statement  12-12
ALTER TABLE Statement  9-20, 9-21, 10-17, 10-20, 10-21, 13-11
ALTER USER Statement  13-11
Ambiguous Column Names  4-11
American National Standards Institute  I-24
ANSI  I-24
ANY Operator  6-15
Application Server  I-5
Archived Redo Log File  E-6
Arguments  3-3, 3-5
Arithmetic Expression  1-9
Arithmetic Operator  1-9
AS Subquery Clause  9-18
Assigning Privileges  13-7
Attributes  I-16, I-19
AVG Function  5-6, 5-7
Index

B

Background Processes  E-3, E-7
BETWEEN Operator  2-10
BREAK Command  7-18
BTITLE Command  7-19

C

CHANGE Command  C-12
Caching Sequence  12-1
Calculations in Expressions  1-9
Cardinality  I-18
Cartesian Product  4-4, 4-5
CASE Expression  3-51, 3-52, 18-12
CASCADE CONSTRAINTS Clause  10-22
Character Data Type in Functions  3-4
Character Strings  2-5, 2-6
CHECK Constraint  10-16
Checkpoint Process  e-8
Child Node  19-10
CLEAR BREAK Command  7-18
COALESCE Function  3-49
COLUMN Command  7-16, 7-17
Column Level Constraints  10-8
Command or Script Files  7-20
COMMENT Statement  9-30
COMMIT Statement  8-2, 8-33, 8-35, 8-39, 8-40, 9-8
Comparison Operator, Comparison Conditions  2-7, 18-4
Composite Column  17-17
Composite Unique Key  10-10
CONCAT Function  3-11
Concatenated Groupings  17-21
Concatenation Operator  1-18
Index

C

Conditional FIRST INSERT 20-7, 20-13, 20-14
Conditional If-Then-Else Logic 3-51
Conditional INSERT ALL 20-7, 20-11
Conditional Processing 3-51
Conditions, Logical 2-15
CONNECT BY Clause 19-5, 19-7, 19-13
CONSTRAINTS 10

CASCADE CONSTRAINTS Clause 10-22
CHECK Constraint 10-16
Column-Level Constraints 10-8
Defining Constraints 10-5
Deleting a Record with an Integrity Constraint 8-22
Disabling 10-20
Dropping a Constraint 10-19
FOREIGN KEY 10-13, 10-14, 10-15, I-19
NOT NULL Constraint 10-7
Primary Key 10-11
READ ONLY Constraint 11-19
REFERENCE Constraint 10-15
Referential Integrity Constraint 10-13
Table-Level Constraints 10-8
UNIQUE Constraint 10-9, 10-10

Controlling Database Access 13
Control File e-5
Correlated Subquery 18-2, 18-13, 18-14, 18-15, 18-21, 18-24
Correlated UPDATE 18-22
Correlation 18-17
COUNT Function 5-8
CREATE DATABASE Statement  16-9
CREATE DIRECTORY Statement  20-20
CREATE INDEX Statement  12-17, 20-24
Creating Scripts  1-26
CREATE SEQUENCE Statement  12-5
CREATE TABLE Statement  9
CREATE USER Statement  13-6
CREATE VIEW Statement  11-7
Cross Tabular Reports  17-9
Cross Tabulation Rows  17-6
Cross Tabulation Values  17-10
CUBE Operator  17-2, 17-6, 17-9
CURRENT_DATE Function  16-6
CURRENT_TIMESTAMP Function  16-7
CURRVAL  9-7, 12-8
CYCLE Clause (Sequences)  12-6

Date Functions  3-6
Data Control Language (DCL) Statements  8-33, 9
Data Definition Language (DDL) Statements  8-33, 9-5, 13
Data Manipulation Language (DML) Statements  8
DML Operations through a View  11-14
Data Dictionary Tables  9-9, D-3
Data Dictionary Cache  E-13, E-14
Data File  E-5
Data from More than One Table (Joins)  4
Data Structures in the Oracle Database  9-3, 9-5

Introduction to Oracle9i: SQL  Index-4
Index

D

Data Types 3-25
Data Warehouse Applications I-8
Database Links 13-19
Database Writer E-8
Date Conversion Functions 3-4, 3-35
Datetime Data Type 9-14
Datetime Functions 16-2
Daylight Savings Time 16-5
DBTIMEZONE Function 16-9
DECODE Expression 3-51, 3-54
DEFAULT Clause 8-26, 8-27, 9-7
Default Date Display 2-6, 3-17
DEFAULT DIRECTORY 20-19
Default Sort Order 2-23
DEFINE Command 7-5, 7-11
Defining Constraints 10-5
DELETE Statement 8-19, 8-20, 13-16
DESCRIBE Command 1-29, 8-7, 10-24, 11-13, C-7
DISABLE Clause 10-20
DISTINCT Keyword 1-4, 1-23, 5-5, 5-10
Dropping a Constraint 10-19
DROP ANY INDEX Statement 12-2
DROP ANY VIEW Statement 11-20
### D

