Disclaimer

This course provides an overview of features and enhancements planned in release 11g. It is intended solely to help you assess the business benefits of upgrading to 11g and to plan your IT projects.

This course in any form, including its course labs and printed matter, contains proprietary information that is the exclusive property of Oracle. This course and the information contained herein may not be disclosed, copied, reproduced, or distributed to anyone outside Oracle without prior written consent of Oracle. This course and its contents are not part of your license agreement nor can they be incorporated into any contractual agreement with Oracle or its subsidiaries or affiliates.

This course is for informational purposes only and is intended solely to assist you in planning for the implementation and upgrade of the product features described. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described in this document remain at the sole discretion of Oracle.

This document contains proprietary information and is protected by copyright and other intellectual property laws. You may copy and print this document solely for your own use in an Oracle training course. The document may not be modified or altered in any way. Except where your use constitutes "fair use" under copyright law, you may not use, share, download, upload, copy, print, display, perform, reproduce, publish, license, post, transmit, or distribute this document in whole or in part without the express authorization of Oracle.

The information contained in this document is subject to change without notice. If you find any problems in the document, please report them in writing to: Oracle University, 500 Oracle Parkway, Redwood Shores, California 94065 USA. This document is not warranted to be error-free.

Restricted Rights Notice

If this documentation is delivered to the United States Government or anyone using the documentation on behalf of the United States Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS
The U.S. Government’s rights to use, modify, reproduce, release, perform, display, or disclose these training materials are restricted by the terms of the applicable Oracle license agreement and/or the applicable U.S. Government contract.

Trademark Notice

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.
Index
A

Alias 1-17
ALL Operator 7-19, 8-16, 8-17
ALTER TABLE Statement 10-35
Alternative Quote (q) Operator 1-23
American National Standards Institute (ANSI) i-30, 4-4, 6-5
Ampersand Substitution 2-29, 2-30, 2-33
AND Operator 2-16, 2-21, C-12
ANY Operator 7-18
Arithmetic Expressions 1-11, 1-15, 1-19
Arithmetic Operators 1-11, 1-12, 3-26
Attributes i-23

B

BETWEEN Operator 2-10
BI Publisher i-14

C

Cartesian Product 6-33, C-5
CASE Expression 4-37, 4-38
Character strings 2-7
CHECK Constraint 10-27
COALESCE Function 4-32, 4-33, 4-34
Column Alias 1-17
Comparison Operators 2-8, 2-9
Concatenation Operator 1-20
Constraints 9-4, 10-2, 10-16, 10-17, 10-18, 10-19, 10-29, 10-30, E-6
Conversion Functions i-5, 3-7, 3-11, 3-12, 3-28, 4-1, 4-4, 4-9
COUNT Function 5-9
CREATE SEQUENCE Statement 11-25
CREATE TABLE Statement 10-7
Creating a Database Connection i-37, i-38, i-39, i-58
Cross Joins 6-34
CURRENT_DATE 3-24, 9-9
CURRVAL 10-9, 10-27, 11-3, 11-22, 11-27, 11-28, 11-29, 11-33, 11-40
D
Data Types 4-28, 10-12, 10-13, 10-14
Database i-2, i-3, i-4, i-8, i-9, i-10, i-11, i-12, i-13,
   i-14, i-15, i-16, i-17, i-18, i-19, i-27, i-28, i-29, i-30, i-33,
   i-35, i-37, i-38, i-39, i-40, i-47, i-49, i-50, i-53, i-54, i-55,
   i-56, i-58, i-59, 1-14, 1-15, 3-4, 3-5, 3-10, 3-16, 3-24, 4-9,
   4-27, 5-27, 6-2, 6-6, 7-8, 9-3, 9-13, 9-15, 9-19, 9-21, 9-25,
   9-26, 9-27, 9-31, 9-39, 9-40, 9-42, 10-3, 10-4, 10-5, 10-6, 10-11,
   10-14, 10-15, 10-17, 10-31, 10-34, 10-36, 10-37, 10-38, 11-4, 11-6, 11-16,
   11-26, 11-28, 11-31, 11-35, 11-37, 11-42, 11-43, C-2, D-17, D-19, E-3,
   E-13, E-15, E-19, E-20
Database Transactions 9-26, 9-27
Date i-18, 1-9, 1-21, 1-33, 2-24, 2-31, 3-3, 3-5, 3-7, 3-8,
   3-15, 3-20, 3-22, 3-23, 3-24, 3-27, 3-28, 3-29, 3-31, 3-33, 4-12,
   4-13, 4-14, 4-22, 9-10, 10-12, 10-14
Datetime Data Types 10-14
DBMS i-17, D-17, D-19
DECODE Function 4-39, 4-40, 4-41
DEFAULT Option 10-9
DELETE Statement 9-21
DESCRIBE Command 1-27
DISTINCT Keyword 5-10
DUAL Table 3-17
Duplicate Rows 1-24
E
Entity Relationship i-21, i-22, i-23, B-3
Equijoins 6-12, 6-35, C-9, C-10, C-11, C-22
Execute SQL D-5, D-20
Execute Statement icon i-44, i-47, 1-8, 1-30, 9-48, 10-40
Explicit Data Type Conversion 4-7, 4-8, 4-9
F
FOR UPDATE clause 9-3, 9-13, 9-19, 9-25, 9-39, 9-42, 9-43, 9-44,
   9-46
Format Model 4-12, 4-14
F
Functions i-5, 2-7, 3-1, 3-2, 3-4, 3-5, 3-6, 3-7, 3-9, 3-10, 3-11, 3-12, 3-13, 3-14, 3-15, 3-16, 3-28, 3-29, 3-30, 4-1, 4-4, 4-9, 4-20, 4-21, 4-24, 4-25, 4-27, 5-1, 5-4, 5-5, 5-6, 5-7, 5-8, 5-11, 5-19, 5-20, 5-26, 7-12, E-17

G
GROUP BY Clause 5-14, 5-15, 5-16, 5-18
Group Functions i-5, 3-5, 5-1, 5-4, 5-5, 5-6, 5-11, 5-19, 5-20, 5-26, 7-12
Group Functions in a Subquery 7-12

H
HAVING Clause 5-22, 5-23, 5-24, 7-13

I
Implicit Data Type Conversion 4-5, 4-6
IN Operator 2-11
Index 10-4, 11-23, 11-34, 11-37, 11-38, 11-39, 11-41, E-14
INSERT Statement 9-6
International Standards Organization (ISO) i-31
INTERSECT Operator 8-19, 8-20
INTERVAL YEAR TO MONTH 10-14

J
Java i-9, i-35, i-56
Joining Tables 6-6, C-7

K
Keywords 1-8, 10-26, D-4

L
LIKE Operator 2-12
Literal 1-21, 1-22, 10-9

M
MINUS Operator 8-22, 8-23
MOD Function 3-19

N
Naming 10-3, 10-5, 10-6, 10-11, 10-15, 10-31, 10-34, 10-37
NEXTVAL 10-9, 10-27, 11-3, 11-22, 11-27, 11-28, 11-29, 11-33, 11-40
NEXTVAL and CURRVAL Pseudocolumns 11-27, 11-28
Nonequijoins 6-3, 6-8, 6-19, 6-22, 6-23, 6-24, 6-25, 6-35, C-14, C-15, C-22
NOT NULL Constraint 10-20
NOT Operator 2-18
NULL Conditions 2-14
Null Value 1-14
Null Values 1-15, 1-20, 5-11, 7-21, 7-22, 9-8
NULLIF Function 4-31
Number Functions 3-16
NVL Function 4-28, 4-29
NVL2 Function 4-30
Object Relational i-16
OLTP i-11, i-16
ON clause 6-3, 6-5, 6-6, 6-8, 6-15, 6-16, 6-18, 6-19, 6-21, 6-22, 6-25, 6-31
ON DELETE CASCADE 10-26
ON DELETE SET NULL 10-26
OR Operator 2-17
Oracle Database 11g i-2, i-3, i-4, i-8, i-9, i-10, i-11, i-14, i-15, i-29, i-33, i-49, i-50, i-53, i-54, i-55, i-56, 3-24, 7-8, 10-14, 10-36, 10-38
Oracle Enterprise Manager Grid Control 10g i-13, i-56
Oracle Fusion Middleware i-12, i-13, i-56
Oracle Server 8-6
Oracle SQL Developer i-2, i-3, i-7, i-8, i-15, i-29, i-32, i-33, i-34, i-35, i-36, i-37, i-38, i-40, i-41, i-45, i-47, i-48, i-49, i-50, i-53, i-57, i-58, i-59, E-20
ORDBMS i-2, i-56
Order 2-40, 3-35, 4-25, 4-45, 6-39, B-2, C-26
ORDER BY Clause 2-23, 8-28
PRIMARY KEY Constraint 10-23
Projection 1-4
P
Pseudocolumns 11-27, 11-28

Q
q operator 1-23
Queries i-5, 5-19, 5-20, 6-2, 7-1, 7-8, 8-4, 8-5, 10-27, C-2
Query i-30, 7-15, D-3, E-11

R
RDBMS i-2, i-18, i-25, i-27, i-56, 9-43
Read Consistency 9-40, 9-41
Read-only tables 10-3, 10-6, 10-11, 10-15, 10-31, 10-34, 10-37
REFERENCES 10-25, 10-26, 10-28
Relational Database i-16, i-18, i-19, i-27, i-28
ROUND and TRUNC Functions 3-30
ROUND Function 3-17
RR Date Format 3-22, 4-22
Rules of Precedence 1-12, 2-20, 2-21

S
Schema i-6, i-51, 10-5, 10-8, 11-1, 11-35, B-2, E-4
SELECT Statement i-5, 1-1, 1-5, 1-19, 1-28, 8-26, 9-43
Selection 1-4, 2-4
Sequences 11-23, 11-24, E-13
Set operators 8-4, 8-5
SET VERIFY ON 2-36
Sorting i-5, 2-1, 2-3, 2-19, 2-22, 2-24, 2-25, 2-26, 2-34, 2-38
SQL Developer i-2, i-3, i-7, i-8, i-9, i-15, i-29, i-32,
i-33, i-34, i-35, i-36, i-37, i-38, i-40, i-41, i-45, i-47, i-48,
i-49, i-50, i-53, i-54, i-57, i-58, i-59, 1-6, 1-8, 1-9, 1-14,
1-17, 1-26, 1-30, 2-28, 2-29, 2-30, 2-31, 2-33, 2-35, 2-36, 6-16,
9-4, 9-21, 9-27, 9-31, 9-32, 9-43, 10-9, 10-40, 11-8, C-10, E-1,
E-2, E-3, E-4, E-16, E-18, E-19, E-20, E-21
Subquery 7-3, 7-4, 7-5, 7-6, 7-8, 7-9, 7-12, 7-15, 7-16,
7-20, 7-21, 7-22, 9-17, 10-32, 10-33
Substitution Variables 2-27, 2-28, 2-31
S
Synonym i-23, 10-4, 11-23, 11-34, 11-41, 11-42, 11-43, E-15
SYSDATE Function 3-24

T
TO_CHAR Function 4-11, 4-16, 4-17, 4-18, 4-19
Transactions 9-26, 9-27
TRUNC Function 3-18

U
UNION ALL Operator 8-16, 8-17
UNION Operator 8-13, 8-14, 8-15
UNIQUE Constraint 10-21, 10-22
Unique Identifier i-23
UPDATE Statement 9-15
USING Clause 6-11, 6-13, 6-14
Using Snippets E-16, E-17

V
VARIANCE 5-5, 5-8, 5-27
VERIFY Command 2-36
Views i-40, 11-6, 11-7, E-3, E-11, E-12

W
When to Create an Index 11-38
WHERE Clause 2-6, 6-10
WITH CHECK OPTION 11-8, 11-9, 11-17

X
XML i-9, i-14, i-37, i-38, i-56
Using Subqueries to Solve Queries
Objectives

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

• Define subqueries
• Describe the types of problems that the subqueries can solve
• List the types of subqueries
• Write single-row and multiple-row subqueries

Objectives

In this lesson, you learn about the more advanced features of the `SELECT` statement. You can write subqueries in the `WHERE` clause of another SQL statement to obtain values based on an unknown conditional value. This lesson also covers single-row subqueries and multiple-row subqueries.
Lesson Agenda

• Subquery: Types, syntax, and guidelines
  • Single-row subqueries:
    – Group functions in a subquery
    – HAVING clause with subqueries
  • Multiple-row subqueries
    – Use ALL or ANY operator
• Null values in a subquery
Using a Subquery to Solve a Problem

Who has a salary greater than Abel’s?

Main query:

Which employees have salaries greater than Abel’s salary?

Subquery:

What is Abel’s salary?

Using a Subquery to Solve a Problem

Suppose you want to write a query to find out who earns a salary greater than Abel’s salary. To solve this problem, you need two queries: one to find how much Abel earns, and a second query to find who earns more than that amount. You can solve this problem by combining the two queries, placing one query inside the other query. The inner query (or subquery) returns a value that is used by the outer query (or main query). Using a subquery is equivalent to performing two sequential queries and using the result of the first query as the search value in the second query.
Subquery Syntax

A subquery is a SELECT statement that is embedded in the clause of another SELECT statement. You can build powerful statements out of simple ones by using subqueries. They can be very useful when you need to select rows from a table with a condition that depends on the data in the table itself.

You can place the subquery in a number of SQL clauses, including the following:
- WHERE clause
- HAVING clause
- FROM clause

In the syntax:

\[
\text{operator includes a comparison condition such as } >, =, \text{ or IN}
\]

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

The subquery is often referred to as a nested SELECT, sub-SELECT, or inner SELECT statement. The subquery generally executes first, and its output is used to complete the query condition for the main (or outer) query.

<table>
<thead>
<tr>
<th>SELECT</th>
<th>select_list</th>
</tr>
</thead>
<tbody>
<tr>
<td>FROM</td>
<td>table</td>
</tr>
<tr>
<td>WHERE</td>
<td>expr \ operator</td>
</tr>
</tbody>
</table>

\[
(\text{SELECT select_list FROM table});
\]
Using a Subquery

In the slide, the inner query determines the salary of employee Abel. The outer query takes the result of the inner query and uses this result to display all the employees who earn more than employee Abel.

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>King</td>
<td>24000</td>
</tr>
<tr>
<td>Hooper</td>
<td>17000</td>
</tr>
<tr>
<td>De Hoan</td>
<td>17000</td>
</tr>
<tr>
<td>Harris</td>
<td>13000</td>
</tr>
<tr>
<td>Higgins</td>
<td>12000</td>
</tr>
</tbody>
</table>
Guidelines for Using Subqueries

• Enclose subqueries in parentheses.
• Place subqueries on the right side of the comparison condition for readability (However, the subquery can appear on either side of the comparison operator.).
• Use single-row operators with single-row subqueries and multiple-row operators with multiple-row subqueries.
Types of Subqueries

• Single-row subquery

Main query
returns
ST_CLERK

Subquery

• Multiple-row subquery

Main query
returns
ST_CLERK
SA_MAN

Subquery

Types of Subqueries

• **Single-row subqueries**: Queries that return only one row from the inner `SELECT` statement.

• **Multiple-row subqueries**: Queries that return more than one row from the inner `SELECT` statement.

**Note**: There are also multiple-column subqueries, which are queries that return more than one column from the inner `SELECT` statement. These are covered in the *Oracle Database 11g: SQL Fundamentals II* course.
Lesson Agenda

• Subquery: Types, syntax, and guidelines
• Single-row subqueries:
  – Group functions in a subquery
  – HAVING clause with subqueries
• Multiple-row subqueries
  – Use ALL or ANY operator
• Null values in a subquery
Single-Row Subqueries

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

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

A single-row subquery is one that returns one row from the inner `SELECT` statement. This type of subquery uses a single-row operator. The slide gives a list of single-row operators.

Example:

Display the employees whose job ID is the same as that of employee 141:

```sql
SELECT last_name, job_id
FROM employees
WHERE job_id = (SELECT job_id
                FROM employees
                WHERE employee_id = 141);
```

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>JOB_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rajs</td>
<td>ST_CLERK</td>
</tr>
<tr>
<td>Davies</td>
<td>ST_CLERK</td>
</tr>
<tr>
<td>Matos</td>
<td>ST_CLERK</td>
</tr>
<tr>
<td>Vargas</td>
<td>ST_CLERK</td>
</tr>
</tbody>
</table>
Executing Single-Row Subqueries

A `SELECT` statement can be considered as a query block. The example in the slide displays employees who do the same job as “Taylor,” but earn more salary than him.

The example consists of three query blocks: the outer query and two inner queries. The inner query blocks are executed first, producing the query results `SA_REP` and `8600`, respectively. The outer query block is then processed and uses the values that were returned by the inner queries to complete its search conditions.

Both inner queries return single values (`SA_REP` and `8600`, respectively), so this SQL statement is called a single-row subquery.

Note: The outer and inner queries can get data from different tables.
Using Group Functions in a Subquery

You can display data from a main query by using a group function in a subquery to return a single row. The subquery is in parentheses and is placed after the comparison condition.

The example in the slide displays the employee last name, job ID, and salary of all employees whose salary is equal to the minimum salary. The MIN group function returns a single value (2500) to the outer query.
The **HAVING** Clause with Subqueries

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

```sql
SELECT department_id, MIN(salary) 
FROM employees 
GROUP BY department_id 
HAVING MIN(salary) > (SELECT MIN(salary) 
FROM employees 
WHERE department_id = 50);
```

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>MIN(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(null)</td>
<td>7000</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>10</td>
</tr>
</tbody>
</table>

---

The **HAVING** Clause with Subqueries

You can use subqueries not only in the **WHERE** clause, but also in the **HAVING** clause. The Oracle server executes the subquery and the results are returned into the **HAVING** clause of the main query. The SQL statement in the slide displays all the departments that have a minimum salary greater than that of department 50.

**Example:**

Find the job with the lowest average salary.

```sql
SELECT job_id, AVG(salary) 
FROM employees 
GROUP BY job_id 
HAVING AVG(salary) = (SELECT MIN(AVG(salary)) 
FROM employees 
GROUP BY job_id);
```

<table>
<thead>
<tr>
<th>JOB_ID</th>
<th>AVG(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST_CLERK</td>
<td>2925</td>
</tr>
</tbody>
</table>
What Is Wrong with This Statement?

A common error with subqueries occurs when more than one row is returned for a single-row subquery.

In the SQL statement in the slide, the subquery contains a `GROUP BY` clause, which implies that the subquery will return multiple rows, one for each group that it finds. In this case, the results of the subquery are 4400, 6000, 2500, 4200, 7000, 17000, and 8300.

The outer query takes those results and uses them in its `WHERE` clause. The `WHERE` clause contains an equal (=) operator, a single-row comparison operator that expects only one value. The = operator cannot accept more than one value from the subquery and, therefore, generates the error. To correct this error, change the = operator to IN.
No Rows Returned by the Inner Query

A common problem with subqueries occurs when no rows are returned by the inner query. In the SQL statement in the slide, the subquery contains a WHERE clause. Presumably, the intention is to find the employee whose name is Haas. The statement is correct, but selects no rows when executed.

Because, there is no employee named Haas. So the subquery returns no rows. The outer query takes the results of the subquery (null) and uses these results in its WHERE clause. The outer query finds no employee with a job ID equal to null, and so returns no rows. If a job existed with a value of null, the row is not returned because comparison of two null values yields a null; therefore, the WHERE condition is not true.
Lesson Agenda

• Subquery: Types, syntax, and guidelines
• Single-row subqueries:
  – Group functions in a subquery
  – HAVING clause with subqueries
• Multiple-row subqueries
  – Use ALL or ANY operator
• Null values in a subquery
Multiple-Row Subqueries

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>Equal to any member in the list</td>
</tr>
<tr>
<td>ANY</td>
<td>Must be preceded by =, !=, &gt;, &lt;, &lt;&gt;, &gt;=. Compares a value to each value in a list or returned by a query. Evaluates to FALSE if the query returns no rows.</td>
</tr>
<tr>
<td>ALL</td>
<td>Must be preceded by =, !=, &gt;, &lt;, &lt;&gt;, &gt;=. Compares a value to every value in a list or returned by a query. Evaluates to TRUE if the query returns no rows.</td>
</tr>
</tbody>
</table>

**Multiple-Row Subqueries**

Subqueries that return more than one row are called multiple-row subqueries. You use a multiple-row operator, instead of a single-row operator, with a multiple-row subquery. The multiple-row operator expects one or more values:

```sql
SELECT last_name, salary, department_id
FROM   employees
WHERE  salary IN (SELECT   MIN(salary)
                    FROM     employees
                    GROUP BY department_id);
```

**Example:**

Find the employees who earn the same salary as the minimum salary for each department.

The inner query is executed first, producing a query result. The main query block is then processed and uses the values that were returned by the inner query to complete its search condition. In fact, the main query appears to the Oracle server as follows:

```sql
SELECT last_name, salary, department_id
FROM   employees
WHERE  salary IN (2500, 4200, 4400, 6000, 7000, 8300, 8600, 17000);
```
Using the **ANY** Operator in Multiple-Row Subqueries

The **ANY** operator (and its synonym, the **SOME** operator) compares a value to each value returned by a subquery. The slide example displays employees who are not IT programmers and whose salary is less than that of any IT programmer. The maximum salary that a programmer earns is $9,000. 

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

### Using the **ANY** Operator in Multiple-Row Subqueries

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

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>JOB_ID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Vargas</td>
<td>ST_CLERK</td>
<td>2600</td>
</tr>
<tr>
<td>2</td>
<td>Massie</td>
<td>ST_CLERK</td>
<td>2850</td>
</tr>
<tr>
<td>3</td>
<td>Devore</td>
<td>ST_CLERK</td>
<td>3100</td>
</tr>
<tr>
<td>4</td>
<td>Rees</td>
<td>ST_CLERK</td>
<td>3500</td>
</tr>
<tr>
<td>5</td>
<td>Whalen</td>
<td>AD_ASSIST</td>
<td>4400</td>
</tr>
<tr>
<td>6</td>
<td>Getz</td>
<td>AC_ACCOUNT</td>
<td>8300</td>
</tr>
<tr>
<td>7</td>
<td>Taylor</td>
<td>SA_REP</td>
<td>8600</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Using the **ALL** Operator in Multiple-Row Subqueries

The **ALL** operator compares a value to every value returned by a subquery. The example in the slide displays employees whose salary is less than the salary of all employees with a job ID of **IT_PROG** and whose job is not **IT_PROG**.

> **ALL** means more than the maximum and **<ALL** means less than the minimum.