- **DROP COLUMN Clause** 9-25
- **DROP INDEX Statement** 12
- **DROP SEQUENCE Statement** 12-14
- **DROP SYNONYM** 12-24
- **DROP TABLE Statement** 9-27
- **DROP UNUSED COLUMNS Clause** 9-26
- **DROP VIEW Statement** 11-20
- **DUAL Table** 3-14, 3-18

Duplicate Records 15-11

### E

- **E-business** 19-6, I-3
- **EDIT Command** C-14
- **Entity** I-16, I-17, I-18
- **Entity Relationship Diagram** I-16, I-17, I-16
- **Equijoins** 4-8, 4-27
- **ESCAPE Option** 2-13
- **Exclusive Locks** 8-46
E

Execute Button (in *SQL*+Plus)  1-7, 1-32
Executing SQL  1-26
EXISTS Operator   18-18, 18-19
Explicit Data Type Conversion  3-25
Expressions
  Calculations in Expressions  1-9
  CASE Expression  3-51, 3-52, 18-12
  DECODE Expression  3-51, 3-54
  If-Then-Else Logic  3-51
External Tables  20
  Conditional *FIRST INSERT*  20-7, 20-13, 20-14
  Conditional *INSERT ALL*  20-7, 20-11
  ORGANIZATION EXTERNAL Clause  20-18, 20-19
  Pivoting *INSERT*  20-7, 20-15
  Unconditional *INSERT*  20-7, 20-10
  *REJECT LIMIT Clause*  20-19
  *TYPE ACCESS_DRIVER_TYPE*  20-19

EXTRACT Function  16-10

F

FOREIGN KEY Constraint  10-13, 10-14, 10-15, 0-19
Format Mode (fm)  3-31
FRACTIONAL_SECONDS_PRECISION  9-15
FROM Clause  1
FROM Clause Query  11-21, 18-2, 18-10
FROM_TZ Function  16-11

Introduction to Oracle9i: SQL Index-7
Index

F
    Functions 3, 5
    AVG (Average) 5-6, 5-7
    Character Data Type in Functions 3-4
    COALESCE Function 3-49
    CONCAT Function 3-11
    COUNT Function 5-8
    CURRENT_DATE Function 16-6
    CURRENT_TIMESTAMP Function 16-7
    Date Conversion Functions 3-4, 3-35
    Datetime Functions 16-2
    DBTIMEZONE Function 16-9
    EXTRACT Function 16-10
        TIMEZONE_ABBR 16-10
        TIMEZONE_REGION 16-10
    FROM_TZ Function 16-11
    INITCAP Function 3-9
    INSTR Function 3-11
    LAST_DAY Function 3-21
    LENGTH Function 3-11
    LOCALTIMESTAMP Function 16-8
    LOWER Function 3-9
    LPAD Function 3-11
    MAX Function 5-6, 5-7
    MIN Function 5-6, 5-7
    MONTHS_BETWEEN Function 3-6, 3-21
    Multiple-row Function 3-4
Functions 3, 5

NEXT_DAY Function 3-21
NULLIF Function 3-48
Number Functions 3-13
NVL Function 3-45, 3-46, 5-5, 5-12
NVL2 Function 3-47
Returning a Value 3-3
ROUND Function 3-14, 3-21, 3-23
SESSIONTIMEZONE Function 16-9
STDDEV Function 5-7
SUBSTR Function 3-11
SUM Function 5-6, 5-7
SYS Function 9-9
SYSDATE Function 3-18, 3-20, 9-7
TO_CHAR Function 3-31, 3-37, 3-39
TO_DATE Function 3-39
TO_NUMBER Function 3-39
TO_TIMESTAMP Function 16-12
TO_YMINTERVAL Function 16-13
TRIM Function 3-11
TRUNC Function 3-15, 3-21, 3-23
TZOFFSET 16-14 Function
UPPER Function 3-9, 3-10
USER Function 9-7

Function-based Indexes 12-21
# Index

## I
- **INITCAP Function** 3-9
- **Inline Views** 11-2, 11-21
- **Inner Query** 6-3, 6-4, 6-5, 18-5
- **INSERT Statement** 8-5, 8-6, 8-11, 13-18, 20-2, 20-7
  - Conditional **FIRST INSERT** 20-7, 20-13, 20-14
  - Conditional **INSERT ALL** 20-7, 20-11
  - Pivoting **INSERT** 20-7, 20-15
  - Unconditional **INSERT** 20-7, 20-10
- **VALUES Clause** 8-5

## INSTR Function 3-11

## Integrity Constraints 8-17, 10-2

## International Standards Organization (ISO) 1-24

## Internet Features 1-7

## INTERSECT Operator 15-12

## INTERVAL YEAR TO MONTH Data Type 9-17

## IS NOT NULL Operator 2-14

## IS NULL Operator 2-14

## iSQL*Plus 1-24

## J
- **Java** 1-23
- **Joining Tables** 1-3, 4
  - Cartesian Product 4-4, 4-5
  - Equijoins 4-8, 4-27
  - Joining a Table to Itself 4-19
  - Joining More than Two Tables 4-13
  - Joining When there is No Matching Record 4-34
J
Joining Tables  1-3, 4
   Left Table  4-32
   Natural Joins  4-24, 4-26
   Nonequijoins  4-14, 4-15
   ON Clause  4-28, 4-29
   Outer Join  4-17, 4-18
   RIGHT Table  4-33
   Three-Way Join  4-30

K
Keywords  1-4, 1-7

L
   LAST_DAY Function  3-21
   LENGTH Function  3-11
   LEVEL Psuedocolumn  19-10
   Library Cache  e-13
   LIKE Operator  2-12
   LIST Command  c-11
   Literal Values  1-20
   Loading Scripts  1-32
   LOCALTIMESTAMP Function  16-8
   Locks  8-45
      Exclusive Locks  8-46
   Logical Conditions  2-15
   Logical Subsets  11-4
   LogWriter (LGWR)  E-6, E-8
   LOWER Function  3-9
   LPAD Function  3-11
<table>
<thead>
<tr>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MAX</strong> Function 5-6, 5-7</td>
</tr>
<tr>
<td><strong>MERGE</strong> Statement 8-28, 8-29</td>
</tr>
<tr>
<td><strong>WHEN NOT MATCHED</strong> Clause 8-31</td>
</tr>
<tr>
<td><strong>MIN</strong> Function 5-6, 5-7</td>
</tr>
<tr>
<td><strong>MINUS</strong> Operator 15-14</td>
</tr>
<tr>
<td><strong>MODIFY</strong> Clause 9-24</td>
</tr>
<tr>
<td><strong>Modify Column</strong> 9-23</td>
</tr>
<tr>
<td><strong>MONTHS_BETWEEN</strong> Function 3-6, 3-21</td>
</tr>
<tr>
<td><strong>Multiple Column Subquery</strong> 6-7, 18-2, 18-8</td>
</tr>
<tr>
<td><strong>Multiple-row Function</strong> 3-4</td>
</tr>
<tr>
<td><strong>Multiple-row Subquery</strong> 6-2, 6-7, 6-14, 18-6</td>
</tr>
<tr>
<td><strong>Multitable Inserts</strong> 20-2, 20-5, 20-7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Naming Conventions for Tables</strong> 9-4</td>
</tr>
<tr>
<td><strong>Naming Indexes</strong> 20-2</td>
</tr>
<tr>
<td><strong>Natural Joins</strong> 4-24, 4-26</td>
</tr>
<tr>
<td><strong>Nested Queries</strong> 6-4, 18-4</td>
</tr>
<tr>
<td><strong>Nested Functions</strong> 3-42</td>
</tr>
<tr>
<td><strong>NEXT_DAY</strong> Function 3-21</td>
</tr>
<tr>
<td><strong>NEXTVAL</strong> Pseudocolumn 9-7, 12-8</td>
</tr>
<tr>
<td><strong>Nonequijoins</strong> 4-14, 4-15</td>
</tr>
<tr>
<td><strong>Nonpairwise Comparisons</strong> 18-7</td>
</tr>
<tr>
<td><strong>Non-unique Indexes</strong> 12-16</td>
</tr>
<tr>
<td><strong>NOT EXISTS</strong> Operator 18-20</td>
</tr>
<tr>
<td><strong>NOT IN</strong> Operator 18-20</td>
</tr>
<tr>
<td><strong>NOT NULL</strong> Constraint 10-7</td>
</tr>
<tr>
<td><strong>NULL</strong> 1-14, 1-15, 2-14, o-19</td>
</tr>
<tr>
<td><strong>NULLIF</strong> Function 3-48</td>
</tr>
<tr>
<td><strong>Number Functions</strong> 3-13</td>
</tr>
<tr>
<td><strong>NVL</strong> Function 3-45, 3-46, 5-5, 5-12</td>
</tr>
<tr>
<td><strong>NVL2</strong> Function 3-47</td>
</tr>
</tbody>
</table>