The **NOT** operator can be used with **IN**, **ANY**, and **ALL** operators.
Lesson Agenda

• Subquery: Types, syntax, and guidelines
• Single-row subqueries:
  – Group functions in a subquery
  – HAVING clause with subqueries
• Multiple-row subqueries
  – Use ALL or ANY operator
• Null values in a subquery
Null Values in a Subquery

The SQL statement in the slide attempts to display all the employees who do not have any subordinates. Logically, this SQL statement should have returned 12 rows. However, the SQL statement does not return any rows. One of the values returned by the inner query is a null value, and, therefore, the entire query returns no rows.

The reason is that all conditions that compare a null value result in a null. So whenever null values are likely to be part of the results set of a subquery, do not use the `NOT IN` operator. The `NOT IN` operator is equivalent to `<> ALL`.

Notice that the null value as part of the results set of a subquery is not a problem if you use the `IN` operator. The `IN` operator is equivalent to `=ANY`. For example, to display the employees who have subordinates, use the following SQL statement:

```sql
SELECT emp.last_name
FROM employees emp
WHERE emp.employee_id IN
  (SELECT mgr.manager_id
   FROM employees mgr);
```

```sql
0 rows selected
```
Null Values in a Subquery (continued)

Alternatively, a WHERE clause can be included in the subquery to display all employees who do not have any subordinates:

```
SELECT last_name FROM employees
WHERE employee_id NOT IN
  (SELECT manager_id
   FROM employees
   WHERE manager_id IS NOT NULL);
```
Summary

In this lesson, you should have learned how to:

• Identify when a subquery can help solve a problem
• Write subqueries when a query is based on unknown values

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

Summary

In this lesson, you should have learned how to use subqueries. A subquery is a `SELECT` statement that is embedded in the clause of another SQL statement. Subqueries are useful when a query is based on a search criterion with unknown intermediate values.

Subqueries have the following characteristics:

• Can pass one row of data to a main statement that contains a single-row operator, such as =, <>, >, >=, <, or <=
• Can pass multiple rows of data to a main statement that contains a multiple-row operator, such as IN
• Are processed first by the Oracle server, after which the `WHERE` or `HAVING` clause uses the results
• Can contain group functions
Practice 7: Overview

This practice covers the following topics:

• Creating subqueries to query values based on unknown criteria
• Using subqueries to find out the values that exist in one set of data and not in another

Practice 7: Overview

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

For practice questions, you may want to create the inner query first. Make sure that it runs and produces the data that you anticipate before you code the outer query.
Practice 7

1. The HR department needs a query that prompts the user for an employee last name. The query then displays the last name and hire date of any employee in the same department as the employee whose name they supply (excluding that employee). For example, if the user enters Zlotkey, find all employees who work with Zlotkey (excluding Zlotkey).

![Enter Substitution Variable]

![Employee Table]

2. Create a report that displays the employee number, last name, and salary of all employees who earn more than the average salary. Sort the results in order of ascending salary.

![Employee Table]
Practice 7 (continued)

3. Write a query that displays the employee number and last name of all employees who work in a department with any employee whose last name contains the letter “u.” Save your SQL statement as `lab_07_03.sql`. Run your query.

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>124 Mourgos</td>
</tr>
<tr>
<td>2</td>
<td>141 Rajs</td>
</tr>
<tr>
<td>3</td>
<td>142 Davies</td>
</tr>
<tr>
<td>4</td>
<td>143 Matos</td>
</tr>
<tr>
<td>5</td>
<td>144 Vargas</td>
</tr>
<tr>
<td>6</td>
<td>103 Hunold</td>
</tr>
<tr>
<td>7</td>
<td>104 Ernst</td>
</tr>
<tr>
<td>8</td>
<td>107 Lorentz</td>
</tr>
</tbody>
</table>

4. The HR department needs a report that displays the last name, department number, and job ID of all employees whose department location ID is 1700.

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>DEPARTMENT_ID</th>
<th>JOB_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Whalen</td>
<td>10 AD_ASST</td>
<td></td>
</tr>
<tr>
<td>King</td>
<td>90 AD_PRES</td>
<td></td>
</tr>
<tr>
<td>Kochhar</td>
<td>90 AD_VP</td>
<td></td>
</tr>
<tr>
<td>De Haan</td>
<td>90 AD_VP</td>
<td></td>
</tr>
<tr>
<td>Higgins</td>
<td>110 AC_MGR</td>
<td></td>
</tr>
<tr>
<td>Gietz</td>
<td>110 AC_ACCOUNT</td>
<td></td>
</tr>
</tbody>
</table>

Modify the query so that the user is prompted for a location ID. Save this to a file named `lab_07_04.sql`.

5. Create a report for HR that displays the last name and salary of every employee who reports to King.

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kochhar</td>
<td>17000</td>
</tr>
<tr>
<td>De Haan</td>
<td>17000</td>
</tr>
<tr>
<td>Mourgos</td>
<td>5800</td>
</tr>
<tr>
<td>Zlotkey</td>
<td>10500</td>
</tr>
<tr>
<td>Hartstein</td>
<td>13000</td>
</tr>
</tbody>
</table>
Practice 7 (continued)

6. Create a report for HR that displays the department number, last name, and job ID for every employee in the Executive department.

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>LAST_NAME</th>
<th>JOB_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>90 King</td>
<td>AD_PRES</td>
</tr>
<tr>
<td>2</td>
<td>90 Kochhar</td>
<td>AD_VP</td>
</tr>
<tr>
<td>3</td>
<td>90 De Haan</td>
<td>AD_VP</td>
</tr>
</tbody>
</table>

If you have the time, complete the following exercise:

7. Modify the query in lab_07_03.sql to display the employee number, last name, and salary of all employees who earn more than the average salary, and who work in a department with any employee whose last name contains a “u.” Resave lab_07_03.sql as lab_07_07.sql. Run the statement in lab_07_07.sql.

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>103 Hunold</td>
<td>9000</td>
</tr>
</tbody>
</table>
Using the 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

Objectives

In this lesson, you learn how to write queries by using set operators.
Lesson Agenda

• Set Operators: Types and guidelines
• Tables used in this lesson
• UNION and UNION ALL operator
• INTERSECT operator
• MINUS operator
• Matching the SELECT statements
• Using the ORDER BY clause in set operations
Set Operators

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>Rows from both queries after eliminating duplications</td>
</tr>
<tr>
<td>UNION ALL</td>
<td>Rows from both queries, including all duplications</td>
</tr>
<tr>
<td>INTERSECT</td>
<td>Rows that are common to both queries</td>
</tr>
<tr>
<td>MINUS</td>
<td>Rows in the first query that are not present in the second query</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.
Set Operator Guidelines

• The expressions in the `SELECT` lists must match in number.
• The data type of each column in the second query must match the data type of its corresponding column in the first query.
• Parentheses can be used to alter the sequence of execution.
• `ORDER BY` clause can appear only at the very end of the statement.

Set Operator Guidelines

• The expressions in the `SELECT` lists of the queries must match in number and data type. Queries that use `UNION`, `UNION ALL`, `INTERSECT`, and `MINUS` operators in their `WHERE` clause must have the same number and data type of columns in their `SELECT` list. The data type of the columns in `SELECT` list of the queries in the compound query may not be exactly the same. The column in second query must be in the same data type group (such as numeric or character) as the corresponding column in the first query.
• Set operators can be used in subqueries.
• You should use parentheses to specify the order of evaluation in queries that use the `INTERSECT` operator with other set operators. This ensures compliance with emerging SQL standards that will give the `INTERSECT` operator greater precedence than the other set operators.
The Oracle Server and Set Operators

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

The Oracle Server and Set Operators

When a query uses set operators, the Oracle server eliminates duplicate rows automatically except in the case of the **UNION ALL** operator. The column names in the output are decided by the column list in the first **SELECT** statement. By default, the output is sorted in ascending order of the first column of the **SELECT** clause.

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

- If both queries select values of **CHAR** data type, of equal length, then the returned values have the **CHAR** data type of that length. If the queries select values of **CHAR** with different lengths, then the returned value is **VARCHAR2** with the length of the larger **CHAR** value.
- If either or both of the queries select values of **VARCHAR2** data type, then the returned values have the **VARCHAR2** data type.

If component queries select numeric data, then the data type of the return values is determined by numeric precedence. If all queries select values of the **NUMBER** type, then the returned values have the **NUMBER** data type. In queries using set operators, the Oracle server does not perform implicit conversion across data type groups. Therefore, if the corresponding expressions of component queries resolve to both character data and numeric data, the Oracle server returns an error.
Lesson Agenda

• Set Operators: Types and guidelines
• Tables used in this lesson
• \texttt{UNION} and \texttt{UNION ALL} operator
• \texttt{INTERSECT} operator
• \texttt{MINUS} operator
• Matching the \texttt{SELECT} statements
• Using the \texttt{ORDER BY} clause in set operations
Tables Used in This Lesson

The tables used in this lesson are:

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

Tables Used in This Lesson

Two tables are used in this lesson. They are the **EMPLOYEES** table and the **JOB_HISTORY** table. You are already familiar with the **EMPLOYEES** table that stores employee details such as a unique identification number, email address, job identification (such as **ST_CLERK**, **SA_REP**, and so on), salary, manager and so on.

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_id** (such as **ST_CLERK**, **SA_REP**, and so on), and the department are recorded in the **JOB_HISTORY** table.

The structure and data from the **EMPLOYEES** and **JOB_HISTORY** tables are shown on the following pages.
### Tables Used in This Lesson (continued)

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.

```
DESCRIBE employees
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Null</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPLOYEE_ID</td>
<td>NOT NULL</td>
<td>NUMBER(6)</td>
</tr>
<tr>
<td>FIRST_NAME</td>
<td></td>
<td>VARCHAR2(20)</td>
</tr>
<tr>
<td>LAST_NAME</td>
<td>NOT NULL</td>
<td>VARCHAR2(25)</td>
</tr>
<tr>
<td>EMAIL</td>
<td>NOT NULL</td>
<td>VARCHAR2(25)</td>
</tr>
<tr>
<td>PHONE_NUMBER</td>
<td></td>
<td>VARCHAR2(20)</td>
</tr>
<tr>
<td>HIRE_DATE</td>
<td>NOT NULL</td>
<td>DATE</td>
</tr>
<tr>
<td>JOB_ID</td>
<td>NOT NULL</td>
<td>VARCHAR2(10)</td>
</tr>
<tr>
<td>SALARY</td>
<td></td>
<td>NUMBER(8,2)</td>
</tr>
<tr>
<td>COMMISSION_PCT</td>
<td></td>
<td>NUMBER(2,2)</td>
</tr>
<tr>
<td>MANAGER_ID</td>
<td></td>
<td>NUMBER(6)</td>
</tr>
<tr>
<td>DEPARTMENT_ID</td>
<td></td>
<td>NUMBER(4)</td>
</tr>
</tbody>
</table>
Tables Used in This Lesson (continued)

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>1</td>
<td>King</td>
<td>AD_PRES</td>
<td>17-JUN-87</td>
<td>90</td>
</tr>
<tr>
<td>2</td>
<td>Kochhar</td>
<td>AD_VP</td>
<td>21-SEP-89</td>
<td>90</td>
</tr>
<tr>
<td>3</td>
<td>De Haan</td>
<td>AD_VP</td>
<td>13-JAN-93</td>
<td>90</td>
</tr>
<tr>
<td>4</td>
<td>Hunold</td>
<td>IT_PROG</td>
<td>03-JAN-90</td>
<td>60</td>
</tr>
<tr>
<td>5</td>
<td>Ernst</td>
<td>IT_PROG</td>
<td>21-MAY-91</td>
<td>60</td>
</tr>
<tr>
<td>6</td>
<td>Lorentz</td>
<td>IT_PROG</td>
<td>07-FEB-99</td>
<td>60</td>
</tr>
<tr>
<td>7</td>
<td>Mourgos</td>
<td>ST_MAN</td>
<td>16-NOV-99</td>
<td>50</td>
</tr>
<tr>
<td>8</td>
<td>Rajs</td>
<td>ST_CLERK</td>
<td>17-OCT-95</td>
<td>50</td>
</tr>
<tr>
<td>9</td>
<td>Davies</td>
<td>ST_CLERK</td>
<td>29-JAN-97</td>
<td>50</td>
</tr>
<tr>
<td>10</td>
<td>Matos</td>
<td>ST_CLERK</td>
<td>15-MAR-98</td>
<td>50</td>
</tr>
<tr>
<td>11</td>
<td>Vargas</td>
<td>ST_CLERK</td>
<td>09-JUL-96</td>
<td>50</td>
</tr>
<tr>
<td>12</td>
<td>Zlotkey</td>
<td>SA_MAN</td>
<td>29-JAN-00</td>
<td>80</td>
</tr>
<tr>
<td>13</td>
<td>Abel</td>
<td>SA_REP</td>
<td>11-MAY-96</td>
<td>80</td>
</tr>
<tr>
<td>14</td>
<td>Taylor</td>
<td>SA_REP</td>
<td>24-MAR-98</td>
<td>80</td>
</tr>
<tr>
<td>15</td>
<td>Grant</td>
<td>SA_REP</td>
<td>24-MAY-99</td>
<td>(null)</td>
</tr>
<tr>
<td>16</td>
<td>Whalen</td>
<td>AD_ASST</td>
<td>17-SEP-87</td>
<td>10</td>
</tr>
<tr>
<td>17</td>
<td>Hartstein</td>
<td>MK_MAN</td>
<td>17-FEB-96</td>
<td>20</td>
</tr>
</tbody>
</table>

DESCRIBE job_history

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>
### Tables Used in This Lesson (continued)

```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>1</td>
<td>102 13-JAN-93</td>
<td>24-JUL-98</td>
<td>IT_PROG</td>
<td>60</td>
</tr>
<tr>
<td>2</td>
<td>101 21-SEP-89</td>
<td>27-OCT-93</td>
<td>AC_ACCOUNT</td>
<td>110</td>
</tr>
<tr>
<td>3</td>
<td>101 28-OCT-93</td>
<td>15-MAR-97</td>
<td>AC_MGR</td>
<td>110</td>
</tr>
<tr>
<td>4</td>
<td>201 17-FEB-96</td>
<td>19-DEC-99</td>
<td>MK_REP</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>114 24-MAR-98</td>
<td>31-DEC-99</td>
<td>ST_CLERK</td>
<td>50</td>
</tr>
<tr>
<td>6</td>
<td>122 01-JAN-99</td>
<td>31-DEC-99</td>
<td>ST_CLERK</td>
<td>50</td>
</tr>
<tr>
<td>7</td>
<td>200 17-SEP-87</td>
<td>17-JUN-93</td>
<td>AD_ASST</td>
<td>90</td>
</tr>
<tr>
<td>8</td>
<td>176 24-MAR-98</td>
<td>31-DEC-98</td>
<td>SA_REP</td>
<td>80</td>
</tr>
<tr>
<td>9</td>
<td>176 01-JAN-99</td>
<td>31-DEC-99</td>
<td>SA_MAN</td>
<td>80</td>
</tr>
<tr>
<td>10</td>
<td>200 01-JUL-94</td>
<td>31-DEC-98</td>
<td>AC_ACCOUNT</td>
<td>90</td>
</tr>
</tbody>
</table>
Lesson Agenda

• Set Operators: Types and guidelines
• Tables used in this lesson
• **UNION** and **UNION ALL** operator
• **INTERSECT** operator
• **MINUS** operator
• Matching the **SELECT** statements
• Using the **ORDER BY** clause in set operations
UNION Operator

The **UNION** operator returns all rows that are 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 being selected must be the same.
- The data types of the columns being selected must be in the same data type group (such as numeric or character).
- 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.
- By default, the output is sorted in ascending order of the columns of the **SELECT** clause.
Using the **UNION** Operator

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

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

---

Using the **UNION** Operator

The **UNION** operator eliminates any duplicate records. If records that occur in both the `employees` and the `job_history` tables are identical, the records are displayed only once. Observe in the output shown in the slide that the record for the employee with the `employee_id` 200 appears twice because the `job_id` is different in each row.

Consider the following example:

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

---

Oracle Database 11g: SQL Fundamentals I 8 - 14
Using the **UNION** Operator (continued)

In the preceding output, employee 200 appears three times. Why? Note 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 to be a duplicate. Observe that the output is sorted in ascending order of the first column of the `SELECT` clause (in this case, `EMPLOYEE_ID`).
UNION ALL Operator

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

Guidelines

The guidelines for `UNION` and `UNION ALL` are the same, with the following two exceptions that pertain to `UNION ALL`: Unlike `UNION`, duplicate rows are not eliminated and the output is not sorted by default.
Using the **UNION ALL** Operator

Display the current and previous departments of all employees.

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

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>JOB_ID</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>AC, PRES</td>
</tr>
<tr>
<td>16</td>
<td>144</td>
<td>ST, CLERK</td>
</tr>
<tr>
<td>17</td>
<td>149</td>
<td>SA, MAN</td>
</tr>
<tr>
<td>18</td>
<td>174</td>
<td>SA, REP</td>
</tr>
<tr>
<td>19</td>
<td>175</td>
<td>SA, REP</td>
</tr>
<tr>
<td>20</td>
<td>175</td>
<td>SA, MAN</td>
</tr>
<tr>
<td>21</td>
<td>175</td>
<td>SA, REP</td>
</tr>
<tr>
<td>22</td>
<td>178</td>
<td>SA, REP</td>
</tr>
<tr>
<td>30</td>
<td>206</td>
<td>AC, ACCOUNT</td>
</tr>
</tbody>
</table>

Using the **UNION ALL** Operator

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

```sql
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 (because it is a duplicate):

<table>
<thead>
<tr>
<th>JOB_ID</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>176</td>
<td>SA, REP</td>
</tr>
</tbody>
</table>
Lesson Agenda

• Set Operators: Types and guidelines
• Tables used in this lesson
• UNION and UNION ALL operator
• INTERSECT operator
• MINUS operator
• Matching the SELECT statements
• Using ORDER BY clause in set operations
The **INTERSECT** operator returns rows that are common to both queries.

**INTERSECT Operator**

Use the **INTERSECT** operator to return all rows that are 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, however, 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 those employees who currently have a job title that is the same as their previous one (that is, they changed jobs but have now gone back to doing the same job they did previously).

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

Using the INTERSECT Operator

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

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

Example:

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

Employee 200 is no longer part of the results because the EMPLOYEES.DEPARTMENT_ID value is different from the JOB_HISTORY.DEPARTMENT_ID value.
Lesson Agenda

- Set Operators: Types and guidelines
- Tables used in this lesson
- UNION and UNION ALL operator
- INTERSECT operator
- MINUS operator
- Matching the SELECT statements
- Using the ORDER BY clause in set operations
The **MINUS** operator returns all the distinct rows selected by the first query, but not present in the second query result set.

**MINUS Operator**

Use the **MINUS** operator to return all distinct rows selected by the first query, but not present in the second query result set (the first `SELECT` statement **MINUS** the second `SELECT` statement).

**Note:** The number of columns must be the same and the data types of the columns being selected by the `SELECT` statements in the queries must belong to the same data type group in all the `SELECT` statements used in the query. The names of the columns, however, need not be identical.
Using 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;
```

Using the **MINUS** Operator

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.
Lesson Agenda

• Set Operators: Types and guidelines
• Tables used in this lesson
• UNION and UNION ALL operator
• INTERSECT operator
• MINUS operator
• Matching the SELECT statements
• Using ORDER BY clause in set operations
Matching the SELECT Statements

• Using the `UNION` operator, display the location ID, department name, and the state where it is located.
• You must match the data type (using the `TO_CHAR` function or any other conversion functions) when columns do not exist in one or the other table.

```sql
SELECT location_id, department_name "Department",
       TO_CHAR(NULL) "Warehouse location"
FROM departments
UNION
SELECT location_id, TO_CHAR(NULL) "Department",
       state_province
FROM locations;
```

Matching the SELECT Statements

Because the expressions in the SELECT lists of the queries must match in number, you can use the dummy columns and the data type conversion functions to comply with this rule. In the slide, the name, Warehouse location, is given as the dummy column heading. The `TO_CHAR` function is used in the first query to match the VARCHAR2 data type of the `state_province` column that is retrieved by the second query. Similarly, the `TO_CHAR` function in the second query is used to match the VARCHAR2 data type of the `department_name` column that is retrieved by the first query.

The output of the query is shown:

<table>
<thead>
<tr>
<th>LOCATION_ID</th>
<th>Department</th>
<th>Warehouse location</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1400 IT</td>
<td>(null)</td>
</tr>
<tr>
<td>2</td>
<td>1400 (null)</td>
<td>Texas</td>
</tr>
<tr>
<td>3</td>
<td>1500 Shipping</td>
<td>(null)</td>
</tr>
<tr>
<td>4</td>
<td>1500 (null)</td>
<td>California</td>
</tr>
<tr>
<td>5</td>
<td>1700 Accounting</td>
<td>(null)</td>
</tr>
<tr>
<td>6</td>
<td>1700 Administration</td>
<td>(null)</td>
</tr>
<tr>
<td>7</td>
<td>1700 Contracting</td>
<td>(null)</td>
</tr>
<tr>
<td>8</td>
<td>1700 Executive</td>
<td>(null)</td>
</tr>
<tr>
<td>9</td>
<td>1700 (null)</td>
<td>Washington</td>
</tr>
</tbody>
</table>

...
Matching the `SELECT` Statement: Example

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

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

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>JOB_ID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>24000</td>
</tr>
<tr>
<td>2</td>
<td>101</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>101</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>101</td>
<td>17000</td>
</tr>
<tr>
<td>5</td>
<td>102</td>
<td>17000</td>
</tr>
<tr>
<td>29</td>
<td>205</td>
<td>12000</td>
</tr>
<tr>
<td>30</td>
<td>205</td>
<td>8300</td>
</tr>
</tbody>
</table>

Matching the `SELECT` Statement: Example

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

The code example in the slide matches the `EMPLOYEE_ID` and `JOB_ID` columns in the `EMPLOYEES` and `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 results shown in the slide, each row in the output that corresponds to a record from the `JOB_HISTORY` table contains a 0 in the `SALARY` column.
Lesson Agenda

• Set Operators: Types and guidelines
• Tables used in this lesson
• UNION and UNION ALL operator
• INTERSECT operator
• MINUS operator
• Match the SELECT statements
• Using the ORDER BY clause in set operations
Using the ORDER BY Clause in Set Operations

- The **ORDER BY** clause can appear only once at the end of the compound query.
- Component queries cannot have individual **ORDER BY** clauses.
- **ORDER BY** clause recognizes only the columns of the first **SELECT** query.
- By default, the first column of the first **SELECT** query is used to sort the output in an ascending order.

**Using the ORDER BY Clause in Set Operations**

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 or an alias. By default, the output is sorted in ascending order in the first column of the first **SELECT** query.

**Note:** The **ORDER BY** clause does not recognize the column names of the second **SELECT** query. To avoid confusion over column names, it is a common practice to **ORDER BY** column positions.

For example, in the following statement, the output will be shown in ascending order of the **job_id**.

```sql
SELECT employee_id, job_id, salary
FROM employees
UNION
SELECT employee_id, job_id, 0
FROM job_history
ORDER BY 2;
```

If you omit the **ORDER BY**, then by default the output will be sorted in the ascending order of **employee_id**. You cannot use the columns from the second query to sort the output.
Summary

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

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

Summary

- The **UNION** operator returns all the distinct rows selected by each query in the compound 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 the case 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 that are 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 8: Overview

In this practice, you create reports by using:

- The UNION operator
- The INTERSECTION operator
- The MINUS operator
Practice 8

1. The HR department needs a list of department IDs for departments that do not contain the job ID ST_CLERK. Use the set operators to create this report.

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

2. The HR department needs a list of countries that have no departments located in them. Display the country ID and the name of the countries. Use the set operators to create this report.

<table>
<thead>
<tr>
<th>COUNTRY_ID</th>
<th>COUNTRY_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DE Germany</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>JOB_ID</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
</tr>
<tr>
<td>3</td>
<td>50</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
</tr>
</tbody>
</table>

4. Create a report that lists the employee IDs and job IDs of those employees who currently have a job title that is the same as their job title when they were initially hired by the company (that is, they changed jobs but have now gone back to doing their original job).

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>JOB_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>176</td>
</tr>
<tr>
<td>2</td>
<td>200</td>
</tr>
</tbody>
</table>
Practice 8 (continued)

5. The HR department needs a report with the following specifications:
   • Last name and department ID of all employees from the EMPLOYEES table, regardless of whether or not they belong to a department
   • Department ID and department name of all departments from the DEPARTMENTS table, regardless of whether or not they have employees working in them

Write a compound query to accomplish this.

<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>(null)</td>
</tr>
<tr>
<td>Davies</td>
<td>50</td>
<td>(null)</td>
</tr>
<tr>
<td>De Haan</td>
<td>90</td>
<td>(null)</td>
</tr>
<tr>
<td>Ernst</td>
<td>60</td>
<td>(null)</td>
</tr>
<tr>
<td>Fay</td>
<td>20</td>
<td>(null)</td>
</tr>
<tr>
<td>Gietz</td>
<td>110</td>
<td>(null)</td>
</tr>
<tr>
<td>Grant</td>
<td>(null)</td>
<td>(null)</td>
</tr>
<tr>
<td>Hartstein</td>
<td>20</td>
<td>(null)</td>
</tr>
<tr>
<td>Higgins</td>
<td>110</td>
<td>(null)</td>
</tr>
<tr>
<td>Hunold</td>
<td>60</td>
<td>(null)</td>
</tr>
<tr>
<td>King</td>
<td>90</td>
<td>(null)</td>
</tr>
<tr>
<td>Kochhar</td>
<td>90</td>
<td>(null)</td>
</tr>
<tr>
<td>Lorentz</td>
<td>60</td>
<td>(null)</td>
</tr>
<tr>
<td>Matos</td>
<td>50</td>
<td>(null)</td>
</tr>
<tr>
<td>Mourgos</td>
<td>50</td>
<td>(null)</td>
</tr>
<tr>
<td>Rajs</td>
<td>50</td>
<td>(null)</td>
</tr>
<tr>
<td>Taylor</td>
<td>80</td>
<td>(null)</td>
</tr>
<tr>
<td>Vargas</td>
<td>50</td>
<td>(null)</td>
</tr>
<tr>
<td>Whalen</td>
<td>10</td>
<td>(null)</td>
</tr>
<tr>
<td>Zlotkey</td>
<td>80</td>
<td>(null)</td>
</tr>
<tr>
<td>(null)</td>
<td>10 Administration</td>
<td></td>
</tr>
<tr>
<td>(null)</td>
<td>20 Marketing</td>
<td></td>
</tr>
<tr>
<td>(null)</td>
<td>50 Shipping</td>
<td></td>
</tr>
<tr>
<td>(null)</td>
<td>60 IT</td>
<td></td>
</tr>
<tr>
<td>(null)</td>
<td>80 Sales</td>
<td></td>
</tr>
<tr>
<td>(null)</td>
<td>90 Executive</td>
<td></td>
</tr>
<tr>
<td>(null)</td>
<td>110 Accounting</td>
<td></td>
</tr>
<tr>
<td>(null)</td>
<td>190 Contracting</td>
<td></td>
</tr>
</tbody>
</table>
Manipulating Data
Objectives

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

- Describe each data manipulation language (DML) statement
- Insert rows into a table
- Update rows in a table
- Delete rows from a table
- Control transactions

Objective

In this lesson, you learn how to use the data manipulation language (DML) statements to insert rows into a table, update existing rows in a table, and delete existing rows from a table. You also learn how to control transactions with the COMMIT, SAVEPOINT, and ROLLBACK statements.
Lesson Agenda

- Adding new rows in a table
  - `INSERT` statement
- Changing data in a table
  - `UPDATE` statement
- Removing rows from a table:
  - `DELETE` statement
  - `TRUNCATE` statement
- Database transactions control using `COMMIT`, `ROLLBACK`, and `SAVEPOINT`
- Read consistency
- `FOR UPDATE` clause in a `SELECT` statement
Data Manipulation Language

- A DML statement is executed when you:
  - Add new rows to a table
  - Modify existing rows in a table
  - Remove existing rows from a table
- A transaction consists of a collection of DML statements that form a logical unit of work.

Data Manipulation Language

Data manipulation language (DML) is a core part of SQL. When you want to add, update, or delete data in the database, you execute a DML statement. A collection of DML statements that form a logical unit of work is called a transaction.

Consider a banking database. When a bank customer transfers money from a savings account to a checking account, the transaction might consist of three separate operations: decreasing the savings account, increasing the checking account, and recording the transaction in the transaction journal. The Oracle server must guarantee that all the three SQL statements are performed to maintain the accounts in proper balance. When something prevents one of the statements in the transaction from executing, the other statements of the transaction must be undone.

Note: Most of the DML statements in this lesson assume that no constraints on the table are violated. Constraints are discussed later in this course.

Note: In SQL Developer, click the Run Script icon or press [F5] to run the DML statements. The feedback messages will be shown on the Script Output tabbed page.
Adding a New Row to a Table

The graphic in the slide illustrates the addition of a new department to the `DEPARTMENTS` table.
**INSERT Statement Syntax**

- Add new rows to a table by using the `INSERT` statement:

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

- With this syntax, only one row is inserted at a time.

**Note:** This statement with the `VALUES` clause adds only one row at a time to a table.
Inserting New Rows

• Insert a new row containing values for each column.
• List values in the default order of the columns in the table.
• Optionally, list the columns in the `INSERT` clause.

```sql
INSERT INTO departments(department_id, department_name, manager_id, location_id)
VALUES (70, 'Public Relations', 100, 1700);
```

• Enclose character and date values within single quotation marks.

Inserting New Rows

Because you can insert a new row that contains values for each column, the column list is not required in the `INSERT` clause. However, if you do not use the column list, the values must be listed according to the default order of the columns in the table, and a value must be provided for each column.

```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>

For clarity, use the column list in the `INSERT` clause. Enclose character and date values within single quotation marks; however, it is not recommended that you enclose numeric values within single quotation marks.
Inserting Rows with Null Values

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

```
INSERT INTO departments (department_id, department_name)
VALUES (30, 'Purchasing');
```

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

```
INSERT INTO departments
VALUES (100, 'Finance', NULL, NULL);
```

### Method Description

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implicit</td>
<td>Omit the column from the column list.</td>
</tr>
<tr>
<td>Explicit</td>
<td>Specify the NULL keyword in the VALUES list; specify the empty string (' ') in the VALUES list for character strings and dates.</td>
</tr>
</tbody>
</table>

Be sure that you can use null values in the targeted column by verifying the Null status with the DESCRIBE command.

The Oracle server automatically enforces all data types, data ranges, and data integrity constraints. Any column that is not listed explicitly obtains a null value in the new row.

Common errors that can occur during user input are checked in the following order:

- Mandatory value missing for a NOT NULL column
- Duplicate value violating any unique or primary key constraint
- Any value violating a CHECK constraint
- Referential integrity maintained for foreign key constraint
- Data type mismatches or values too wide to fit in column

**Note:** Use of the column list is recommended as it makes the INSERT statement more readable and reliable, or less prone to mistakes.
Inserting Special Values

The `SYSDATE` function records the current date and time.

```
INSERT INTO employees (employee_id, first_name, last_name, email, phone_number, hire_date, job_id, salary, commission_pct, manager_id, department_id)
VALUES (113, 'Louis', 'Popp', 'LPOPP', '515.124.4567', SYSDATE, 'AC_ACCOUNT', 6900, NULL, 205, 110);
```

Inserting Special Values

You can use functions to enter special values in your table.

The slide example records information for employee Popp in the `EMPLOYEES` table. It supplies the current date and time in the `HIRE_DATE` column. It uses the `SYSDATE` function that returns the current date and time of the database server. You may also use the `CURRENT_DATE` function to get the current date in the session time zone. You can also use the `USER` function when inserting rows in a table. The `USER` function records the current username.

Confirming Additions to the Table

```
SELECT employee_id, last_name, job_id, hire_date, commission_pct
FROM employees
WHERE employee_id = 113;
```
Inserting Specific Date and Time Values

• Add a new employee.

```
INSERT INTO employees
VALUES      (114,
              'Den', 'Raphealy',
              'DRAPHEAL', '515.127.4561',
              TO_DATE('FEB 3, 1999', 'MON DD, YYYY'),
              'SA_REP', 11000, 0.2, 100, 60);
```

• Verify your addition.

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>EMAIL</th>
<th>PHONE_NUMBER</th>
<th>HIRE_DATE</th>
<th>JOB_ID</th>
<th>SALARY</th>
<th>COMMISSION_PCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>114</td>
<td>Den</td>
<td>Raphealy</td>
<td>DRAPHEAL 515.127.4561</td>
<td>03-FEB-99</td>
<td>SA_REP</td>
<td>1/000</td>
<td>0.2</td>
<td></td>
</tr>
</tbody>
</table>

Inserting Specific Date and Time Values

The DD-MON-RR format is generally used to insert a date value. With the RR format, the system provides the correct century automatically.

You may also supply the date value in the DD-MON-YYYY format. This is recommended because it clearly specifies the century and does not depend on the internal RR format logic of specifying the correct century.

If a date must be entered in a format other than the default format (for example, with another century or a specific time), you must use the TO_DATE function.

The example in the slide records information for employee Raphealy in the EMPLOYEES table. It sets the HIRE_DATE column to be February 3, 1999.
Creating a Script

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

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

Creating a Script

You can save commands with substitution variables to a file and execute the commands in the file. The example in the slide records information for a department in the `DEPARTMENTS` table.

Run the script file and you are prompted for input for each of the ampersand (&) substitution variables. After entering a value for the substitution variable, click the OK button. The values that you input are then substituted into the statement. This enables you to run the same script file over and over, but supply a different set of values each time you run it.
Copying Rows from Another Table

- Write your INSERT statement with a subquery:

```sql
INSERT INTO sales_reps(id, name, salary, commission_pct)
SELECT employee_id, last_name, salary, commission_pct
FROM employees
WHERE job_id LIKE '%REP%';
```

- Do not use the VALUES clause.
- Match the number of columns in the INSERT clause to those in the subquery.
- Inserts all the rows returned by the subquery in the table, `sales_reps`.

Copying Rows from Another Table

You can use the INSERT statement to add rows to a table where the values are derived from existing tables. In the slide example, for the INSERT INTO statement to work, you must have already created the `sales_reps` table using the CREATE TABLE statement. CREATE TABLE is discussed in the next lesson titled “Using DDL Statements to Create and Manage Tables.”

In place of the VALUES clause, you use a subquery.

**Syntax**

```
INSERT INTO table [ column (, column) ] subquery;
```

In the syntax:
- `table` is the name of the table
- `column` is the name of the column in the table to populate
- `subquery` is the subquery that returns rows to the table

The number of columns and their data types in the column list of the INSERT clause must match the number of values and their data types in the subquery. Zero or more rows are added depending on the number of rows returned by the subquery. To create a copy of the rows of a table, use SELECT * in the subquery:

```
INSERT INTO copy_emp
SELECT *
FROM employees;
```
Lesson Agenda

• Adding new rows in a table
  – INSERT statement

• Changing data in a table
  – UPDATE statement

• Removing rows from a table:
  – DELETE statement
  – TRUNCATE statement

• Database transactions control using COMMIT, ROLLBACK, and SAVEPOINT

• Read consistency

• FOR UPDATE clause in a SELECT statement
Changing Data in a Table

The slide illustrates changing the department number for employees in department 60 to department 80.
UPDATE Statement Syntax

- Modify existing values in a table with the `UPDATE` statement:

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

- Update more than one row at a time (if required).

**UPDATE Statement Syntax**

You can modify the existing values in a table by using the `UPDATE` statement.

In the syntax:

- `table` is the name of the table
- `column` is the name of the column in the table to populate
- `value` is the corresponding value or subquery for the column
- `condition` identifies the rows to be updated and is composed of column names, expressions, constants, subqueries, and comparison operators

Confirm the update operation by querying the table to display the updated rows.

For more information, see the section on “UPDATE” in the *Oracle Database SQL Language Reference 11g, Release 1 (11.1)*.

**Note:** In general, use the primary key column in the `WHERE` clause to identify a single row for update. Using other columns can unexpectedly cause several rows to be updated. For example, identifying a single row in the `EMPLOYEES` table by name is dangerous, because more than one employee may have the same name.
Updating Rows in a Table

- Values for a specific row or rows are modified if you specify the `WHERE` clause:

```
UPDATE employees
SET department_id = 50
WHERE employee_id = 113;
1 row updated
```

- Values for all the rows in the table are modified if you omit the `WHERE` clause:

```
UPDATE copy_emp
SET department_id = 110;
2 rows updated
```

- Specify `SET column_name= NULL` to update a column value to `NULL`.

For example, an employee who was a `SA_REP` has now changed his job to an `IT_PROG`. Therefore, his `JOB_ID` needs to be updated and the commission field needs to be set to `NULL`.

```
UPDATE employees
SET job_id = 'IT_PROG', commission_pct = NULL
WHERE employee_id = 114;
```

**Note:** The `COPY_EMP` table has the same data as the `EMPLOYEES` table.
Updating Two Columns with a Subquery

Update employee 113’s job and salary to match those of employee 205.

```sql
UPDATE employees
SET job_id = (SELECT job_id
              FROM employees
              WHERE employee_id = 205),
    salary = (SELECT salary
               FROM employees
               WHERE employee_id = 205)
WHERE employee_id = 113;
```

Updating Two Columns with a Subquery

You can update multiple columns in the SET clause of an UPDATE statement by writing multiple subqueries. The syntax is as follows:

```
UPDATE table
SET column = (SELECT column
              FROM table
              WHERE condition)
[ ,
 column = (SELECT column
            FROM table
            WHERE condition)]
[WHERE condition ] ;
```

The example in the slide can also be written as follows:

```
UPDATE employees
SET (job_id, salary) = (SELECT job_id, salary
                         FROM employees
                         WHERE employee_id = 205)
WHERE employee_id = 113;
```
Updating Rows Based on Another Table

You can use the subqueries in the UPDATE statements to update values in a table. The example in the slide updates the COPY_EMP table based on the values from the EMPLOYEES table. It changes the department number of all employees with employee 200’s job ID to employee 100’s current department number.
Lesson Agenda

- Adding new rows in a table
  - INSERT statement
- Changing data in a table
  - UPDATE statement
- Removing rows from a table:
  - DELETE statement
  - TRUNCATE statement
- Database transactions control using COMMIT, ROLLBACK, and SAVEPOINT
- Read consistency
- FOR UPDATE clause in a SELECT statement
Removing a Row from a Table

The Contracting department has been removed from the DEPARTMENTS table (assuming no constraints on the DEPARTMENTS table are violated), as shown by the graphic in the slide.
DELETE Statement

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

```
DELETE [FROM] table 
[WHERE condition];
```

DELETE Statement Syntax

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

In the syntax:
- `table` is the name of the table
- `condition` identifies the rows to be deleted, and is composed of column names, expressions, constants, subqueries, and comparison operators

**Note:** If no rows are deleted, the message “0 rows deleted” is returned (in the Script Output tab in SQL Developer)

For more information, see the section on “DELETE” in *Oracle Database SQL Language Reference 11g, Release 1 (11.1).*
Deleting Rows from a Table

- Specific rows are deleted if you specify the `WHERE` clause:

```sql
DELETE FROM departments
WHERE department_name = 'Finance';
```

- All rows in the table are deleted if you omit the `WHERE` clause:

```sql
DELETE FROM copy_emp;
```

Deleting Rows from a Table

You can delete specific rows by specifying the `WHERE` clause in the `DELETE` statement. The first example in the slide deletes the Accounting department from the `DEPARTMENTS` table. You can confirm the delete operation by displaying the deleted rows using the `SELECT` statement.

```sql
SELECT *
FROM departments
WHERE department_name = 'Finance';
```

However, if you omit the `WHERE` clause, all rows in the table are deleted. The second example in the slide deletes all rows from the `COPY_EMP` table, because no `WHERE` clause was specified.

**Example:**

Remove rows identified in the `WHERE` clause.

```sql
DELETE FROM employees WHERE employee_id = 114;
DELETE FROM departments WHERE department_id IN (30, 40);
```
Deleting Rows Based on Another Table

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

```sql
DELETE FROM employees
WHERE department_id =
(SELECT department_id
 FROM departments
WHERE department_name LIKE '%Public%');
```

Deleting Rows Based on Another Table

You can use the subqueries to delete rows from a table based on values from another table. The example in the slide deletes all employees in a department where the department name contains the string `Admin`. The subquery searches the `DEPARTMENTS` table to find the department number based on the department name containing the string `Public`. The subquery then feeds the department number to the main query, which deletes rows of data from the `EMPLOYEES` table based on this department number.
TRUNCATE Statement

- Removes all rows from a table, leaving the table empty and the table structure intact
- Is a data definition language (DDL) statement rather than a DML statement; cannot easily be undone
- Syntax:
  
  ```sql
  TRUNCATE TABLE table_name;
  ```

- Example:
  
  ```sql
  TRUNCATE TABLE copy_emp;
  ```

TRUNCATE Statement

A more efficient method of emptying a table is by using the TRUNCATE statement. You can use the TRUNCATE statement to quickly remove all rows from a table or cluster. Removing rows with the TRUNCATE statement is faster than removing them with the DELETE statement for the following reasons:

- The TRUNCATE statement is a data definition language (DDL) statement and generates no rollback information. Rollback information is covered later in this lesson.
- Truncating a table does not fire the delete triggers of the table.

If the table is the parent of a referential integrity constraint, you cannot truncate the table. You need to disable the constraint before issuing the TRUNCATE statement. Disabling constraints is covered in a subsequent lesson.
Lesson Agenda

• Adding new rows in a table
  – INSERT statement
• Changing data in a table
  – UPDATE statement
• Removing rows from a table:
  – DELETE statement
  – TRUNCATE statement
• Database transactions control using COMMIT, ROLLBACK, and SAVEPOINT
• Read consistency
• FOR UPDATE clause in a SELECT statement
Database Transactions

A database transaction consists of one of the following:

- DML statements that constitute one consistent change to the data
- One DDL statement
- One data control language (DCL) statement

Database Transactions

The Oracle server ensures data consistency based on transactions. Transactions give you more flexibility and control when changing data, and they ensure data consistency in the event of user process failure or system failure.

Transactions consist of DML statements that constitute one consistent change to the data. For example, a transfer of funds between two accounts should include the debit in one account and the credit to another account of the same amount. Both actions should either fail or succeed together; the credit should not be committed without the debit.

Transaction Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data manipulation language (DML)</td>
<td>Consists of any number of DML statements that the Oracle server treats as a single entity or a logical unit of work</td>
</tr>
<tr>
<td>Data definition language (DDL)</td>
<td>Consists of only one DDL statement</td>
</tr>
<tr>
<td>Data control language (DCL)</td>
<td>Consists of only one DCL statement</td>
</tr>
</tbody>
</table>
Database Transactions: Start and End

• Begin when the first DML SQL statement is executed.

• End with one of the following events:
  – A COMMIT or ROLLBACK statement is issued.
  – A DDL or DCL statement executes (automatic commit).
  – The user exits SQL Developer or SQL*Plus.
  – The system crashes.

Database Transaction: Start and End

When does a database transaction start and end?

A transaction begins when the first DML statement is encountered and ends when one of the following occurs:

• A COMMIT or ROLLBACK statement is issued.
• A DDL statement, such as CREATE, is issued.
• A DCL statement is issued.
• The user exits SQL Developer or SQL*Plus.
• A machine fails or the system crashes.

After one transaction ends, the next executable SQL statement automatically starts the next transaction.

A DDL statement or a DCL statement is automatically committed and therefore implicitly ends a transaction.
Advantages of COMMIT and ROLLBACK Statements

With COMMIT and ROLLBACK statements, you can:

• Ensure data consistency
• Preview data changes before making changes permanent
• Group logically-related operations

Advantages of COMMIT and ROLLBACK Statements

With the COMMIT and ROLLBACK statements, you have control over making changes to the data permanent.
Explicit Transaction Control Statements

You can control the logic of transactions by using the `COMMIT`, `SAVEPOINT`, and `ROLLBACK` statements.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMIT</td>
<td>Ends the current transaction by making all pending data changes permanent</td>
</tr>
<tr>
<td><code>SAVEPOINT name</code></td>
<td>Marks a savepoint within the current transaction</td>
</tr>
<tr>
<td><code>ROLLBACK</code></td>
<td><code>ROLLBACK</code> ends the current transaction by discarding all pending data changes.</td>
</tr>
<tr>
<td><code>ROLLBACK TO SAVEPOINT name</code></td>
<td><code>ROLLBACK TO SAVEPOINT</code> rolls back the current transaction to the specified savepoint, thereby discarding any changes and/or savepoints that were created after the savepoint to which you are rolling back. If you omit the <code>TO SAVEPOINT</code> clause, the <code>ROLLBACK</code> statement rolls back the entire transaction. Because savepoints are logical, there is no way to list the savepoints that you have created.</td>
</tr>
</tbody>
</table>

**Note:** You cannot `COMMIT` to a `SAVEPOINT`. `SAVEPOINT` is not ANSI-standard SQL.
**Rolling Back Changes to a Marker**

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