*Introduction to Oracle9i: SQL Index-13*
Object Privileges 13-2
Object Relational Database Management System (ORDBMS) I-2, I-7, I-12
Object-oriented Programming I-7
ON Clause 4-28, 4-29
ON DELETE CASCADE Clause 10-15
ON DELETE SET NULL Clause 10-15
On Line Transaction Processing I-8
OR REPLACE Clause 11-12
Oracle Instance E-3, E-7, I-23
Oracle9i Application Server I-4
Oracle9i Database I-4
ORDER BY Clause 2, 15-20
Default Sort Order 2-23
Order of Precedence 1-12
ORGANIZATION EXTERNAL Clause 20-18, 20-19
Outer Join 4-17, 4-18
Outer Query 6-5, 18-5

Pairwise Comparisions 18-7
Parameter File E-6
Parent-child Relationship 19-4
Password File E-6
Pivoting INSERT 20-7, 20-15
Primary Key 10-11
PRIOR Clause 19-7
Privileges 13
Object Privileges 13-2
Process Monitor E-8
Program Global Area E-16
Projection 1-3
PUBLIC Keyword 13-5

Introduction to Oracle9i: SQL Index-14
Read Consistency  8-43, 8-44
READ ONLY Constraint  11-19
REM Command  7-21
REFERENCE Constraint  10-13, 10-15
Referential Integrity Constraint  10-13
REJECT LIMIT Clause  20-19
Relational Database Management System (RDBMS)  I-2, I-13, I-14
Relationships  I-16
RENAME Command  9-28
Restricting Rows  2-2
Retrieving Data from a View  11-10
Returning a Value  3-3
REVOKE Command  13-17
ROLLBACK Statement  8-2, 8-33, 8-35, 8-38, 8-41, E-20
Rollback Segment  e-20
ROLLUP Clause  17-2, 17-6, 17-7, 17-8
Root Node  19-10
ROUND Function  3-14, 3-21, 3-23
Row  I-19, 17-8
ROWNUMBER Psuedocolumn
RR Date Format  3-41
Rules of Precedence  1-13, 2-19
Index

S

SAVE Command  C-14
SAVEPOINT Statement  8-2, 8-35, 8-36
Scalar Subquery  18-11
Schema  9-6, 13-4
Script or Command Files  7-20, 7-22, C-2
   Creating Scripts  1-26
   Loading Scripts  1-32
Search  2-12
SELECT Statement  1
Selection  1-3
Sequences  9-13, 12
   Caching Sequence Values  12-11
   CREATE SEQUENCE Statement  12-5
   CURRVAL  9-7, 12-8
   CYCLE Clause  12-6
   Generating Unique Numbers  12-3
   NEXTVAL  9-7, 12-8
Server Architecture  e-2
SESSIONTIMEZONE Function  16-9
SET Command  7-12
SET Clause  8-15
SET Operators  15-2, 15-3
SET TIME_ZONE Clause  16-9
SET UNUSED Clause  9-26
SET VERIFY ON Command  7-7
Sets of Rows  5-3
Shared Global Area  I-23, E-7
Shared SQL Area  E-14
Single Ampersand Substitution  7-4

Introduction to Oracle9i: SQL Index-16
S

Single Row Function  3-4
Single Row Operators  6-8
Single Row Subqueries  6-2, 6-7
SMON Process  E-8
SOME Operator  6-15
Sorting Results with the ORDER BY Clause  2
   Default Sort Order  2-23
Spool File  D-5
Structured Query Language (SQL)  1-2, 1-21, 1-22, 1-2, 1-24, 1-25
SQL Buffer  C-3
SQL Scripts  D-2
SQL*Plus  C
   SQL*Plus Commands  C-2
   SQL*Plus Script File  7-3
SQL: 1999 Compliance  4-6, 4-22, 4-30
START Command  C-14
START WITH Clause  19-5, 19-6
Statement  1-4
Statement Level Rollback  8-42
STDDEV Function  5-7
Subqueries  6, 8-16, 8-21, 8-23, 9-18, 11-21, 18-2, 18-3, 18-10