```sql
UPDATE...
SAVEPOINT update_done;
SAVEPOINT update_done succeeded.
INSERT...
ROLLBACK TO update_done;
ROLLBACK TO succeeded.
```

**Rolling Back Changes to a Marker**

You can create a marker in the current transaction by using the `SAVEPOINT` statement, which divides the transaction into smaller sections. You can then discard pending changes up to that marker by using the `ROLLBACK TO SAVEPOINT` statement.

Note, if you create a second savepoint with the same name as an earlier savepoint, the earlier savepoint is deleted.
 Implicit Transaction Processing

• An automatic commit occurs in the following circumstances:
  – A DDL statement is issued
  – A DCL statement is issued
  – Normal exit from SQL Developer or SQL*Plus, without explicitly issuing COMMIT or ROLLBACK statements

• An automatic rollback occurs when there is an abnormal termination of SQL Developer or SQL*Plus or a system failure.

### Status | Circumstances
--- | ---
Automatic commit | DDL statement or DCL statement issued
 | SQL Developer or SQL*Plus exited normally, without explicitly issuing COMMIT or ROLLBACK commands
Automatic rollback | Abnormal termination of SQL Developer or SQL*Plus or system failure

**Note:** In SQL*Plus, the AUTOCOMMIT command can be toggled ON or OFF. If set to ON, each individual DML statement is committed as soon as it is executed. You cannot roll back the changes. If set to OFF, the COMMIT statement can still be issued explicitly. Also, the COMMIT statement is issued when a DDL statement is issued or when you exit SQL*Plus. The SET AUTOCOMMIT ON/OFF command is skipped in SQL Developer. DML is committed on a normal exit from SQL Developer only if you have the Autocommit preference enabled. To enable Autocommit, perform the following:

• In the Tools menu, select Preferences. In the Preferences dialog box, expand Database and select Worksheet Parameters.
• On the right pane, check the Autocommit in SQL Worksheet option. Click OK.
Implicit Transaction Processing (continued)

System Failures

When a transaction is interrupted by a system failure, the entire transaction is automatically rolled back. This prevents the error from causing unwanted changes to the data and returns the tables to the state at the time of the last commit. In this way, the Oracle server protects the integrity of the tables.

In SQL Developer, a normal exit from the session is accomplished by selecting Exit from the File menu. In SQL*Plus, a normal exit is accomplished by entering the `EXIT` command at the prompt. Closing the window is interpreted as an abnormal exit.
State of the Data Before **COMMIT** or **ROLLBACK**

- The previous state of the data can be recovered.
- The current user can review the results of the DML operations by using the `SELECT` statement.
- Other users *cannot* view the results of the DML statements issued by the current user.
- The affected rows are *locked*; other users cannot change the data in the affected rows.

**State of the Data Before COMMIT or ROLLBACK**

Every data change made during the transaction is temporary until the transaction is committed. The state of the data before COMMIT or ROLLBACK statements are issued can be described as follows:

- Data manipulation operations primarily affect the database buffer; therefore, the previous state of the data can be recovered.
- The current user can review the results of the data manipulation operations by querying the tables.
- Other users cannot view the results of the data manipulation operations made by the current user. The Oracle server institutes read consistency to ensure that each user sees data as it existed at the last commit.
- The affected rows are locked; other users cannot change the data in the affected rows.
State of the Data After COMMIT

- Data changes are saved in the database.
- The previous state of the data is overwritten.
- All users can view the results.
- Locks on the affected rows are released; those rows are available for other users to manipulate.
- All savepoints are erased.

State of the Data After COMMIT

Make all pending changes permanent by using the COMMIT statement. Here is what happens after a COMMIT statement:

- Data changes are written to the database.
- The previous state of the data is no longer available with normal SQL queries.
- All users can view the results of the transaction.
- The locks on the affected rows are released; the rows are now available for other users to perform new data changes.
- All savepoints are erased.
Committing Data

• Make the changes:

```sql
DELETE FROM employees
WHERE employee_id = 99999;
1 rows deleted

INSERT INTO departments
VALUES (290, 'Corporate Tax', NULL, 1700);
1 rows inserted
```

• Commit the changes:

```sql
COMMIT;
COMMIT succeeded.
```

Committing Data

In the example in the slide, a row is deleted from the EMPLOYEES table and a new row is inserted into the DEPARTMENTS table. The changes are saved by issuing the COMMIT statement.

Example:

Remove departments 290 and 300 in the DEPARTMENTS table and update a row in the EMPLOYEES table. Save the data change.

```sql
DELETE FROM departments
WHERE department_id IN (290, 300);

UPDATE employees
SET department_id = 80
WHERE employee_id = 206;

COMMIT;
```
State of the Data After `ROLLBACK`

Discard all pending changes by using the `ROLLBACK` statement:

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

```sql
DELETE FROM copy_emp;
ROLLBACK;
```
State of the Data After **ROLLBACK**: Example

While attempting to remove a record from the **TEST** table, you may accidentally empty the table. However, you can correct the mistake, reissue a proper statement, and make the data change permanent.
Statement-Level Rollback

- If a single DML statement fails during execution, only that statement is rolled back.
- The Oracle server implements an implicit savepoint.
- All other changes are retained.
- The user should terminate transactions explicitly by executing a `COMMIT` or `ROLLBACK` statement.

Statement-Level Rollback

A part of a transaction can be discarded through an implicit rollback if a statement execution error is detected. If a single DML statement fails during execution of a transaction, its effect is undone by a statement-level rollback, but the changes made by the previous DML statements in the transaction are not discarded. They can be committed or rolled back explicitly by the user.

The Oracle server issues an implicit commit before and after any DDL statement. So, even if your DDL statement does not execute successfully, you cannot roll back the previous statement because the server issued a commit.

Terminate your transactions explicitly by executing a `COMMIT` or `ROLLBACK` statement.
Lesson Agenda

• Adding new rows in a table
  – INSERT statement

• Changing data in a table
  – UPDATE statement

• Removing rows from a table:
  – DELETE statement
  – TRUNCATE statement

• Database transactions control using COMMIT, ROLLBACK, and SAVEPOINT

• Read consistency

• FOR UPDATE clause in a SELECT statement
Read Consistency

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

Read Consistency

Database users access the database in two ways:
• Read operations (SELECT statement)
• Write operations (INSERT, UPDATE, DELETE statements)

You need read consistency so that the following occur:
• The database reader and writer are ensured a consistent view of the data.
• Readers do not view data that is in the process of being changed.
• Writers are ensured that the changes to the database are done in a consistent manner.
• Changes made by one writer do not disrupt or conflict with the changes being made by another writer.

The purpose of read consistency is to ensure that each user sees data as it existed at the last commit, before a DML operation started.

Note: The same user can login as different sessions. Each session maintains read consistency in the manner described above, even if they are the same users.
Implementing Read Consistency

Read consistency is an automatic implementation. It keeps a partial copy of the database in the undo segments. The read-consistent image is constructed from the committed data in the table and the old data that is being changed and is not yet committed from the undo segment.

When an insert, update, or delete operation is made on the database, the Oracle server takes a copy of the data before it is changed and writes it to a undo segment.

All readers, except the one who issued the change, see the database as it existed before the changes started; they view the undo segment’s “snapshot” of the data.

Before the changes are committed to the database, only the user who is modifying the data sees the database with the alterations. Everyone else sees the snapshot in the undo segment. This guarantees that readers of the data read consistent data that is not currently undergoing change.

When a DML statement is committed, the change made to the database becomes visible to anyone issuing a SELECT statement after the commit is done. The space occupied by the old data in the undo segment file is freed for reuse.

If the transaction is rolled back, the changes are undone:

- The original, older version of the data in the undo segment is written back to the table.
- All users see the database as it existed before the transaction began.
Lesson Agenda

• Adding new rows in a table
  – INSERT statement
• Changing data in a table
  – UPDATE statement
• Removing rows from a table:
  – DELETE statement
  – TRUNCATE statement
• Database transactions control using COMMIT, ROLLBACK, and SAVEPOINT
• Read consistency
• FOR UPDATE clause in a SELECT statement
FOR UPDATE Clause in a SELECT Statement

- Locks the rows in the EMPLOYEES table where job_id is SA_REP.

```
SELECT employee_id, salary, commission_pct, job_id
FROM employees
WHERE job_id = 'SA_REP'
FOR UPDATE
ORDER BY employee_id;
```

- Lock is released only when you issue a ROLLBACK or a COMMIT.
- If the SELECT statement attempts to lock a row that is locked by another user, then the database waits until the row is available, and then returns the results of the SELECT statement.

FOR UPDATE Clause in a SELECT Statement

When you issue a SELECT statement against the database to query some records, no locks are placed on the selected rows. In general, this is required because the number of records locked at any given time is (by default) kept to the absolute minimum: only those records that have been changed but not yet committed are locked. Even then, others will be able to read those records as they appeared before the change (the “before image” of the data). There are times, however, when you may want to lock a set of records even before you change them in your program. Oracle offers the FOR UPDATE clause of the SELECT statement to perform this locking.

When you issue a SELECT...FOR UPDATE statement, the relational database management system (RDBMS) automatically obtains exclusive row-level locks on all the rows identified by the SELECT statement, thereby holding the records “for your changes only.” No one else will be able to change any of these records until you perform a ROLLBACK or a COMMIT.

You can append the optional keyword NOWAIT to the FOR UPDATE clause to tell the Oracle server not to wait if the table has been locked by another user. In this case, control will be returned immediately to your program or to your SQL Developer environment so that you can perform other work, or simply wait for a period of time before trying again. Without the NOWAIT clause, your process will block until the table is available, when the locks are released by the other user through the issue of a COMMIT or a ROLLBACK command.
FOR UPDATE Clause: Examples

- You can use the FOR UPDATE clause in a SELECT statement against multiple tables.

```sql
SELECT e.employee_id, e.salary, e.commission_pct
FROM employees e JOIN departments d
USING (department_id)
WHERE job_id = 'ST_CLERK'
AND location_id = 1500
FOR UPDATE
ORDER BY e.employee_id;
```

- Rows from both the EMPLOYEES and DEPARTMENTS tables are locked.
- Use FOR UPDATE OF column_name to qualify the column you intend to change, then only the rows from that specific table are locked.

FOR UPDATE Clause: Examples

In the example in the slide, the statement locks rows in the EMPLOYEES table with JOB_ID set to ST_CLERK and LOCATION_ID set to 1500, and locks rows in the DEPARTMENTS table with departments in LOCATION_ID set as 1500.

You can use the FOR UPDATE OF column_name to qualify the column that you intend to change. The OF list of the FOR UPDATE clause does not restrict you to changing only those columns of the selected rows. Locks are still placed on all rows; if you simply state FOR UPDATE in the query and do not include one or more columns after the OF keyword, the database will lock all identified rows across all the tables listed in the FROM clause.

The following statement locks only those rows in the EMPLOYEES table with ST_CLERK located in LOCATION_ID 1500. No rows are locked in the DEPARTMENTS table:

```sql
SELECT e.employee_id, e.salary, e.commission_pct
FROM employees e JOIN departments d
USING (department_id)
WHERE job_id = 'ST_CLERK' AND location_id = 1500
FOR UPDATE OF e.salary
ORDER BY e.employee_id;
```
FOR UPDATE Clause: Examples (continued)

In the following example, the database is instructed to wait for five seconds for the row to become available, and then return control to you.

```
SELECT employee_id, salary, commission_pct, job_id
FROM employees
WHERE job_id = 'SA_REP'
FOR UPDATE WAIT 5
ORDER BY employee_id;
```
Summary

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSERT</td>
<td>Adds a new row to the table</td>
</tr>
<tr>
<td>UPDATE</td>
<td>Modifies existing rows in the table</td>
</tr>
<tr>
<td>DELETE</td>
<td>Removes existing rows from the table</td>
</tr>
<tr>
<td>TRUNCATE</td>
<td>Removes all rows from a table</td>
</tr>
<tr>
<td>COMMIT</td>
<td>Makes all pending changes permanent</td>
</tr>
<tr>
<td>SAVEPOINT</td>
<td>Is used to roll back to the savepoint marker</td>
</tr>
<tr>
<td>ROLLBACK</td>
<td>Discards all pending data changes</td>
</tr>
</tbody>
</table>

For the SELECT statement, you can use the FOR UPDATE clause to lock rows for your changes only.

Remember that the Oracle server guarantees a consistent view of data at all times.
Practice 9: Overview

This practice covers the following topics:
• Inserting rows into the tables
• Updating and deleting rows in the table
• Controlling transactions

Practice 9: Overview

In this practice, you add rows to the MY_EMPLOYEE table, update and delete data from the table, and control your transactions. You run a script to create the MY_EMPLOYEE table.
Practice 9

The HR department wants you to create SQL statements to insert, update, and delete employee data. As a prototype, you use the `MY_EMPLOYEE` table before giving the statements to the HR department.

**Note:** For all the DML statements, use the Run Script icon (or press [F5]) to execute the query. This way you get to see the feedback messages on the Script Output tab page. For SELECT queries, continue to use the Execute Statement icon or press [F9] to get the formatted output on the Results tab page.

Insert data into the `MY_EMPLOYEE` table.
1. Run the statement in the `lab_09_01.sql` script to build the `MY_EMPLOYEE` table used in this practice.
2. Describe the structure of the `MY_EMPLOYEE` table to identify the column names.

<table>
<thead>
<tr>
<th>Name</th>
<th>Null</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>NOT NULL</td>
<td>NUMBER(4)</td>
</tr>
<tr>
<td>LAST_NAME</td>
<td></td>
<td>VARCHAR2(25)</td>
</tr>
<tr>
<td>FIRST_NAME</td>
<td></td>
<td>VARCHAR2(25)</td>
</tr>
<tr>
<td>USERID</td>
<td></td>
<td>VARCHAR2(8)</td>
</tr>
<tr>
<td>SALARY</td>
<td></td>
<td>NUMBER(9,2)</td>
</tr>
</tbody>
</table>

3. Create an INSERT statement to 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. *Do not enter all rows yet.*

<table>
<thead>
<tr>
<th>ID</th>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>USERID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Patel</td>
<td>Ralph</td>
<td>rpatel</td>
<td>895</td>
</tr>
<tr>
<td>2</td>
<td>Dancs</td>
<td>Betty</td>
<td>bdancs</td>
<td>860</td>
</tr>
<tr>
<td>3</td>
<td>Biri</td>
<td>Ben</td>
<td>bbir</td>
<td>1100</td>
</tr>
<tr>
<td>4</td>
<td>Newman</td>
<td>Chad</td>
<td>cnewman</td>
<td>750</td>
</tr>
<tr>
<td>5</td>
<td>Ropeburn</td>
<td>Audrey</td>
<td>aropeburn</td>
<td>1550</td>
</tr>
</tbody>
</table>

4. Populate the `MY_EMPLOYEE` table with the second row of the sample data from the preceding list. This time, list the columns explicitly in the INSERT clause.
Practice 9 (continued)

5. Confirm your addition to the table.

<table>
<thead>
<tr>
<th>ID</th>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>USERID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Patel</td>
<td>Ralph</td>
<td>rpatel</td>
<td>895</td>
</tr>
<tr>
<td>2</td>
<td>Dancs</td>
<td>Betty</td>
<td>bdancs</td>
<td>860</td>
</tr>
</tbody>
</table>

6. Write an INSERT statement in a dynamic reusable script file to load the remaining rows into the MY_EMPLOYEE table. The script should prompt for all the columns (ID, LAST_NAME, FIRST_NAME, USERID, and SALARY). Save this script to a lab_09_06.sql file.

7. Populate the table with the next two rows of the sample data listed in step 3 by running the INSERT statement in the script that you created.

8. Confirm your additions to the table.

<table>
<thead>
<tr>
<th>ID</th>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>USERID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Patel</td>
<td>Ralph</td>
<td>rpatel</td>
<td>895</td>
</tr>
<tr>
<td>2</td>
<td>Dancs</td>
<td>Betty</td>
<td>bdancs</td>
<td>860</td>
</tr>
<tr>
<td>3</td>
<td>Biri</td>
<td>Ben</td>
<td>bbiri</td>
<td>1100</td>
</tr>
<tr>
<td>4</td>
<td>Newman</td>
<td>Chad</td>
<td>cnewman</td>
<td>750</td>
</tr>
</tbody>
</table>

9. Make the data additions permanent.

Update and delete data in the MY_EMPLOYEE table.

10. Change the last name of employee 3 to Drexler.

11. Change the salary to $1,000 for all employees who have a salary less than $900.

12. Verify your changes to the table.

<table>
<thead>
<tr>
<th>ID</th>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>USERID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Patel</td>
<td>Ralph</td>
<td>rpatel</td>
<td>1000</td>
</tr>
<tr>
<td>2</td>
<td>Dancs</td>
<td>Betty</td>
<td>bdancs</td>
<td>1000</td>
</tr>
<tr>
<td>3</td>
<td>Drexler</td>
<td>Ben</td>
<td>bbiri</td>
<td>1100</td>
</tr>
<tr>
<td>4</td>
<td>Newman</td>
<td>Chad</td>
<td>cnewman</td>
<td>1000</td>
</tr>
</tbody>
</table>

13. Delete Betty Dancs from the MY_EMPLOYEE table.

14. Confirm your changes to the table.

<table>
<thead>
<tr>
<th>ID</th>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>USERID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Patel</td>
<td>Ralph</td>
<td>rpatel</td>
<td>1000</td>
</tr>
<tr>
<td>2</td>
<td>Drexler</td>
<td>Ben</td>
<td>bbiri</td>
<td>1100</td>
</tr>
</tbody>
</table>
Practice 9 (continued)

15. Commit all pending changes.

Control data transaction to the **MY_EMPLOYEE** table.

16. Populate the table with the last row of the sample data listed in step 3 by using the statements in the script that you created in step 6. Run the statements in the script.

17. Confirm your addition to the table.

<table>
<thead>
<tr>
<th>ID</th>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>USERID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Patel</td>
<td>Ralph</td>
<td>rpatel</td>
<td>1000</td>
</tr>
<tr>
<td>2</td>
<td>Drexler</td>
<td>Ben</td>
<td>bdbiri</td>
<td>1100</td>
</tr>
<tr>
<td>3</td>
<td>Newman</td>
<td>Chad</td>
<td>cnewman</td>
<td>1000</td>
</tr>
<tr>
<td>4</td>
<td>Ropeburn</td>
<td>Audrey</td>
<td>aropebur</td>
<td>1550</td>
</tr>
</tbody>
</table>

18. Mark an intermediate point in the processing of the transaction.

19. Delete all the rows from the **MY_EMPLOYEE** table.

20. Confirm that the table is empty.

21. Discard the most recent **DELETE** operation without discarding the earlier **INSERT** operation.

22. Confirm that the new row is still intact.

<table>
<thead>
<tr>
<th>ID</th>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>USERID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Patel</td>
<td>Ralph</td>
<td>rpatel</td>
<td>1000</td>
</tr>
<tr>
<td>2</td>
<td>Drexler</td>
<td>Ben</td>
<td>bdbiri</td>
<td>1100</td>
</tr>
<tr>
<td>3</td>
<td>Newman</td>
<td>Chad</td>
<td>cnewman</td>
<td>1000</td>
</tr>
<tr>
<td>4</td>
<td>Ropeburn</td>
<td>Audrey</td>
<td>aropebur</td>
<td>1550</td>
</tr>
</tbody>
</table>

23. Make the data addition permanent.

If you have the time, complete the following exercise:

24. Modify the **lab_09_06.sql** script such that the **USERID** is generated automatically by concatenating the first letter of the first name and the first seven characters of the last name. The generated **USERID** must be in lowercase. Hence, the script should not prompt for the USERID. Save this script to a file named **lab_09_24.sql**.

25. Run the script, **lab_09_24.sql** to insert the following record:

<table>
<thead>
<tr>
<th>ID</th>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>USERID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Anthony</td>
<td>Mark</td>
<td>manthony</td>
<td>1230</td>
</tr>
</tbody>
</table>

26. Confirm that the new row was added with correct **USERID**.

<table>
<thead>
<tr>
<th>ID</th>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>USERID</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Anthony</td>
<td>Mark</td>
<td>manthony</td>
<td>1230</td>
</tr>
</tbody>
</table>
Using DDL Statements to Create and Manage Tables
Objectives

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

- Categorize the main database objects
- Review the table structure
- List the data types that are available for columns
- Create a simple table
- Explain how constraints are created at the time of table creation
- Describe how schema objects work

Objectives

In this lesson, you are introduced to the data definition language (DDL) statements. You are taught the basics of how to create simple tables, alter them, and remove them. The data types available in DDL are shown and schema concepts are introduced. Constraints are discussed in this lesson. Exception messages that are generated from violating constraints during DML operations are shown and explained.
Lesson Agenda

• Database objects
  – Naming rules
• CREATE TABLE statement:
  – Access another user’s tables
  – DEFAULT option
• Data types
• Overview of constraints: NOT NULL, PRIMARY KEY, FOREIGN KEY, CHECK constraints
• Creating a table using a subquery
• ALTER TABLE
  – Read-only tables
• DROP TABLE statement
Database Objects

The Oracle database can contain multiple data structures. Each structure should be outlined in the database design so that it can be created during the build stage of database development.

- **Table:** Stores data
- **View:** Subset of data from one or more tables
- **Sequence:** Generates numeric values
- **Index:** Improves the performance of some queries
- **Synonym:** Gives alternative name to an object

Oracle Table Structures

- Tables can be created at any time, even when users are using the database.
- You do not need to specify the size of a table. The size is ultimately defined by the amount of space allocated to the database as a whole. It is important, however, to estimate how much space a table will use over time.
- Table structure can be modified online.

Note: More database objects are available, but are not covered in this course.

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table</td>
<td>Basic unit of storage; composed of rows</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 numeric values</td>
</tr>
<tr>
<td>Index</td>
<td>Improves the performance of some queries</td>
</tr>
<tr>
<td>Synonym</td>
<td>Gives alternative name to an object</td>
</tr>
</tbody>
</table>
Naming Rules

Table names and column names:
- Must begin with a letter
- Must be 1–30 characters long
- Must contain only A–Z, a–z, 0–9, _, $, and #
- Must not duplicate the name of another object owned by the same user
- Must not be an Oracle server–reserved word

Naming Rules
You name database tables and columns according to the standard rules for naming any Oracle database object:
- Table names and column names must begin with a letter and be 1–30 characters long.
- Names must contain only the characters A–Z, a–z, 0–9, _ (underscore), $, and # (legal characters, but their use is discouraged).
- Names must not duplicate the name of another object owned by the same Oracle server user.
- Names must not be an Oracle server–reserved word.
  - You may also use quoted identifiers to represent the name of an object. A quoted identifier begins and ends with double quotation marks (“”). If you name a schema object using a quoted identifier, then you must use the double quotation marks whenever you refer to that object. Quoted identifiers can be reserved words, although this is not recommended.

Naming Guidelines
Use descriptive names for tables and other database objects.

Note: Names are not case-sensitive. For example, EMPLOYEES is treated to be the same name as eMPloyees or EMPLOYEES. However, quoted identifiers are case-sensitive.

For more information, see the section on Schema Object Names and Qualifiers in the Oracle Database SQL Language Reference 11g, Release 1 (11.1).
Lesson Agenda

• Database objects
  – Naming rules
• CREATE TABLE statement:
  – Access another user’s tables
  – DEFAULT option
• Data types
• Overview of constraints: NOT NULL, PRIMARY KEY, FOREIGN KEY, CHECK constraints
• Creating a table using a subquery
• ALTER TABLE
  – Read-only tables
• DROP TABLE statement
CREATE TABLE Statement

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

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

You specify:
- Table name
- Column name, column data type, and column size

CREATE TABLE Statement

You create tables to store data by executing the SQL CREATE TABLE statement. This statement is one of the DDL statements that are a subset of the SQL statements used to create, modify, or remove Oracle database structures. These statements have an immediate effect on the database and they also record information in the data dictionary.

To create a table, a user must have the CREATE TABLE privilege and a storage area in which to create objects. The database administrator (DBA) uses data control language (DCL) statements to grant privileges to users.

In the syntax:
- schema Is the same as the owner’s name
- table Is the name of the table
- DEFAULT expr Specifies a default value if a value is omitted in the INSERT statement
- column Is the name of the column
- datatype Is the column’s data type and length
Referencing Another User’s Tables

- Tables belonging to other users are not in the user’s schema.
- You should use the owner’s name as a prefix to those tables.

Referencing Another User’s Tables

A schema is a collection of logical structures of data or schema objects. A schema is owned by a database user and has the same name as that user. Each user owns a single schema.

Schema objects can be created and manipulated with SQL and include tables, views, synonyms, sequences, stored procedures, indexes, clusters, and database links.

If a table does not belong to the user, the owner’s name must be prefixed to the table. For example, if there are schemas named USERA and USERB, and both have an EMPLOYEES table, then if USERA wants to access the EMPLOYEES table that belongs to USERB, USERA must prefix the table name with the schema name:

```
SELECT *
FROM userB.employees;
```

If USERB wants to access the EMPLOYEES table that is owned by USERA, USERB must prefix the table name with the schema name:

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

- Specify a default value for a column during an insert.
  
  
  ... hire_date DATE DEFAULT SYSDATE, ...

- Literal values, expressions, or SQL functions are legal values.
- Another column’s name or a pseudocolumn are illegal values.
- The default data type must match the column data type.

CREATE TABLE hire_dates
  (id NUMBER(8),
   hire_date DATE DEFAULT SYSDATE);

DEFAULT Option

When you define a table, you can specify that a column should be given a default value by using the DEFAULT option. This option prevents null values from entering the columns when a row is inserted without a value for the column. The default value can be a literal, an expression, or a SQL function (such as SYSDATE or USER), but the value cannot be the name of another column or a pseudocolumn (such as NEXTVAL or CURRVAL). The default expression must match the data type of the column.

Consider the following examples:

  INSERT INTO hire_dates values(45, NULL);
  The above statement will insert the null value rather than the default value.

  INSERT INTO hire_dates(id) values(35);
  The above statement will insert SYSDATE for the HIRE_DATE column.

Note: In SQL Developer, click the Run Script icon or press [F5] to run the DDL statements. The feedback messages will be shown on the Script Output tabbed page.
Creating Tables

• Create the table:

```sql
CREATE TABLE dept
  (deptno  NUMBER(2),
   dname   VARCHAR2(14),
   loc     VARCHAR2(13),
   create_date DATE DEFAULT SYSDATE);
```

• Confirm table creation:

```sql
DESCRIBE dept
```

Creating Tables

The example in the slide creates the `DEPT` table with four columns: `DEPTNO`, `DNAME`, `LOC`, and `CREATE_DATE`. The `CREATE_DATE` column has a default value. If a value is not provided for an `INSERT` statement, the system date is automatically inserted.

To confirm that the table was created, run the `DESCRIBE` command.

Because creating a table is a DDL statement, an automatic commit takes place when this statement is executed.
Lesson Agenda

• Database objects
  – Naming rules
• CREATE TABLE statement:
  – Access another user’s tables
  – DEFAULT option
• Data types
• Overview of constraints: NOT NULL, PRIMARY KEY, FOREIGN KEY, CHECK constraints
• Creating a table using a subquery
• ALTER TABLE
  – Read-only tables
• DROP TABLE statement
## Data Types

When you identify a column for a table, you need to provide a data type for the column. There are several data types available:

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

### Data Types

When you identify a column for a table, you need to provide a data type for the column. There are several data types available:

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

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAW(size)</td>
<td>Raw binary data of length size (A maximum size must be specified: maximum size is 2,000.)</td>
</tr>
<tr>
<td>LONG RAW</td>
<td>Raw binary data of variable length (up to 2 GB)</td>
</tr>
<tr>
<td>BLOB</td>
<td>Binary data (up to 4 GB)</td>
</tr>
<tr>
<td>BFILE</td>
<td>Binary data stored in an external file (up to 4 GB)</td>
</tr>
<tr>
<td>ROWID</td>
<td>A base-64 number system representing the unique address of a row in its table</td>
</tr>
</tbody>
</table>

Guidelines

- A LONG column is not copied when a table is created using a subquery.
- A LONG column cannot be included in a GROUP BY or an ORDER BY clause.
- Only one LONG column can be used per table.
- No constraints can be defined on a LONG column.
- You might want to use a CLOB column rather than a LONG column.
Datetime Data Types

You can use several datetime data types:

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

Note: These datetime data types are available with Oracle9i and later releases. The datetime data types are discussed in detail in the lesson titled “Managing Data in Different Time Zones” in the Oracle Database 11g: SQL Fundamentals II course.

Also, for more information about the datetime data types, see the topics TIMESTAMP Datatype, INTERVAL YEAR TO MONTH Datatype, and INTERVAL DAY TO SECOND Datatype in Oracle Database SQL Language Reference 11g, Release 1 (11.1).
Lesson Agenda

- Database objects
  - Naming rules
- CREATE TABLE statement:
  - Access another user’s tables
  - DEFAULT option
- Data types
- **Overview of constraints:** NOT NULL, PRIMARY KEY, FOREIGN KEY, CHECK constraints
- Creating a table using a subquery
- ALTER TABLE
  - Read-only tables
- DROP TABLE statement
Including Constraints

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

Constraints

The Oracle server uses constraints to prevent invalid data entry into tables.

You can use constraints to do the following:
• Enforce rules on the data in a table whenever a row is inserted, updated, or deleted from that table. The constraint must be satisfied for the operation to succeed.
• Prevent the deletion of a table if there are dependencies from other tables.
• Provide rules for Oracle tools, such as Oracle Developer.

Data Integrity Constraints

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT NULL</td>
<td>Specifies that the column cannot contain a null value</td>
</tr>
<tr>
<td>UNIQUE</td>
<td>Specifies a column or combination of columns whose values must be unique for all rows in the table</td>
</tr>
<tr>
<td>PRIMARY KEY</td>
<td>Uniquely identifies each row of the table</td>
</tr>
<tr>
<td>FOREIGN KEY</td>
<td>Establishes and enforces a referential integrity between the column and a column of the referenced table such that values in one table match values in another table.</td>
</tr>
<tr>
<td>CHECK</td>
<td>Specifies a condition that must be true</td>
</tr>
</tbody>
</table>
Constraint Guidelines

• You can name a constraint, or the Oracle server generates a name by using the SYS_Cn format.
• Create a constraint at either of the following times:
  – At the same time as the creation of the table
  – After the creation of the table
• Define a constraint at the column or table level.
• View a constraint in the data dictionary.

Constraint Guidelines

All constraints are stored in the data dictionary. Constraints are easy to reference if you give them a meaningful name. Constraint names must follow the standard object-naming rules, except that the name cannot be the same as another object owned by the same user. If you do not name your constraint, the Oracle server generates a name with the format SYS_Cn, where n is an integer so that the constraint name is unique.

Constraints can be defined at the time of table creation or after the creation of the table. You can define a constraint at the column or table level. Functionally, a table-level constraint is the same as a column-level constraint.

For more information, see the section on “Constraints” in Oracle Database SQL Language Reference 11g, Release 1 (11.1).
Defining Constraints

The slide gives the syntax for defining constraints when creating a table. You can create constraints at either the column level or table level. Constraints defined at the column level are included when the column is defined. Table-level constraints are defined at the end of the table definition and must refer to the column or columns on which the constraint pertains in a set of parentheses. It is mainly the syntax that differentiates the two; otherwise, functionally, a column-level constraint is the same as a table-level constraint.

NOT NULL constraints must be defined at the column level.

Constraints that apply to more than one column must be defined at the table level.

In the syntax:
- `schema` Is the same as the owner’s name
- `table` Is the name of the table
- `DEFAULT expr` Specifies a default value to be used if a value is omitted in the `INSERT` statement
- `column` Is the name of the column
- `datatype` Is the column’s data type and length
- `column_constraint` Is an integrity constraint as part of the column definition
- `table_constraint` Is an integrity constraint as part of the table definition
Defining Constraints

• Example of a column-level constraint:

```sql
CREATE TABLE employees(
    employee_id  NUMBER(6)
    CONSTRAINT emp_emp_id_pk PRIMARY KEY,
    first_name   VARCHAR2(20),
    ...);
```

• Example of a table-level constraint:

```sql
CREATE TABLE employees(
    employee_id  NUMBER(6),
    first_name   VARCHAR2(20),
    ...
    job_id       VARCHAR2(10) NOT NULL,
    CONSTRAINT emp_emp_id_pk
    PRIMARY KEY (EMPLOYEE_ID));
```

Defining Constraints (continued)

Constraints are usually created at the same time as the table. Constraints can be added to a table after its creation and also be temporarily disabled.

Both examples in the slide create a primary key constraint on the `EMPLOYEE_ID` column of the `EMPLOYEES` table.

1. The first example uses the column-level syntax to define the constraint.
2. The second example uses the table-level syntax to define the constraint.

More details about the primary key constraint are provided later in this lesson.
### NOT NULL Constraint

The **NOT NULL** constraint ensures that the column contains no null values. Columns without the **NOT NULL** constraint can contain null values by default. **NOT NULL** constraints must be defined at the column level. In the EMPLOYEES table, the EMPLOYEE_ID column inherits a **NOT NULL** constraint as it is defined as a primary key. Otherwise, the LAST_NAME, EMAIL, HIRE_DATE, and JOB_ID columns have the **NOT NULL** constraint enforced on them.

**Note:** Primary key constraint is discussed in detail later in this lesson.
**UNIQUE Constraint**

A `UNIQUE` key integrity constraint requires that every value in a column or a set of columns (key) be unique—that is, no two rows of a table can have duplicate values in a specified column or a set of columns. The column (or set of columns) included in the definition of the `UNIQUE` key constraint is called the *unique key*. If the `UNIQUE` constraint comprises more than one column, that group of columns is called a *composite unique key*.

`UNIQUE` constraints enable the input of nulls unless you also define `NOT NULL` constraints for the same columns. In fact, any number of rows can include nulls for columns without the `NOT NULL` constraints because nulls are not considered equal to anything. A null in a column (or in all columns of a composite `UNIQUE` key) always satisfies a `UNIQUE` constraint.

**Note:** Because of the search mechanism for the `UNIQUE` constraints on more than one column, you cannot have identical values in the non-null columns of a partially null composite `UNIQUE` key constraint.

---

**Example:**

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>EMAIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>King</td>
<td>SKING</td>
</tr>
<tr>
<td>101</td>
<td>Kochhar</td>
<td>NKOCCHAR</td>
</tr>
<tr>
<td>102</td>
<td>De Haan</td>
<td>LDHEAAN</td>
</tr>
<tr>
<td>103</td>
<td>Hurold</td>
<td>AHUHOLD</td>
</tr>
<tr>
<td>104</td>
<td>Ernst</td>
<td>BERNST</td>
</tr>
<tr>
<td>107</td>
<td>Lorenz</td>
<td>BLORENZ</td>
</tr>
</tbody>
</table>

---

**Allowed:**

- INSERT INTO
  - 201 SMITH JSMITH

**Not allowed:**

- INSERT INTO
  - 205 SMITH JSMITH
UNIQUE Constraint

Defined at either the table level or the column level:

```sql
CREATE TABLE employees(
  employee_id      NUMBER(6),
  last_name        VARCHAR2(25) NOT NULL,
  email            VARCHAR2(25),
  salary           NUMBER(8,2),
  commission_pct   NUMBER(2,2),
  hire_date        DATE NOT NULL,
  ...
CONSTRAINT emp_email_uk UNIQUE(email));
```

UNIQUE Constraint (continued)

UNIQUE constraints can be defined at the column level or table level. You define the constraint at the table level when you want to create a composite unique key. A composite key is defined when there is not a single attribute that can uniquely identify a row. In that case, you can have a unique key that is composed of two or more columns, the combined value of which is always unique and can identify rows.

The example in the slide applies the UNIQUE constraint to the EMAIL column of the EMPLOYEES table. The name of the constraint is EMP_EMAIL_UK.

Note: The Oracle server enforces the UNIQUE constraint by implicitly creating a unique index on the unique key column or columns.
PRIMARY KEY Constraint

A PRIMARY KEY constraint creates a primary key for the table. Only one primary key can be created for each table. The PRIMARY KEY constraint is a column or a set of columns that uniquely identifies each row in a table. This constraint enforces the uniqueness of the column or column combination and ensures that no column that is part of the primary key can contain a null value.

Note: Because uniqueness is part of the primary key constraint definition, the Oracle server enforces the uniqueness by implicitly creating a unique index on the primary key column or columns.
FOREIGN KEY Constraint

The FOREIGN KEY (or referential integrity) constraint designates a column or a combination of columns as a foreign key and establishes a relationship with a primary key or a unique key in the same table or a different table.

In the example in the slide, DEPARTMENT_ID has been defined as the foreign key in the EMPLOYEES table (dependent or child table); it references the DEPARTMENT_ID column of the DEPARTMENTS table (the referenced or parent table).

Guidelines

- A foreign key value must match an existing value in the parent table or be NULL.
- Foreign keys are based on data values and are purely logical, rather than physical, pointers.
FOREIGN KEY Constraint

Defined at either the table level or the column level:

```sql
CREATE TABLE employees(
    employee_id      NUMBER(6),
    last_name        VARCHAR2(25) NOT NULL,
    email            VARCHAR2(25),
    salary           NUMBER(8,2),
    commission_pct   NUMBER(2,2),
    hire_date        DATE NOT NULL,
    department_id    NUMBER(4),

    CONSTRAINT emp_dept_id FOREIGN KEY (department_id) REFERENCES departments(department_id),
    CONSTRAINT emp_email_uk UNIQUE(email));
```

FOREIGN KEY Constraint (continued)

FOREIGN KEY constraints can be defined at the column or table constraint level. A composite foreign key must be created by using the table-level definition.

The example in the slide defines a FOREIGN KEY constraint on the DEPARTMENT_ID column of the EMPLOYEES table, using table-level syntax. The name of the constraint is EMP_DEPT_FK.

The foreign key can also be defined at the column level, provided that the constraint is based on a single column. The syntax differs in that the keywords FOREIGN KEY do not appear. For example:

```sql
CREATE TABLE employees
(
    ...
    department_id NUMBER(4) CONSTRAINT emp_deptid_fk REFERENCES departments(department_id),
    ...
)
```
FOREIGN KEY Constraint: Keywords

- **FOREIGN KEY**: Defines the column in the child table at the table-constraint level
- **REFERENCES**: Identifies the table and column in the parent table
- **ON DELETE CASCADE**: Deletes the dependent rows in the child table when a row in the parent table is deleted
- **ON DELETE SET NULL**: Converts dependent foreign key values to null

The foreign key is defined in the child table and the table containing the referenced column is the parent table. The foreign key is defined using a combination of the following keywords:

- **FOREIGN KEY** is used to define the column in the child table at the table-constraint level.
- **REFERENCES** identifies the table and the column in the parent table.
- **ON DELETE CASCADE** indicates that when a row in the parent table is deleted, the dependent rows in the child table are also deleted.
- **ON DELETE SET NULL** indicates that when a row in the parent table is deleted, the foreign key values are set to null.

The default behavior is called the *restrict rule*, which disallows the update or deletion of referenced data.

Without the **ON DELETE CASCADE** or the **ON DELETE SET NULL** options, the row in the parent table cannot be deleted if it is referenced in the child table.
CHECK Constraint

• Defines a condition that each row must satisfy
• The following expressions are not allowed:
  – References to CURRVAL, NEXTVAL, LEVEL, and ROWNUM pseudocolumns
  – Calls to SYSDATE, UID, USER, and USERENV functions
  – Queries that refer to other values in other rows

```
..., salary NUMBER(2)
  CONSTRAINT emp_salary_min
  CHECK (salary > 0),...
```

CHECK Constraint

The CHECK constraint defines a condition that each row must satisfy. The condition can use the same constructs as the query conditions, with the following exceptions:

• References to the CURRVAL, NEXTVAL, LEVEL, and ROWNUM pseudocolumns
• Calls to SYSDATE, UID, USER, and USERENV functions
• Queries that refer to other values in other rows

A single column can have multiple CHECK constraints that refer to the column in its definition. There is no limit to the number of CHECK constraints that you can define on a column.

CHECK constraints can be defined at the column level or table level.

```
CREATE TABLE employees
  (...  
    salary NUMBER(8,2) CONSTRAINT emp_salary_min
    CHECK (salary > 0),  
  ...) 
```
CREATE TABLE: Example

The example in the slide shows the statement that is used to create the EMPLOYEES table in the HR schema.
Violating Constraints

When you have constraints in place on columns, an error is returned if you try to violate the constraint rule. For example, if you try to update a record with a value that is tied to an integrity constraint, an error is returned.

In the example in the slide, department 55 does not exist in the parent table, DEPARTMENTS, and so you receive the “parent key not found” violation ORA-02291.

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

Department 55 does not exist.

**ORA-02291: integrity constraint (ORA100.EXP_DEPT_PK) violated — parent key not found**

*Cause: A foreign key value has no matching primary key value.

*Action: Delete the foreign key or add a matching primary key.
Violating Constraints

You cannot delete a row that contains a primary key that is used as a foreign key in another table.

```
DELETE FROM departments
WHERE department_id = 60;
```

Violating Constraints (continued)

If you attempt to delete a record with a value that is tied to an integrity constraint, an error is returned.

The example in the slide tries to delete department 60 from the DEPARTMENTS table, but it results in an error because that department number is used as a foreign key in the EMPLOYEES table. If the parent record that you attempt to delete has child records, then you receive the “child record found” violation ORA-02292.

The following statement works because there are no employees in department 70:

```
DELETE FROM departments
WHERE department_id = 70;
```

1 rows deleted
Lesson Agenda

• Database objects
  – Naming rules
• CREATE TABLE statement:
  – Access another user’s tables
  – DEFAULT option
• Data types
• Overview of constraints: NOT NULL, PRIMARY KEY, FOREIGN KEY, CHECK constraints
• Creating a table using a subquery
• ALTER TABLE
  – Read-only tables
• DROP TABLE statement
Creating a Table Using a Subquery

- Create a table and insert rows by combining the **CREATE TABLE** statement and the **AS subquery** option.

```sql
CREATE TABLE table
[(column, column...)]
AS subquery;
```

- Match the number of specified columns to the number of subquery columns.
- Define columns with column names and default values.

### Guidelines

- The table is created with the specified column names, and the rows retrieved by the **SELECT** statement are inserted into the table.
- The column definition can contain only the column name and default value.
- If column specifications are given, the number of columns must equal the number of columns in the subquery **SELECT** list.
- If no column specifications are given, the column names of the table are the same as the column names in the subquery.
- The column data type definitions and the **NOT NULL** constraint are passed to the new table. Note that only the explicit **NOT NULL** constraint will be inherited. The **PRIMARY KEY** column will not pass the **NOT NULL** feature to the new column. Any other constraint rules are not passed to the new table. However, you can add constraints in the column definition.
Creating a Table Using a Subquery (continued)

The example in the slide creates a table named DEPT80, which contains details of all the employees working in department 80. Notice that the data for the DEPT80 table comes from the EMPLOYEES table.

You can verify the existence of a database table and check the column definitions by using the DESCRIBE command.

However, be sure to provide a column alias when selecting an expression. The expression SALARY*12 is given the alias ANNSAL. Without the alias, the following error is generated:

```
Error starting at line 1 in command:
CREATE TABLE dept80
AS SELECT employee_id, last_name, salary*12 ANNSAL,
       hire_date
FROM employees
WHERE department_id = 80
Error at Command Line:3 Column:6
Error report:
SQL Error: 0RA-00998: must name this expression with a column alias
00998. 00000 - “must name this expression with a column alias”
```

*Cause:
*Action:
Lesson Agenda

- Database objects
  - Naming rules
- `CREATE TABLE` statement:
  - Access another user’s tables
  - `DEFAULT` option
- Data types
- Overview of constraints: `NOT NULL`, `PRIMARY KEY`, `FOREIGN KEY`, `CHECK` constraints
- Creating a table using a subquery
- `ALTER TABLE`
  - Read-only tables
- `DROP TABLE` statement
ALTER TABLE Statement

Use the ALTER TABLE statement to:

• Add a new column
• Modify an existing column definition
• Define a default value for the new column
• Drop a column
• Rename a column
• Change table to read-only status

ALTER TABLE Statement

After you create a table, you may need to change the table structure for any of the following reasons:

• You omitted a column.
• Your column definition or its name needs to be changed.
• You need to remove columns.
• You want to put the table into the read-only mode

You can do this by using the ALTER TABLE statement.
Read-Only Tables

Use the `ALTER TABLE` syntax to put a table into the read-only mode:

- Prevents DDL or DML changes during table maintenance
- Change it back into read/write mode

```
ALTER TABLE employees READ ONLY;
-- perform table maintenance and then
-- return table back to read/write mode
ALTER TABLE employees READ WRITE;
```

Read-Only Tables

With Oracle Database 11g, you can specify `READ ONLY` to place a table in the read-only mode. When the table is in the `READ-ONLY` mode, you cannot issue any DML statements that affect the table or any `SELECT ... FOR UPDATE` statements. You can issue DDL statements as long as they do not modify any data in the table. Operations on indexes associated with the table are allowed when the table is in the `READ ONLY` mode.

Specify `READ/WRITE` to return a read-only table to the read/write mode.

**Note:** You can drop a table that is in the `READ ONLY` mode. For information about the `ALTER TABLE` statement, see the course titled *Oracle Database 10g SQL Fundamentals II*. 
Lesson Agenda

• Database objects
  – Naming rules
• CREATE TABLE statement:
  – Access another user's tables
  – DEFAULT option
• Data types
• Overview of constraints: NOT NULL, PRIMARY KEY, FOREIGN KEY, CHECK constraints
• Creating a table using a subquery
• ALTER TABLE
  – Read-only tables
• DROP TABLE statement
Dropping a Table

- Moves a table to the recycle bin
- Removes the table and all its data entirely if the `PURGE` clause is specified
- Invalidates dependent objects and removes object privileges on the table