   AS  Subquery Clause  9-18
   Correlated Subquery  18-2, 18-13, 18-14, 18-15, 18-21, 18-24
   Correlated UPDATE  18-22
   FROM Clause Query  11-21, 18-2, 18-10
   Group Functions in a Subquery  6-10
   Inner Query  6-3, 6-4, 6-5, 18-5
   Multiple Column Subquery  6-7, 18-2, 18-8
   Multiple-row Subquery  6-2, 6-7, 6-14, 18-6
   Nested Queries  6-4, 18-4
   No Rows Returned from the Subquery  6-13
   Outer Query  6-5, 18-5
   Placement of the Subquery  6-4
   Scalar Subquery  18-11
   Single Row Subqueries  6-2, 6-7

Subsets, Logical  11-4
Substitution Variables  7-2, 7-3
SUBSTR Function  3-11
SUM Function  5-6, 5-7
Summary Results for Groups of Rows  5-18
Superaggregate Rows  17-7, 17-8, 17-9
SYS Function  9-9
Synonym  9-3, 12-2, 12-3, 12-23, 13-3
SYSDATE Function  3-18, 3-20, 9-7
System Development Life Cycle  I-10
System Global Area  I-23, E-3, E-8
Table Aliases 4-12
Table Level Constraints 10-8
Table Prefixes 4-11
Three-Way Join 4-30
Time Zone 16-3
TIMESTAMP Data Type 9-16
   TIMESTAMP WITH TIME ZONE 9-15
   TIMESTAMP WITH LOCAL TIME 9-16
   INTERVAL YEAR TO MONTH 9-17
TIMEZONE_ABBR 16-10
TIMEZONE_REGION 16-10
TO_CHAR Function 3-31, 3-37, 3-39
TO_DATE Function 3-39
TO_NUMBER Function 3-39
TO_TIMESTAMP Function 16-12
TO_YMINTERVAL Function 16-13
Top-n Analysis 11-2, 11-22, 11-23, 11-24
Transactions 8-32
Tree Structured Report 19
TRIM Function 3-11
TRUNC Function 3-15, 3-21, 3-23
TRUNCATE TABLE Statement 9-29
TTITLE Command 7-19
Tuple l-19
TYPE ACCESS_DRIVER_TYPE 20-19
TZOFFSET Function 16-14
UNDEFINE Command 7-11
UNION Operator 15-7, 15-8, 15-11
UNION Operator 15-10, 15-11
UNIQUE Constraint 10-9, 10-10
Unique Identifier I-18
Unique Index 10-10, 12-6
UPDATE Statement 8, 13-14
  SET Clause 8-15
  Correlated UPDATE 18-22
UPPER Function 3-9, 3-10
Users - (Creating) 13-6
USER Function 9-7
User Process E-10
USER_CATALOG Dictionary View 9-10
USER_COL_COMMENTS Dictionary View 9-30
USER_COL_PRIVS_MADE Dictionary View D-4
USER_CONS_COLUMNS Dictionary View 10-19, 10-25
USER_CONSTRAINTS Dictionary View 10-4, 10-19, 10-24
USER_DB_LINKS Dictionary View 13-19
USER_INDEXES Dictionary View 12-20
USER_OBJECTS Dictionary View 9-10, D-4
USER_SEQUENCES Dictionary View 12-7
USER_TAB_COMMENTS Dictionary View 9-30
USER_TAB_PRIVS_MADE Dictionary View D-4
USER_TABLES Dictionary View 9-10, D-4
USER_UNUSED_COL_TABS Dictionary View 9-26
USING Clause 4-26, 13-20
UTC (Coordinated Universal Time) 9-15
VALUES Clause 8-5
Variance 5-7
VERIFY Command 7-7
Views 9-3, 11

Guidelines for Creating a View 11-8
Inline Views 11-2, 11-21
OR REPLACE Clause 11-12
Retrieving Data from a View 11-10
Simple and Complex 11-6
USING Clause 4-26
WITH READ ONLY Clause 11-18

V$TIMEZONE_NAME Dictionary View 16-11

WHEN NOT MATCHED Clause 8-31
WHERE Clause 2

Restricting Rows 2-2
Wildcard Symbol 2-12
WITH Clause 18-2, 18-26
WITH CHECK OPTION Clause 8-25, 11-17, 13-13, 13-14, 13-15, 13-18
WITH READ ONLY Clause 11-18

XML I-23

Year 2000 Compliance 3-17