```
DROP TABLE dept80;
```

Dropping a Table

The `DROP TABLE` statement moves a table to the recycle bin or removes the table and all its data from the database entirely. Unless you specify the `PURGE` clause, the `DROP TABLE` statement does not result in space being released back to the tablespace for use by other objects, and the space continues to count towards the user’s space quota. Dropping a table invalidates the dependent objects and removes object privileges on the table.

When you drop a table, the database loses all the data in the table and all the indexes associated with it.

Syntax

```
DROP TABLE table [PURGE]
```

In the syntax, `table` is the name of the table.

Guidelines

- All the data is deleted from the table.
- Any views and synonyms remain, but are invalid.
- Any pending transactions are committed.
- Only the creator of the table or a user with the `DROP ANY TABLE` privilege can remove a table.

Note: Use the `FLASHBACK TABLE` statement to restore a dropped table from the recycle bin. This is discussed in detail in the course titled *Oracle Database 11g: SQL Fundamentals II*.
Summary

In this lesson, you should have learned how to use the `CREATE TABLE` statement to create a table and include constraints:

- Categorize the main database objects
- Review the table structure
- List the data types that are available for columns
- Create a simple table
- Explain how constraints are created at the time of table creation
- Describe how schema objects work

Summary

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

**CREATE TABLE**
- Use the `CREATE TABLE` statement to create a table and include constraints.
- Create a table based on another table by using a subquery.

**DROP TABLE**
- Remove rows and a table structure.
- When executed, this statement cannot be rolled back.
Practice 10: Overview

This practice covers the following topics:

- Creating new tables
- Creating a new table by using the `CREATE TABLE AS` syntax
- Verifying that tables exist
- Setting a table to read-only status
- Dropping tables

Practice 10: Overview

Create new tables by using the `CREATE TABLE` statement. Confirm that the new table was added to the database. You also learn to set the status of a table as `READ ONLY` and then revert to `READ/WRITE`.

Note: For all the DDL and DML statements, click the Run Script icon (or press [F5]) to execute the query in SQL Developer. This way you get to see the feedback messages on the Script Output tab page. For SELECT queries, continue to click the Execute Statement icon or press [F9] to get the formatted output on the Results tab page.
Practice 10

1. Create the DEPT table based on the following table instance chart. Save the statement in a script called lab_10_01.sql, and then execute the statement in the script to create the table. Confirm that the table is created.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>ID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Type</td>
<td>Primary key</td>
<td></td>
</tr>
<tr>
<td>Nulls/Unique</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FK Table</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FK Column</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data type</td>
<td>NUMBER</td>
<td>VARCHAR2</td>
</tr>
<tr>
<td>Length</td>
<td>7</td>
<td>25</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Null</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>NOT NULL</td>
<td>NUMBER(7)</td>
</tr>
<tr>
<td>NAME</td>
<td></td>
<td>VARCHAR2(25)</td>
</tr>
</tbody>
</table>

2. Populate the DEPT table with data from the DEPARTMENTS table. Include only columns that you need.

3. Create the EMP table based on the following table instance chart. Save the statement in a script called lab_10_03.sql, and then execute the statement in the script to create the table. Confirm that the table is created.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>ID</th>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>DEPT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Type</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nulls/Unique</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FK Table</td>
<td>DEPT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FK Column</td>
<td>ID</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data type</td>
<td>NUMBER</td>
<td>VARCHAR2</td>
<td>VARCHAR2</td>
<td>NUMBER</td>
</tr>
<tr>
<td>Length</td>
<td>7</td>
<td>25</td>
<td>25</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Null</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td></td>
<td>NUMBER(7)</td>
</tr>
<tr>
<td>LAST_NAME</td>
<td></td>
<td>VARCHAR2(25)</td>
</tr>
<tr>
<td>FIRST_NAME</td>
<td></td>
<td>VARCHAR2(25)</td>
</tr>
<tr>
<td>DEPT_ID</td>
<td></td>
<td>NUMBER(7)</td>
</tr>
</tbody>
</table>

Oracle Database 11g: SQL Fundamentals I  10 - 41
Practice 10 (continued)

4. 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.

5. Alter the EMPLOYEES2 table status to read-only.

6. Try to insert the following row in the EMPLOYEES2 table:

<table>
<thead>
<tr>
<th>ID</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>SALARY</th>
<th>DEPT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>Grant</td>
<td>Marcie</td>
<td>5678</td>
<td>10</td>
</tr>
</tbody>
</table>

You get the following error message:

```
Error starting at line 1 in command:
INSERT INTO employees2
VALUES (34, 'Grant', 'Marcie', 5678, 10)
Error at Command Line:1 Column:12
Error report:
SQL Error: ORA-12081: update operation not allowed on table "ORA16"."EMPLOYEES2"
12081. 00000 -  "update operation not allowed on table \"%s\".\"%s\"
*Cause:  An attempt was made to update a read-only materialized view.
*Action: No action required. Only Oracle is allowed to update a read-only materialized view.
```

7. Revert the EMPLOYEES2 table to the read/write status. Now, try to insert the same row again. You should get the following messages:

```
ALTER TABLE employees2 succeeded.
1 rows inserted
```

8. Drop the EMPLOYEES2 table.
Creating Other Schema Objects
Objectives

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

- Create simple and complex views
- Retrieve data from views
- Create, maintain, and use sequences
- Create and maintain indexes
- Create private and public synonyms

Objectives

In this lesson, you are introduced to the view, sequence, synonym, and index objects. You are taught the basics of creating and using views, sequences, and indexes.
Lesson Agenda

• Overview of views:
  – Creating, modifying, and retrieving data from a view
  – Data manipulation language (DML) operations on a view
  – Dropping a view

• Overview of sequences:
  – Creating, using, and modifying a sequence
  – Cache sequence values
  – \texttt{NEXTVAL} and \texttt{CURRVAL} pseudocolumns

• Overview of indexes
  – Creating, dropping indexes

• Overview of synonyms
  – Creating, dropping synonyms
Database Objects

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table</td>
<td>Basic unit of storage; composed of rows</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 numeric values</td>
</tr>
<tr>
<td>Index</td>
<td>Improves the performance of data retrieval queries</td>
</tr>
<tr>
<td>Synonym</td>
<td>Gives alternative names to objects</td>
</tr>
</tbody>
</table>

Database Objects

There are several other objects in a database in addition to tables. In this lesson, you learn about the views, sequences, indexes, and synonyms.

With views, you can present and hide data from the tables.

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 data retrieval 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 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.
Advantages of Views

- Views restrict access to the data because it displays selected 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 the section on “CREATE VIEW” in the Oracle Database SQL Language Reference 11g, Release 1 (11.1).
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

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

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

In the syntax:
- `OR REPLACE` Re-creates the view if it already exists
- `FORCE` Creates the view regardless of whether or not the base tables exist
- `NOFORCE` Creates the view only if the base tables exist (This is the default.)
- `view` Is the name of the view
- `alias` Specifies names for the expressions selected by the view’s query (The number of aliases must match the number of expressions selected by the view.)
- `subquery` Is a complete `SELECT` statement (You can use aliases for the columns in the `SELECT` list.)
- `WITH CHECK OPTION` Specifies that only those rows that are 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

**Note:** In SQL Developer, click the Run Script icon or press [F5] to run the data definition language (DDL) statements. The feedback messages will be shown on the Script Output tabbed page.
Creating a View

- Create the EMPVU80 view, which contains details of the employees in department 80:

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

- Describe the structure of the view by using the iSQL*Plus DESCRIBE command:

```
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 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

- The subquery that defines a view can contain complex SELECT syntax, including joins, groups, and subqueries.
- If you do not specify a constraint name for the view created with the WITH CHECK OPTION, the system assigns a default name in the SYS_Cn format.
- You can use the OR REPLACE option to change the definition of the view without dropping and re-creating it, or regranting the 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;
```

• 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 in 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.

Alternatively, you can use an alias after the CREATE statement and before the SELECT subquery. The number of aliases listed must match the number of expressions selected in the subquery.

```
CREATE OR REPLACE VIEW salvu50 (ID_NUMBER, NAME, ANN_SALARY)
AS SELECT employee_id, last_name, salary*12
FROM employees
WHERE department_id = 50;
```

CREATE VIEW succeeded.
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.

```
SELECT *
FROM salvu50;
```

<table>
<thead>
<tr>
<th>D_NUMBER</th>
<th>NAME</th>
<th>ANN_SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Mourgos</td>
<td>69000</td>
</tr>
<tr>
<td>2</td>
<td>Rops</td>
<td>42000</td>
</tr>
<tr>
<td>3</td>
<td>DeWe</td>
<td>37200</td>
</tr>
<tr>
<td>4</td>
<td>Netoo</td>
<td>21300</td>
</tr>
<tr>
<td>5</td>
<td>Yorgas</td>
<td>36000</td>
</tr>
</tbody>
</table>
Modifying a View

- Modify the EMPVU80 view by using a 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;
```

- Column aliases in the CREATE OR REPLACE 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 OR REPLACE 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 OR REPLACE 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 JOIN departments d
ON (e.department_id = d.department_id)
GROUP BY d.department_name;
```

Creating a Complex View

The example in the slide creates a complex view of department names, minimum salaries, maximum salaries, and the 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 `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>Administration</td>
<td>4400</td>
<td>4400</td>
<td>4400</td>
</tr>
<tr>
<td>Accounting</td>
<td>8300</td>
<td>12000</td>
<td>10150</td>
</tr>
<tr>
<td>IT</td>
<td>4200</td>
<td>9000</td>
<td>6400</td>
</tr>
<tr>
<td>Executive</td>
<td>17000</td>
<td>24000</td>
<td>19333.333333333333333333</td>
</tr>
<tr>
<td>Shipping</td>
<td>2500</td>
<td>5800</td>
<td>3500</td>
</tr>
<tr>
<td>Sales</td>
<td>8600</td>
<td>11000</td>
<td>10033.333333333333333333</td>
</tr>
<tr>
<td>Marketing</td>
<td>6000</td>
<td>13000</td>
<td>9500</td>
</tr>
</tbody>
</table>
Rules for 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

Rules for 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`).

Oracle Database 11g: SQL Fundamentals I  11 - 15
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

Rules for 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. You cannot add data to a view if the view contains NOT NULL columns without default values in the base table. All the required values must be present in the view. Remember that you are adding values directly to the underlying table through the view.

For more information, see the section on “CREATE VIEW” in Oracle Database SQL Language Reference 11g, Release 1 (11.1).
Using the **WITH CHECK OPTION** Clause

- You can ensure that DML operations performed on the view stay in the domain of the view by using the **WITH CHECK OPTION** clause:

```
CREATE OR REPLACE VIEW empvu20
AS SELECT *
FROM    employees
WHERE   department_id = 20
        WITH CHECK OPTION CONSTRAINT empvu20_ck ;
```

- Any attempt to **INSERT** a row with a `department_id` other than 20, or to **UPDATE** the department number for any row in the view fails because it violates the **WITH CHECK OPTION** constraint.

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

causes:

```
Error report:
SQL Error: ORA-01402: view WITH CHECK OPTION where-clause violation
01402. 00000 -  "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. With the **WITH CHECK OPTION** clause, therefore, the view can see only the 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 operation on any row in the view results in an Oracle server error.
CREATE OR REPLACE VIEW empvu10
  (employee_number, employee_name, job_title)
AS SELECT employee_id, last_name, job_id
  FROM employees
WHERE department_id = 10
WITH READ ONLY;

Denying DML Operations (continued)

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

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

Similarly, any attempt to insert a row or modify a row using the view with a read-only constraint results in the same error.

Error report:
SQL Error: ORA-42399: cannot perform a DML operation on a read-only view
Removing a View

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

DROP VIEW view;

DROP VIEW empvu80;
DROP VIEW empvu80 succeeded.

Removing a View

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

In the syntax:

`view` is the name of the view
Practice 11: Overview of Part 1

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
• Removing views

Practice 11: Overview of Part 1

Part 1 of this lesson’s practice provides you with a variety of exercises in creating, using, and removing views. Complete questions 1–6 at the end of this lesson.
Lesson Agenda

• Overview of views:
  – Creating, modifying, and retrieving data from a view
  – DML operations on a view
  – Dropping a view

• Overview of sequences:
  – Creating, using, and modifying a sequence
  – Cache sequence values
  – NEXTVAL and CURRVAL pseudocolumns

• Overview of indexes
  – Creating, dropping indexes

• Overview of synonyms
  – Creating, dropping synonyms
Sequences

A sequence is a database object that creates integer values. You can create sequences and then use them to generate numbers.

### Sequences

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table</td>
<td>Basic unit of storage; composed of rows</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 numeric values</td>
</tr>
<tr>
<td>Index</td>
<td>Improves the performance of some queries</td>
</tr>
<tr>
<td>Synonym</td>
<td>Gives alternative names to objects</td>
</tr>
</tbody>
</table>
Sequences

A sequence:
• Can automatically generate unique numbers
• Is a shareable object
• Can be used to create a primary key value
• Replaces application code
• Speeds up the efficiency of accessing sequence values when cached in memory

Sequences (continued)

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

You can define a sequence to generate unique values or to recycle and use the same numbers again.

A typical usage for sequences is to create a primary key value, which must be unique for each row. A 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 independent of tables. Therefore, the same sequence can be used for multiple tables.
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}];
```

CREATE SEQUENCE Statement: Syntax

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.)
Creating a Sequence

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

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

Creating a Sequence (continued)

<table>
<thead>
<tr>
<th>CYCLE</th>
<th>NOCYCLE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Specifies whether the sequence continues to generate values after reaching its maximum or minimum value. (NOCYCLE is the default option.)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CACHE n</th>
<th>NOCACHE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Specifies how many values the Oracle server preallocates and keeps in memory. (By default, the Oracle server caches 20 values.)</td>
</tr>
</tbody>
</table>

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 the section on “CREATE SEQUENCE” in the *Oracle Database SQL Language Reference 11g, Release 1 (11.1)*.

**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.
**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.

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.

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. However, **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 you reference `sequence.CURRVAL`, the last value returned to that user’s process is displayed.
NEXTVAL and CURRVAL Pseudocolumns (continued)

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 the sections on “Pseudocolumns” and “CREATE SEQUENCE” in Oracle Database SQL Language Reference 11g, Release 1 (11.1).
Using a Sequence

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

```
INSERT INTO departments(department_id,
  department_name, location_id)
VALUES (dept_deptid_seq.NEXTVAL,
  'Support', 2500);
```

- View the current value for the `DEPT_DEPTID_SEQ` sequence:

```
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 to generate a new department number as follows.

You can view the current value of the sequence using the `sequence_name.CURRVAL`, as shown in the second slide example. The output of the query is shown below:

```
<table>
<thead>
<tr>
<th>CURRVAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
<tr>
<td>120</td>
</tr>
</tbody>
</table>
```

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

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

**Note:** The preceding example assumes that a sequence called `EMPLOYEE_SEQ` has already been created to generate new employee numbers.
Caching Sequence Values

- 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

Caching Sequence Values

You can 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 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. However, if you do so, each table can contain gaps in the sequential numbers.
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;
```

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 the section on “ALTER SEQUENCE” in *Oracle Database SQL Language Reference 11g, Release 1 (11.1).*
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.
- To remove a sequence, use the **DROP** statement:

```
DROP SEQUENCE dept_deptid_seq;
```

Guidelines for Modifying a Sequence

- You must be the owner or have the **ALTER** privilege for the sequence to modify it. You must be the owner or have the **DROP ANY SEQUENCE** privilege to remove 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 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;
```

- The error:

```
Error report:
SQL Error: 0RA-04009: MAXVALUE cannot be made to be less than the current value
04009. 00000 - "MAXVALUE cannot be made to be less than the current value"
*Cause:   the current value exceeds the given MAXVALUE
*Action:  make sure that the new MAXVALUE is larger than the current value
```
Lesson Agenda

• Overview of views:
  – Creating, modifying, and retrieving data from a view
  – DML operations on a view
  – Dropping a view
• Overview of sequences:
  – Creating, using, and modifying a sequence
  – Cache sequence values
  – \texttt{NEXTVAL} and \texttt{CURRVAL} pseudocolumns
• Overview of indexes
  – Creating, dropping indexes
• Overview of synonyms
  – Creating, dropping synonyms
### Indexes

Indexes are database objects that you can create to improve the performance of some queries. Indexes can also be created automatically by the server when you create a primary key or a unique constraint.

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table</td>
<td>Basic unit of storage; composed of rows</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 numeric values</td>
</tr>
<tr>
<td>Index</td>
<td>Improves the performance of some queries</td>
</tr>
<tr>
<td>Synonym</td>
<td>Gives alternative names to objects</td>
</tr>
</tbody>
</table>
Indexes

An index:

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

Indexes (continued)

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 disk I/O by using an indexed path to locate data quickly. An index is used and maintained automatically by the Oracle server. After an index is created, no direct activity is required by the user.

Indexes are logically and physically independent of the table that 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, the corresponding indexes are also dropped.

For more information, see the section on “Schema Objects: Indexes” in *Oracle Database Concepts 11g, Release 1 (11.1).*
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.

How Are Indexes Created?

You can create two types of indexes.

**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 constraint. The name of the index is the name that is given to the constraint.

**Nonunique index:** This is an index that a user can create. For example, you can create the FOREIGN KEY column index for a join in a query to improve the speed of retrieval.

**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 [UNIQUE][BITMAP]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);
```

---

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

Specify `UNIQUE` to indicate that the value of the column (or columns) upon which the index is based must be unique. Specify `BITMAP` to indicate that the index is to be created with a bitmap for each distinct key, rather than indexing each row separately. Bitmap indexes store the rowids associated with a key value as a bitmap.

For more information, see the section on “CREATE INDEX” in Oracle Database SQL Language Reference 11g, Release 1 (11.1).
## Index Creation Guidelines

### Create an index when:

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

### Do not create an index when:

| ✗ | The columns are not often used as a condition in the query |
| ✗ | The table is small or 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 |

---

### Index Creation Guidelines

**More Is Not Always Better**

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

**When to Create an Index**

Therefore, you should create indexes only if:

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

Remember that if you want to enforce uniqueness, you should define a unique constraint in the table definition. A unique index is then created automatically.
Removing an Index

• Remove an index from the data dictionary by using the `DROP INDEX` command:

```
DROP INDEX index;
```

• Remove the `emp_last_name_idx` index from the data dictionary:

```
DROP INDEX emp_last_name_idx;
```

• To drop an index, you must be the owner of the index or have the `DROP ANY INDEX` privilege.

Removing an Index

You cannot modify indexes. To change an index, you must drop it and then re-create it.

Remove an index definition from the data dictionary by issuing the `DROP INDEX` statement. To drop an index, you must be the owner of the index or have the `DROP ANY INDEX` privilege.

In the syntax, `index` is the name of the index.

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

• Overview of views:
  – Creating, modifying, and retrieving data from a view
  – DML operations on a view
  – Dropping a view

• Overview of sequences:
  – Creating, using, and modifying a sequence
  – Cache sequence values
  – NEXTVAL and CURRVAL pseudocolumns

• Overview of indexes
  – Creating, dropping indexes

• Overview of synonyms
  – Creating, dropping synonyms
Synonyms

Synonyms are database objects that enable you to call a table by another name. You can create synonyms to give an alternative name to a table.
Creating a Synonym for an Object

Simplify access to objects by creating a synonym (another name for an object). With synonyms, you can:

- Create an easier reference to a table that is 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 that is 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 that is 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 that are owned by the same user.

For more information, see the section on “CREATE SYNONYM” in Oracle Database SQL Language Reference 11g, Release 1 (11.1).
Creating and Removing Synonyms

Creating a Synonym
The slide example creates a synonym for the `DEPT_SUM_VU` view for quicker reference.
The database administrator can create a public synonym that is accessible to all users. The following example creates a public synonym named `DEPT` for Alice’s `DEPARTMENTS` table:

```
CREATE PUBLIC SYNONYM dept;
CREATE SYNONYM succeeded.
```

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

```
DROP PUBLIC SYNONYM dept;
```

For more information, see the section on “DROP SYNONYM” in Oracle Database SQL Language Reference 11g, Release 1 (11.1).
Summary

In this lesson, you should have learned how to:

• Create, use, and remove views
• Automatically generate sequence numbers by using a sequence generator
• Create indexes to improve speed of query retrieval
• Use synonyms to provide alternative names for objects

Summary

In this lesson, you should have learned about database objects such as views, sequences, indexes, and synonyms.
Practice 11: Overview of Part 2

This practice covers the following topics:
- Creating sequences
- Using sequences
- Creating nonunique indexes
- Creating synonyms

Practice 11: Overview of Part 2

Part 2 of this lesson’s practice provides you with a variety of exercises in creating and using a sequence, an index, and a synonym.
Complete questions 7–10 at the end of this lesson.
Practice 11

Part 1

1. The staff in the HR department wants to hide some of the data in the EMPLOYEES table. They want a view called EMPLOYEES_VU based on the employee numbers, employee names, and department numbers from the EMPLOYEES table. They want the heading for the employee name to be EMPLOYEE.

2. Confirm that the view works. 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>1</td>
<td>100 King</td>
<td>90</td>
</tr>
<tr>
<td>2</td>
<td>101 Kochhar</td>
<td>90</td>
</tr>
<tr>
<td>3</td>
<td>102 De Haan</td>
<td>90</td>
</tr>
<tr>
<td>4</td>
<td>103 Hunold</td>
<td>60</td>
</tr>
<tr>
<td>5</td>
<td>104 Ernst</td>
<td>60</td>
</tr>
</tbody>
</table>

... ...

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>EMPLOYEE</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>205 Higgins</td>
<td>110</td>
</tr>
<tr>
<td>20</td>
<td>206 Gietz</td>
<td>110</td>
</tr>
</tbody>
</table>

3. Using your EMPLOYEES_VU view, write a query for the HR department 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>90</td>
</tr>
<tr>
<td>Hunold</td>
<td>60</td>
</tr>
<tr>
<td>Ernst</td>
<td>60</td>
</tr>
</tbody>
</table>

... ...

<table>
<thead>
<tr>
<th>EMPLOYEE</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Higgins</td>
<td>110</td>
</tr>
<tr>
<td>Gietz</td>
<td>110</td>
</tr>
</tbody>
</table>
Practice 11 (continued)

4. Department 50 needs access to its employee data. Create a view named \texttt{DEPT50} that contains the employee numbers, employee last names, and department numbers for all employees in department 50. You have been asked to label the view columns \texttt{EMPNO, EMPLOYEE,} and \texttt{DEPTNO}. For security purposes, do not allow an employee to be reassigned to another department through the view.

5. Display the structure and contents of the \texttt{DEPT50} view.

\begin{tabular}{|l|l|l|}
\hline
\textbf{Name} & \textbf{Null} & \textbf{Type} \\
\hline
\texttt{EMPNO} & \texttt{NOT NULL} & \texttt{NUMBER(6)} \\
\texttt{EMPLOYEE} & \texttt{NOT NULL} & \texttt{VARCHAR2(25)} \\
\texttt{DEPTNO} & \texttt{} & \texttt{NUMBER(4)} \\
\hline
\end{tabular}

\begin{tabular}{|c|c|c|}
\hline
\texttt{EMPNO} & \texttt{EMPLOYEE} & \texttt{DEPTNO} \\
\hline
1 & 124 Mourgos & 50 \\
2 & 141 Rajs & 50 \\
3 & 142 Davies & 50 \\
4 & 143 Matos & 50 \\
5 & 144 Vargas & 50 \\
\hline
\end{tabular}

6. Test your view. Attempt to reassign Matos to department 80.
Practice 11 (continued)

Part 2

7. You need a sequence that can be used with the PRIMARY KEY column of the DEPT table. The sequence should start at 200 and have a maximum value of 1,000. Have your sequence increment by 10. Name the sequence DEPT_ID_SEQ.

8. To test your sequence, write a script to insert two rows in the DEPT table. Name your script lab_11_08.sql. Be sure to use the sequence that you created for the ID column. Add two departments: Education and Administration. Confirm your additions. Run the commands in your script.

9. Create a nonunique index on the NAME column in the DEPT table.

10. Create a synonym for your EMPLOYEES table. Call it EMP.
Table Descriptions
Schema Description

Overall Description
The Oracle database sample schemas portray a sample company that operates worldwide to fill orders for several different products. The company has three divisions:
- **Human Resources**: Tracks information about the employees and facilities
- **Order Entry**: Tracks product inventories and sales through various channels
- **Sales History**: Tracks business statistics to facilitate business decisions

Each of these divisions is represented by a schema. In this course, you have access to the objects in all the schemas. However, the emphasis of the examples, demonstrations, and practices is on the Human Resources (HR) schema.

All scripts necessary to create the sample schemas reside in the $ORACLE_HOME/demo/schema/ folder.

Human Resources (HR)
This is the schema that is used in this course. In the Human Resource (HR) records, each employee has an identification number, email address, job identification code, salary, and manager. Some employees earn commissions in addition to their salary.

The company also tracks information about jobs within the organization. Each job has an identification code, job title, and a minimum and maximum salary range for the job. Some employees have been with the company for a long time and have held different positions within the company. When an employee resigns, the duration the employee was working for, the job identification number, and the department are recorded.

The sample company is regionally diverse, so it tracks the locations of its warehouses and departments. Each employee is assigned to a department, and each department is identified either by a unique department number or a short name. Each department is associated with one location, and each location has a full address that includes the street name, postal code, city, state or province, and the country code.

In places where the departments and warehouses are located, the company records details such as the country name, currency symbol, currency name, and the region where the country is located geographically.
The **HR** Entity Relationship Diagram

**DEPARTMENTS**
- `department_id`
- `department_name`
- `manager_id`
- `location_id`

**LOCATIONS**
- `location_id`
- `street_address`
- `postal_code`
- `city`
- `state_province`
- `country_id`

**COUNTRIES**
- `country_id`
- `country_name`
- `region_id`

**REGIONS**
- `region_id`
- `region_name`

**EMPLOYEES**
- `employee_id`
- `first_name`
- `last_name`
- `email`
- `phone_number`
- `hire_date`
- `job_id`
- `salary`
- `commission_pct`
- `manager_id`
- `department_id`

**JOBS**
- `job_id`
- `job_title`
- `min_salary`
- `max_salary`

**JOB_HISTORY**
- `employee_id`
- `start_date`
- `end_date`
- `job_id`
- `department_id`
The Human Resources (HR) Table Descriptions

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>COUNTRY_ID</th>
<th>COUNTRY_NAME</th>
<th>REGION_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CA</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>DE</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>UK</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>US</td>
<td>2</td>
</tr>
</tbody>
</table>

Canada  
Germany  
United Kingdom  
United States of America
The Human Resources (HR) Table Descriptions

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>1</td>
<td>10 Administration</td>
<td>200</td>
<td>1700</td>
</tr>
<tr>
<td>2</td>
<td>20 Marketing</td>
<td>201</td>
<td>1800</td>
</tr>
<tr>
<td>3</td>
<td>50 Shipping</td>
<td>124</td>
<td>1500</td>
</tr>
<tr>
<td>4</td>
<td>60 IT</td>
<td>103</td>
<td>1400</td>
</tr>
<tr>
<td>5</td>
<td>80 Sales</td>
<td>149</td>
<td>2500</td>
</tr>
<tr>
<td>6</td>
<td>90 Executive</td>
<td>100</td>
<td>1700</td>
</tr>
<tr>
<td>7</td>
<td>110 Accounting</td>
<td>205</td>
<td>1700</td>
</tr>
<tr>
<td>8</td>
<td>190 Contracting</td>
<td>(null)</td>
<td>1700</td>
</tr>
</tbody>
</table>
The Human Resources (HR) Table Descriptions

DESCRIBE employees

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

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>SALARY</th>
<th>COMM</th>
<th>MANAGER_ID</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Beamon</td>
<td>King</td>
<td>SHING</td>
<td>515.133.4567</td>
<td>10-JUN-07</td>
<td>AD_PRES</td>
<td>24000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Neena</td>
<td>Kochhar</td>
<td>NHODHNIJ</td>
<td>515.133.4568</td>
<td>21-SEP-89</td>
<td>AD_VP</td>
<td>17000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Lux</td>
<td>Da Haian</td>
<td>LODHAAN</td>
<td>515.133.4569</td>
<td>15-APR-03</td>
<td>AD_VP</td>
<td>17000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Alexander</td>
<td>Harid</td>
<td>AHARIJOI</td>
<td>506.233.4567</td>
<td>03-JAN-00</td>
<td>IT_PROG</td>
<td>30000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Bruce</td>
<td>Ernst</td>
<td>BERNST</td>
<td>506.233.4568</td>
<td>21-AUG-91</td>
<td>IT_PROG</td>
<td>8000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Diana</td>
<td>Lieroltz</td>
<td>DJOBERTZ</td>
<td>506.233.6567</td>
<td>07-DEC-99</td>
<td>IT_PROG</td>
<td>4200</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Kaim</td>
<td>Monigro</td>
<td>NIMOURGOS</td>
<td>560.123.6234</td>
<td>16-OCT-99</td>
<td>ST_MAN</td>
<td>5500</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Trenna</td>
<td>Raps</td>
<td>TRASS</td>
<td>550.231.0639</td>
<td>17-OCT-85</td>
<td>ST_CLERK</td>
<td>3500</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Curtis</td>
<td>Daves</td>
<td>CDAIVES</td>
<td>500.121.2634</td>
<td>25-JAN-97</td>
<td>ST_CLERK</td>
<td>3100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Kendrick</td>
<td>Mates</td>
<td>MMATOS</td>
<td>550.123.2674</td>
<td>10-DEC-98</td>
<td>ST_CLERK</td>
<td>2500</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Peter</td>
<td>Vargas</td>
<td>PVARO</td>
<td>500.121.2689</td>
<td>08-JUL-88</td>
<td>ST_CLERK</td>
<td>2500</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Bevinka</td>
<td>Zulette</td>
<td>BZOLETTE</td>
<td>011.44.2644.42659</td>
<td>25-AUG-09</td>
<td>SA_MAN</td>
<td>10500</td>
<td>0.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Denis</td>
<td>Axel</td>
<td>AXELZ</td>
<td>011.44.2644.426267</td>
<td>11-MAY-88</td>
<td>SA_REF</td>
<td>11000</td>
<td>0.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Jamal</td>
<td>Taylor</td>
<td>JTAYLOR</td>
<td>011.44.2644.426265</td>
<td>24-MAR-98</td>
<td>SA_REF</td>
<td>8500</td>
<td>0.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Emmanuel</td>
<td>Grant</td>
<td>EGANT</td>
<td>011.44.2644.426263</td>
<td>24-MAY-88</td>
<td>SA_REF</td>
<td>7000</td>
<td>0.15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Jennifer</td>
<td>Whalen</td>
<td>ANAHLEI</td>
<td>515.133.4444</td>
<td>17-SEP-97</td>
<td>AD_ASST</td>
<td>4600</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Michael</td>
<td>Hartstein</td>
<td>MHAETSTE</td>
<td>515.133.5555</td>
<td>17-DEC-98</td>
<td>MK_MAN</td>
<td>13000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>Pat</td>
<td>Fey</td>
<td>FFAY</td>
<td>550.133.6666</td>
<td>17-AUG-97</td>
<td>MK_MAN</td>
<td>6000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>Shelley</td>
<td>Higgins</td>
<td>SHINGIN</td>
<td>515.133.0689</td>
<td>07-JUN-94</td>
<td>AC_MGR</td>
<td>12000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>William</td>
<td>Gaetz</td>
<td>WGAEZT</td>
<td>515.133.8881</td>
<td>07-JUN-94</td>
<td>AC_ACC</td>
<td>8300</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Human Resources (HR) Table Descriptions

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>1</td>
<td>102 13-JAN-93</td>
<td>24-JUL-98</td>
<td>IT_PROG</td>
<td>60</td>
</tr>
<tr>
<td>2</td>
<td>101 21-SEP-89</td>
<td>27-OCT-93</td>
<td>AC_ACCOUNT</td>
<td>110</td>
</tr>
<tr>
<td>3</td>
<td>101 28-OCT-93</td>
<td>15-MAR-97</td>
<td>AC_MGR</td>
<td>110</td>
</tr>
<tr>
<td>4</td>
<td>201 17-FEB-96</td>
<td>19-DEC-99</td>
<td>MK_REP</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>114 24-MAR-98</td>
<td>31-DEC-99</td>
<td>ST_CLERK</td>
<td>50</td>
</tr>
<tr>
<td>6</td>
<td>122 01-JAN-99</td>
<td>31-DEC-99</td>
<td>ST_CLERK</td>
<td>50</td>
</tr>
<tr>
<td>7</td>
<td>200 17-SEP-87</td>
<td>17-JUN-93</td>
<td>AD_ASST</td>
<td>90</td>
</tr>
<tr>
<td>8</td>
<td>176 24-MAR-98</td>
<td>31-DEC-98</td>
<td>SA_REP</td>
<td>80</td>
</tr>
<tr>
<td>9</td>
<td>176 01-JAN-99</td>
<td>31-DEC-99</td>
<td>SA_MAN</td>
<td>80</td>
</tr>
<tr>
<td>10</td>
<td>200 01-JUL-94</td>
<td>31-DEC-98</td>
<td>AC_ACCOUNT</td>
<td>90</td>
</tr>
</tbody>
</table>
The Human Resources (HR) Table Descriptions

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>1</td>
<td>AD_PRES</td>
<td>20000</td>
<td>40000</td>
</tr>
<tr>
<td>2</td>
<td>Administration Vice President</td>
<td>15000</td>
<td>30000</td>
</tr>
<tr>
<td>3</td>
<td>AD_ASST</td>
<td>3000</td>
<td>6000</td>
</tr>
<tr>
<td>4</td>
<td>AC_MGR</td>
<td>8200</td>
<td>16000</td>
</tr>
<tr>
<td>5</td>
<td>AC_ACCOUNT</td>
<td>4200</td>
<td>9000</td>
</tr>
<tr>
<td>6</td>
<td>SA_MAN</td>
<td>10000</td>
<td>20000</td>
</tr>
<tr>
<td>7</td>
<td>SA_REP</td>
<td>6000</td>
<td>12000</td>
</tr>
<tr>
<td>8</td>
<td>ST_MAN</td>
<td>5500</td>
<td>8500</td>
</tr>
<tr>
<td>9</td>
<td>ST_CLERK</td>
<td>2000</td>
<td>5000</td>
</tr>
<tr>
<td>10</td>
<td>IT_PROG</td>
<td>4000</td>
<td>10000</td>
</tr>
<tr>
<td>11</td>
<td>MK_MAN</td>
<td>9000</td>
<td>15000</td>
</tr>
<tr>
<td>12</td>
<td>MK_REP</td>
<td>4000</td>
<td>9000</td>
</tr>
</tbody>
</table>
The Human Resources (HR) Table Descriptions

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></td>
<td>NOT NULL</td>
</tr>
<tr>
<td>STATE_PROVINCE</td>
<td></td>
<td>VARCHAR2(30)</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>COUNTRY_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1400 2014 Jabberwocky Rd</td>
<td>26192</td>
<td>Southlake</td>
<td>Texas</td>
<td>US</td>
</tr>
<tr>
<td>2</td>
<td>1500 2011 Interiors Blvd</td>
<td>99238</td>
<td>South San Francisco</td>
<td>California</td>
<td>US</td>
</tr>
<tr>
<td>3</td>
<td>1700 2004 Chladie Rd</td>
<td>98199</td>
<td>Seattle</td>
<td>Washington</td>
<td>US</td>
</tr>
<tr>
<td>4</td>
<td>1800 400 Bloor St. W.</td>
<td>ON M5S 1X8</td>
<td>Toronto</td>
<td>Ontario</td>
<td>CA</td>
</tr>
<tr>
<td>5</td>
<td>2500 Magdelen Centre, The Oxford Science Park</td>
<td>OX9 9ZB</td>
<td>Oxford</td>
<td>Oxford</td>
<td>UK</td>
</tr>
</tbody>
</table>
The Human Resources (HR) Table Descriptions

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>
Oracle Join Syntax
Objectives

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

- Write `SELECT` statements to access data from more than one table using equijoins and nonequijoins
- Join a table to itself by using a self-join
- View data that generally does not meet a join condition by using outer joins
- Generate a Cartesian product of all rows from two or more tables

Objectives

This lesson explains how to obtain data from more than one table. A join is used to view information from multiple tables. Therefore, you can join tables together to view information from more than one table.

Note: Information on joins is found in the section on “SQL Queries and Subqueries: Joins” in *Oracle Database SQL Language Reference 11g, Release 1 (11.1).*
Obtaining Data from Multiple Tables

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

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

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

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

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

Cartesian Products

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

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

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

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

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

---

### EMPLOYEES (20 rows)

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Abel</td>
<td>90</td>
</tr>
<tr>
<td>2</td>
<td>Davies</td>
<td>90</td>
</tr>
<tr>
<td>3</td>
<td>De Haan</td>
<td>90</td>
</tr>
<tr>
<td>4</td>
<td>Ernst</td>
<td>90</td>
</tr>
</tbody>
</table>

### DEPARTMENTS (8 rows)

<table>
<thead>
<tr>
<th>DEPARTMENT_ID</th>
<th>DEPARTMENT_NAME</th>
<th>LOCATION_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Administration</td>
<td>1700</td>
</tr>
<tr>
<td>2</td>
<td>Marketing</td>
<td>1800</td>
</tr>
<tr>
<td>3</td>
<td>Shipping</td>
<td>1500</td>
</tr>
<tr>
<td>4</td>
<td>IT</td>
<td>1400</td>
</tr>
<tr>
<td>5</td>
<td>Sales</td>
<td>2500</td>
</tr>
<tr>
<td>6</td>
<td>Executive</td>
<td>1700</td>
</tr>
<tr>
<td>7</td>
<td>Accounting</td>
<td>1700</td>
</tr>
<tr>
<td>8</td>
<td>Contracting</td>
<td>1700</td>
</tr>
</tbody>
</table>

---

**Cartesian product: 20 x 8 = 160 rows**

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>DEPARTMENT_ID</th>
<th>LOCATION_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>90</td>
</tr>
<tr>
<td>2</td>
<td>101</td>
<td>90</td>
</tr>
<tr>
<td>3</td>
<td>102</td>
<td>90</td>
</tr>
<tr>
<td>4</td>
<td>103</td>
<td>90</td>
</tr>
</tbody>
</table>

150 205 110 1700
160 206 110 1700

---

Oracle Database 11g: SQL Fundamentals I  C - 5
Types of Oracle-Proprietary Joins

- Equijoin
- Nonequijoin
- Outer join
- Self-join

Types of Joins
To join tables, you can use Oracle’s join syntax.

Note: Before the Oracle9i release, the join syntax was proprietary. The SQL:1999–compliant join syntax does not offer any performance benefits over the Oracle-proprietary join syntax.
Oracle does not have an equivalent syntax to support the FULL OUTER JOIN of the SQL:1999–compliant join syntax.
Joining Tables Using Oracle Syntax

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

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

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

Joining Tables Using Oracle Syntax

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

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

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

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

- Use table prefixes to qualify column names that are in multiple tables.
- Use table prefixes to improve performance.
- Instead of full table name prefixes, use table aliases.
- Table aliases give a table a shorter name.
  - Keeps SQL code smaller, uses less memory
- Use column aliases to distinguish columns that have identical names, but reside in different tables.

Qualifying Ambiguous Column Names

When joining two or more tables, you need to qualify the names of the columns with the table name to avoid ambiguity. Without the table prefixes, the DEPARTMENT_ID column in the SELECT list could be from either the DEPARTMENTS table or the EMPLOYEES table. Therefore, it is necessary to add the table prefix to execute your query. If there are no common column names between the two tables, there is no need to qualify the columns. However, using a table prefix improves performance, because you tell the Oracle server exactly where to find the columns.

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

The table name is specified in full, followed by a space and then the table alias. For example, the EMPLOYEES table can be given an alias of e, and the DEPARTMENTS table an alias of d.

Guidelines

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

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

Note: Equijoins are also called simple joins or inner joins.
Retrieving Records with Equijoins

In the example in the slide:

- **The SELECT clause specifies the column names to retrieve:**
  - Employee last name, employee number, and department number, which are columns in the EMPLOYEES table
  - Department number, department name, and location ID, which are columns in the DEPARTMENTS table

- **The FROM clause specifies the two tables that the database must access:**
  - EMPLOYEES table
  - DEPARTMENTS table

- **The WHERE clause specifies how the tables are to be joined:**
  \[
  e\text{.department_id} = d\text{.department_id}
  \]

Because the DEPARTMENT_ID column is common to both tables, it must be prefixed with the table alias to avoid ambiguity. Other columns that are not present in both the tables need not be qualified by a table alias, but it is recommended for better performance.

**Note:** SQL Developer suffixes a “_1” to differentiate between the two DEPARTMENT_IDS.
Retrieving Records with Equijoins: Example

In the example in the slide, the `LOCATIONS` table is joined to the `DEPARTMENTS` table by the `LOCATION_ID` column, which is the only column of the same name in both the tables. Table aliases are used to qualify the columns and avoid ambiguity.
Additional Search Conditions Using the \textit{AND} Operator

In addition to the join, you may have criteria for your \texttt{WHERE} clause to restrict the rows under consideration for one or more tables in the join. The example in the slide limits the rows of output to those with a department ID equal to 20 or 50:

For example, to display employee Matos’ department number and department name, you need an additional condition in the \texttt{WHERE} clause.

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

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>DEPARTMENT_ID</th>
<th>DEPARTMENT_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matos</td>
<td>50</td>
<td>Shipping</td>
</tr>
</tbody>
</table>
Joining More than Two Tables

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

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

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>DEPARTMENT_NAME</th>
<th>CITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abel</td>
<td>Sales</td>
<td>Oxford</td>
</tr>
<tr>
<td>Davies</td>
<td>Shipping</td>
<td>South San Francisco</td>
</tr>
<tr>
<td>De Haan</td>
<td>Executive</td>
<td>Seattle</td>
</tr>
<tr>
<td>Ernst</td>
<td>IT</td>
<td>Southlake</td>
</tr>
<tr>
<td>Fay</td>
<td>Marketing</td>
<td>Toronto</td>
</tr>
</tbody>
</table>

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

A nonequijoin is a join condition containing something other than an equality operator.

The relationship between the EMPLOYEES table and the JOB_GRADES table is an example of a nonequijoin. The SALARY column in the EMPLOYEES table ranges between the values in the LOWEST_SAL and HIGHEST_SAL columns of the JOB_GRADES table. Therefore, each employee can be graded based on the salary. The relationship is obtained using an operator other than the equality operator (=).
Retrieving Records with Nonequijoins

The example in the slide creates a nonequijoin to evaluate an employee’s salary grade. The salary must be between any pair of the low and high salary ranges.

It is important to note that all employees appear exactly once when this query is executed. No employee is repeated in the list. There are two reasons for this:

• None of the rows in the job grade table contain grades that overlap. That is, the salary value for an employee can lie only between the low salary and high salary values of one of the rows in the salary grade table.

• All of the employees’ salaries lie within the limits that are provided by the job grade table. That is, no employee earns less than the lowest value contained in the LOWEST_SAL column or more than the highest value contained in the HIGHEST_SAL column.

Note: Other conditions (such as <= and >=) can be used, but BETWEEN is the simplest. Remember to specify the low value first and the high value last when using the BETWEEN condition. The Oracle server translates the BETWEEN condition to a pair of AND conditions. Therefore, using BETWEEN has no performance benefits, but should be used only for logical simplicity.

Table aliases have been specified in the example in the slide for performance reasons, not because of possible ambiguity.
### Returning Records with No Direct Match with Outer Joins

If a row does not satisfy a join condition, the row does not appear in the query result. For example, in the equijoin condition of the `EMPLOYEES` and `DEPARTMENTS` tables, department ID 190 does not appear because there are no employees with that department ID recorded in the `EMPLOYEES` table. Similarly, there is an employee whose `DEPARTMENT_ID` is set to `NULL`, so this row will also not appear in the query result of an equijoin. To return the department record that does not have any employees, or to return the employee record that does not belong to any department, you can use the outer join.

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

<table>
<thead>
<tr>
<th><strong>DEPARTMENT_ID</strong></th>
<th><strong>LAST_NAME</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>Higgins</td>
</tr>
<tr>
<td>20</td>
<td>Gottz</td>
</tr>
</tbody>
</table>
Outer Joins: Syntax

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

\[
\text{SELECT } \text{table1.column}, \text{table2.column} \\
\text{FROM } \text{table1, table2} \\
\text{WHERE } \text{table1.column} (+) = \text{table2.column};
\]

\[
\text{SELECT } \text{table1.column}, \text{table2.column} \\
\text{FROM } \text{table1, table2} \\
\text{WHERE } \text{table1.column} = \text{table2.column}(+);
\]

Outer Joins: Syntax

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

In the syntax:
- \text{table1.column} = is the condition that joins (or relates) the tables together
- \text{table2.column (+)} is the outer join symbol, which can be placed on either side of the \text{WHERE} clause condition, but not on both sides (Place the outer join symbol following the name of the column in the table without the matching rows.)
Using Outer Joins

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

**Outer Join Restrictions**

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

**Note:** Oracle’s join syntax does not have an equivalent for the `FULL OUTER JOIN` of the `SQL:1999– compliant join syntax.`
Outer Join: Another Example

The query in the above example retrieves all rows in the \texttt{EMPLOYEES} table, even if there is no match in the \texttt{DEPARTMENTS} table.
Joining a Table to Itself

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

- Find Lorentz in the EMPLOYEES table by looking at the LAST_NAME column.
- Find the manager number for Lorentz by looking at the MANAGER_ID column. Lorentz’s manager number is 103.
- Find the name of the manager with EMPLOYEE_ID 103 by looking at the LAST_NAME column. Hunold’s employee number is 103, so Hunold is Lorentz’s manager.

In this process, you look in the table twice. The first time you look in the table to find Lorentz in the LAST_NAME column and the MANAGER_ID value of 103. The second time you look in the EMPLOYEE_ID column to find 103 and the LAST_NAME column to find Hunold.
Self-Join: Example

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

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

In this appendix, you should have learned how to use joins to display data from multiple tables by using Oracle-proprietary syntax.

Summary

There are multiple ways to join tables.

Types of Joins
- Cartesian products
- Equijoins
- Nonequijoins
- Outer joins
- Self-joins

Cartesian Products

A Cartesian product results in a display of all combinations of rows. This is done by omitting the WHERE clause.

Table Aliases
- Table aliases speed up database access.
- Table aliases can help to keep SQL code smaller by conserving memory.
Practice C: Overview

This practice covers the following topics:

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

Practice C: Overview

This practice is intended to give you practical experience in extracting data from more than one table using the Oracle join syntax.
**Practice C**

1. Write a query for the HR department to produce the addresses of all the departments. Use the `LOCATIONS` and `COUNTRIES` tables. Show the location ID, street address, city, state or province, and country in the output. Run the query.

   ![Location Table](image1)

2. The HR department needs a report of all employees. Write a query to display the last name, department number, and department name for all employees. Run the query.

   ![Employee Table](image2)
Practice C (continued)

3. The HR department needs a report of employees in Toronto. Display the last name, job, department number, and department name for all employees who work in Toronto.

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>JOB_ID</th>
<th>DEPARTMENT_ID</th>
<th>DEPARTMENT_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hartstein</td>
<td>MK_MAN</td>
<td>20</td>
<td>Marketing</td>
</tr>
<tr>
<td>Fay</td>
<td>MK_REP</td>
<td>20</td>
<td>Marketing</td>
</tr>
</tbody>
</table>

4. Create a report to display the employees’ last names and employee number along with their managers’ last names and manager number. Label the columns Employee, Emp#, Manager, and Mgr#, respectively. Save your SQL statement as lab_c_04.sql.

<table>
<thead>
<tr>
<th>Employee</th>
<th>Emp#</th>
<th>Manager</th>
<th>Mgr#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kochhar</td>
<td>101</td>
<td>King</td>
<td>100</td>
</tr>
<tr>
<td>De Haan</td>
<td>102</td>
<td>King</td>
<td>100</td>
</tr>
<tr>
<td>Hunold</td>
<td>103</td>
<td>De Haan</td>
<td>102</td>
</tr>
<tr>
<td>Ernst</td>
<td>104</td>
<td>Hunold</td>
<td>103</td>
</tr>
<tr>
<td>Lorentz</td>
<td>107</td>
<td>Hunold</td>
<td>103</td>
</tr>
<tr>
<td>Mourgos</td>
<td>124</td>
<td>King</td>
<td>100</td>
</tr>
<tr>
<td>Rajs</td>
<td>141</td>
<td>Mourgos</td>
<td>124</td>
</tr>
<tr>
<td>Davies</td>
<td>142</td>
<td>Mourgos</td>
<td>124</td>
</tr>
<tr>
<td>Matos</td>
<td>143</td>
<td>Mourgos</td>
<td>124</td>
</tr>
<tr>
<td>Vargas</td>
<td>144</td>
<td>Mourgos</td>
<td>124</td>
</tr>
<tr>
<td>Whalen</td>
<td>200</td>
<td>Kochhar</td>
<td>101</td>
</tr>
<tr>
<td>Hartstein</td>
<td>201</td>
<td>King</td>
<td>100</td>
</tr>
<tr>
<td>Fay</td>
<td>202</td>
<td>Hartstein</td>
<td>201</td>
</tr>
<tr>
<td>Higgins</td>
<td>205</td>
<td>Kochhar</td>
<td>101</td>
</tr>
<tr>
<td>Gietz</td>
<td>206</td>
<td>Higgins</td>
<td>205</td>
</tr>
</tbody>
</table>
Practice C (continued)

5. Modify `lab_c_04.sql` to display all employees including King, who has no manager. Order the results by the employee number. Save your SQL statement as `lab_c_05.sql`. Run the query in `lab_c_05.sql`.

<table>
<thead>
<tr>
<th>Employee</th>
<th>EMP#</th>
<th>Manager</th>
<th>Mgr#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hunold</td>
<td>103</td>
<td>De Haan</td>
<td>102</td>
</tr>
<tr>
<td>Fay</td>
<td>202</td>
<td>Hartstein</td>
<td>201</td>
</tr>
<tr>
<td>Gietz</td>
<td>206</td>
<td>Higgins</td>
<td>205</td>
</tr>
<tr>
<td>Lorentz</td>
<td>107</td>
<td>Hunold</td>
<td>103</td>
</tr>
<tr>
<td>Ernst</td>
<td>104</td>
<td>Hunold</td>
<td>103</td>
</tr>
<tr>
<td>Hartstein</td>
<td>201</td>
<td>King</td>
<td>100</td>
</tr>
<tr>
<td>Zlotkey</td>
<td>149</td>
<td>King</td>
<td>100</td>
</tr>
<tr>
<td>Mourgos</td>
<td>124</td>
<td>King</td>
<td>100</td>
</tr>
<tr>
<td>De Haan</td>
<td>102</td>
<td>King</td>
<td>100</td>
</tr>
<tr>
<td>Kochhar</td>
<td>101</td>
<td>King</td>
<td>100</td>
</tr>
</tbody>
</table>

6. Create a report for the HR department that displays employee last names, department numbers, and all employees who work in the same department as a given employee. Give each column an appropriate label. Save the script to a file named `lab_c_06.sql`.

<table>
<thead>
<tr>
<th>DEPARTMENT</th>
<th>EMPLOYEE</th>
<th>COLLEAGUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td>Fay</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>Hartstein</td>
</tr>
<tr>
<td>3</td>
<td>50</td>
<td>Davies</td>
</tr>
<tr>
<td>4</td>
<td>50</td>
<td>Davies</td>
</tr>
<tr>
<td>5</td>
<td>50</td>
<td>Davies</td>
</tr>
<tr>
<td>6</td>
<td>50</td>
<td>Davies</td>
</tr>
<tr>
<td>7</td>
<td>50</td>
<td>Matos</td>
</tr>
<tr>
<td>8</td>
<td>50</td>
<td>Matos</td>
</tr>
<tr>
<td>9</td>
<td>50</td>
<td>Matos</td>
</tr>
<tr>
<td>10</td>
<td>50</td>
<td>Matos</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DEPARTMENT</th>
<th>EMPLOYEE</th>
<th>COLLEAGUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>110 Higgins</td>
<td>Gietz</td>
</tr>
</tbody>
</table>
7. The HR department needs a report on job grades and salaries. To familiarize yourself with the `JOB_GRADES` table, first show the structure of the `JOB_GRADES` table. Then create a query that displays the name, job, department name, salary, and grade for all employees.

<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>

3 rows selected

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>JOB_ID</th>
<th>DEPARTMENT_NAME</th>
<th>SALARY</th>
<th>GRADE_LEVEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vargas</td>
<td>ST_CLERK</td>
<td>Shipping</td>
<td>2500</td>
<td>A</td>
</tr>
<tr>
<td>Matos</td>
<td>ST_CLERK</td>
<td>Shipping</td>
<td>2600</td>
<td>A</td>
</tr>
<tr>
<td>Davies</td>
<td>ST_CLERK</td>
<td>Shipping</td>
<td>3100</td>
<td>B</td>
</tr>
<tr>
<td>Rajs</td>
<td>ST_CLERK</td>
<td>Shipping</td>
<td>3500</td>
<td>B</td>
</tr>
<tr>
<td>Lorentz</td>
<td>IT_PROG</td>
<td>IT</td>
<td>4200</td>
<td>B</td>
</tr>
<tr>
<td>Whalen</td>
<td>AD_ASST</td>
<td>Administration</td>
<td>4400</td>
<td>B</td>
</tr>
<tr>
<td>Mourgos</td>
<td>ST_MAN</td>
<td>Shipping</td>
<td>5800</td>
<td>B</td>
</tr>
<tr>
<td>Ernst</td>
<td>IT_PROG</td>
<td>IT</td>
<td>6000</td>
<td>C</td>
</tr>
<tr>
<td>Fay</td>
<td>MK_REP</td>
<td>Marketing</td>
<td>6000</td>
<td>C</td>
</tr>
<tr>
<td>Gietz</td>
<td>AC_ACCOUNT</td>
<td>Accounting</td>
<td>8300</td>
<td>C</td>
</tr>
<tr>
<td>De Haan</td>
<td>AD_VP</td>
<td>Executive</td>
<td>17000</td>
<td>E</td>
</tr>
<tr>
<td>King</td>
<td>AD_PRES</td>
<td>Executive</td>
<td>24000</td>
<td>E</td>
</tr>
</tbody>
</table>
Practice C (continued)

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

8. The HR department wants to determine the names of all employees who were hired after Davies. Create a query to display the name and hire date of any employee hired after employee Davies.

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>HIRE_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lorentz</td>
<td>07-FEB-99</td>
</tr>
<tr>
<td>Mourgos</td>
<td>16-NOV-99</td>
</tr>
<tr>
<td>Matos</td>
<td>15-MAR-98</td>
</tr>
<tr>
<td>Vargas</td>
<td>09-JUL-98</td>
</tr>
<tr>
<td>Zlotkey</td>
<td>29-JAN-00</td>
</tr>
<tr>
<td>Taylor</td>
<td>24-MAR-98</td>
</tr>
<tr>
<td>Grant</td>
<td>24-MAY-99</td>
</tr>
<tr>
<td>Fay</td>
<td>17-AUG-97</td>
</tr>
</tbody>
</table>

9. The HR department needs to find the names and hire dates for all employees who were hired before their managers, along with their managers’ names and hire dates. Save the script to a file named lab_c_09.sql.

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>HIRE_DATE</th>
<th>LAST_NAME_1</th>
<th>HIRE_DATE_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vhaleen</td>
<td>17-SEP-87</td>
<td>Kochhar</td>
<td>21-SEP-89</td>
</tr>
<tr>
<td>Hunold</td>
<td>03-JAN-90</td>
<td>De Haan</td>
<td>13-JAN-93</td>
</tr>
<tr>
<td>Vargas</td>
<td>09-JUL-98</td>
<td>Mourgos</td>
<td>16-NOV-99</td>
</tr>
<tr>
<td>Matos</td>
<td>15-MAR-98</td>
<td>Mourgos</td>
<td>16-NOV-99</td>
</tr>
<tr>
<td>Davies</td>
<td>29-JAN-97</td>
<td>Mourgos</td>
<td>16-NOV-99</td>
</tr>
<tr>
<td>Rajs</td>
<td>17-OCT-95</td>
<td>Mourgos</td>
<td>16-NOV-99</td>
</tr>
<tr>
<td>Grant</td>
<td>24-MAY-99</td>
<td>Zlotkey</td>
<td>29-JAN-00</td>
</tr>
<tr>
<td>Taylor</td>
<td>24-MAR-98</td>
<td>Zlotkey</td>
<td>29-JAN-00</td>
</tr>
<tr>
<td>Abel</td>
<td>11-MAY-96</td>
<td>Zlotkey</td>
<td>29-JAN-00</td>
</tr>
</tbody>
</table>
Using SQL*Plus
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

Objectives

You might want to create SELECT statements that can be used again and again. This appendix 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 Interaction

SQL statements

SQL*Plus

Server

Query results

Buffer

SQL scripts

SQL and SQL*Plus

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

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

Features of SQL*Plus

- Accepts ad hoc entry of statements
- Accepts SQL input from files
- Provides a line editor for modifying SQL statements
- Controls environmental settings
- Formats query results into basic reports
- Accesses local and remote databases
### SQL Statements Versus SQL*Plus Commands

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

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

---

### 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:
- 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 repeated 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 the 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

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

To log in from a Windows environment:
1. Select Start > Programs > Oracle > Application Development > SQL*Plus.
2. Enter the username, password, and database name.

To log in from a command-line environment:
1. Log on to your machine.
2. Enter the `sqlplus` command shown in the slide.

In the syntax:

- `username` Your database username
- `password` Your database password (Your password is visible if you enter it here.)
- `@database` 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.
Changing the Settings of SQL*Plus Environment

You can optionally change the look of the SQL*Plus environment by using the SQL*Plus Properties dialog box.

In the SQL*Plus window, right-click the title bar and in the shortcut menu that appears, select Properties. You can then use the Colors tab of the SQL*Plus Properties dialog box to set Screen Background and Screen Text.
Displaying Table Structure

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

```
DESCRIBE tablename
```

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 `DEPARTMENTS` table, use this command:

```
SQL> DESCRIBE DEPARTMENTS
Name                     Null?    Type
------------------------ -------- -------------------------------
DEPARTMENT_ID            NOT NULL NUMBER(4)
DEPARTMENT_NAME          NOT NULL VARCHAR2(30)
MANAGER_ID               NUMBER(6)
LOCATION_ID              NUMBER(4)
```
Displaying Table Structure

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>NUMBER(6)</td>
<td></td>
</tr>
<tr>
<td>LOCATION_ID</td>
<td>NUMBER(4)</td>
<td></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
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]**
  - Inserts an indefinite number of lines
- **I[NPUT] text**
  - Inserts a line consisting of `text`
- **L[IST]**
  - Lists all lines in the SQL buffer
- **L[IST] n**
  - Lists one line (specified by `n`)
- **L[IST] m n**
  - Lists a range of lines (from `m` to `n`) inclusive
- **R[UN]**
  - Displays and runs the current SQL statement in the buffer
- **n**
  - Specifies the line to make the current line
- **n text**
  - Replaces line `n` with `text`
- **0 text**
  - Inserts a line before line 1

### SQL*Plus Editing Commands (continued)

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I[NPUT] text</td>
<td>Inserts a line consisting of <code>text</code></td>
</tr>
<tr>
<td>L[IST] n</td>
<td>Lists one line (specified by <code>n</code>)</td>
</tr>
<tr>
<td>L[IST] m n</td>
<td>Lists a range of lines (from <code>m</code> to <code>n</code>) inclusive</td>
</tr>
<tr>
<td>R[UN]</td>
<td>Displays and runs the current SQL statement in the buffer</td>
</tr>
<tr>
<td>n</td>
<td>Specifies the line to make the current line</td>
</tr>
<tr>
<td>n text</td>
<td>Replaces line <code>n</code> with <code>text</code></td>
</tr>
<tr>
<td>0 text</td>
<td>Inserts a line before line 1</td>
</tr>
</tbody>
</table>

**Note:** You can enter only one SQL*Plus command for each 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 LIST command to display the contents of the SQL buffer. The asterisk (*) 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 (n) of the line that you want to edit. The new current line is displayed.
- Use the APPEND 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.
Using the CHANGE Command

LIST
1* SELECT * from employees

```
c/employees/departments
1* SELECT * from departments
```

LIST
1* SELECT * from departments

Using the CHANGE Command

- Use L[IST] to display the contents of the buffer.
- Use the C[CHANGE] 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</td>
<td>Saves current contents of SQL buffer to a file. Use APPEND to add to an</td>
</tr>
<tr>
<td></td>
<td>existing file; use REPLACE to overwrite an existing file. The default</td>
</tr>
<tr>
<td></td>
<td>extension is .sql.</td>
</tr>
<tr>
<td>GET filename</td>
<td>Writes the contents of a previously saved file to the SQL buffer. The</td>
</tr>
<tr>
<td></td>
<td>default extension for the file name is .sql.</td>
</tr>
<tr>
<td>START filename</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</td>
<td>.ext]</td>
</tr>
<tr>
<td>SPOOL filename</td>
<td>Stores query results in a file. OFF closes the spool file. OUT closes the</td>
</tr>
<tr>
<td></td>
<td>spool file and sends the file results to the printer.</td>
</tr>
<tr>
<td>OFF</td>
<td>OUT]</td>
</tr>
<tr>
<td>EXIT</td>
<td>Quits SQL*Plus</td>
</tr>
</tbody>
</table>
Using the SAVE, START, and EDIT Commands

**SAVE**
Use the `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.

**START**
Use the `START` command to run a script in SQL*Plus. You can also, alternatively, use the symbol `@` to run a script.

```sql
@my_query
```
Using the SAVE, START, and EDIT Commands (continued)

EDIT

Use the EDIT command to edit an existing script. This opens an editor with the script file in it. When you have made the changes, quit the editor to return to the SQL*Plus command line.

Note: The “/” is a delimiter that signifies the end of the statement. When encountered in a file, SQL*Plus runs the statement prior to this delimiter. The delimiter must be the first character of a new line immediately following the statement.
SERVEROUTPUT Command

- Use the `SET SERVEROUT[PUT]` command to control whether to display the output of stored procedures or PL/SQL blocks in SQL*Plus.
- The `DBMS_OUTPUT` line length limit is increased from 255 bytes to 32767 bytes.
- The default size is now unlimited.
- Resources are not preallocated when `SERVEROUTPUT` is set.
- Because there is no performance penalty, use `UNLIMITED` unless you want to conserve physical memory.

```sql
SET SERVEROUT[PUT] {ON | OFF} [SIZE {n | UNL[IMITED]}}
 [FORMAT {WRA[PPED] | WOR[D_WRAPPED] | TRU[NCATED]}}
```

SERVEROUTPUT Command

Most of the PL/SQL programs perform input and output through SQL statements, to store data in database tables or query those tables. All other PL/SQL input/output is done through APIs that interact with other programs. For example, the `DBMS_OUTPUT` package has procedures such as `PUT_LINE`. To see the result outside of PL/SQL requires another program, such as SQL*Plus, to read and display the data passed to `DBMS_OUTPUT`.

SQL*Plus does not display `DBMS_OUTPUT` data unless you first issue the SQL*Plus command `SET SERVEROUTPUT ON` as follows:

```sql
SET SERVEROUTPUT ON
```

**Note**
- `SIZE` sets the number of bytes of the output that can be buffered within the Oracle Database server. The default is `UNLIMITED`. n cannot be less than 2000 or greater than 1,000,000.
- For additional information about `SERVEROUTPUT`, see the *Oracle Database PL/SQL User's Guide and Reference 11g*. 
Using the SQL*Plus SPOOL Command

The SPOOL command stores query results in a file or optionally sends the file to a printer. The SPOOL command has been enhanced. You can now append to, or replace an existing file, where previously you could only use SPOOL to create (and replace) a file. REPLACE is the default.

To spool output generated by commands in a script without displaying the output on the screen, use SET TERMOUT OFF. SET TERMOUT OFF does not affect output from commands that run interactively.

You must use quotes around file names containing white space. To create a valid HTML file using SPOOL APPEND commands, you must use PROMPT or a similar command to create the HTML page header and footer. The SPOOL APPEND command does not parse HTML tags. SET SQLPLUSCOMPATIBILITY to 9.2 or earlier to disable the CREATE, APPEND and SAVE parameters.
Using the AUTOTRACE Command

- Displays a report after the successful execution of SQL DML statements such as SELECT, INSERT, UPDATE, or DELETE.
- The report can now include execution statistics and the query execution path.

```
SET AUTOTRACE {ON | OFF | TRACE[ONLY]} [EXP[LAIN]]
               [STATISTICS]]
```

```
SET AUTOTRACE ON
-- The AUTOTRACE report includes both the optimizer
-- execution path and the SQL statement execution
-- statistics
```

Using the AUTOTRACE Command

`EXPLAIN` shows the query execution path by performing an `EXPLAIN PLAN`. `STATISTICS` displays SQL statement statistics. The formatting of your AUTOTRACE report may vary depending on the version of the server to which you are connected and the configuration of the server. The `DBMS_XPLAN` package provides an easy way to display the output of the `EXPLAIN PLAN` command in several predefined formats.

Note

- For additional information about the package and subprograms, see the Oracle Database PL/SQL Packages and Types Reference 11g guide.
- For additional information about the `EXPLAIN PLAN`, see the Oracle Database SQL Reference 11g.
- For additional information about Execution Plans and the statistics, see the Oracle Database Performance Tuning Guide 11g.
Summary

In this appendix, you should have learned how to use SQL*Plus as an environment to do the following:

- 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.
Performing DML and DDL Operations Using the SQL Developer GUI
Objectives

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

• Perform data definition language (DDL) operations using the SQL Developer menu options
• Perform data manipulation language (DML) operations using the SQL Developer menu options

Objectives

In this appendix, you are introduced to the graphical tool, SQL Developer. You learn how to use the SQL Developer GUI interface for your database development tasks.

This appendix is an add-on to what was discussed in the “Introduction” lesson. It assumes that you have already created a database connection and that you are ready to browse objects in the SQL Developer interface.
Browsing Database Objects

- Create a database connection object.
- Use the Connections Navigator to:
  - Browse through many objects in a database schema
  - Review the definitions of objects at a glance

After you have created a database connection, you can use the Connections Navigator to browse through many objects in a database schema including Tables, Views, Indexes, Packages, Procedures, Triggers, Types, and so on.

SQL Developer uses the left pane for navigation to find and select objects, and the right pane to display information about the selected objects. You can customize many aspects of the appearance of SQL Developer by setting preferences.

You can see the definition of the objects broken into tabs of information that is pulled out of the data dictionary. For example, if you select a table in the Navigator, the details about columns, constraints, grants, statistics, triggers, and so on are displayed on an easy-to-read tabbed page.

If you want to see the definition of the EMPLOYEES table as shown in the slide, perform the following steps:
1. Expand the Connections node in the Connections Navigator.
2. Expand Tables.
3. Click EMPLOYEES. By default, the Columns tab is selected. It shows the column description of the table. By using the Data tab, you can view the table’s data and also enter new rows, update data, and commit these changes to the database.
Creating a Schema Object

- SQL Developer supports the creation of any schema object by using the context menu.
- Edit the objects by using an edit dialog or one of the many context-sensitive menus.
- View the DDL for adjustments such as creating a new object or editing an existing schema object.

SQL Developer supports the creation of any schema object by executing a SQL statement in the SQL Worksheet. Alternatively, you can create objects by using the context menus. When created, you can edit the objects using an edit dialog or one of the many context-sensitive menus.

As new objects are created or existing objects are edited, the data definition language (DDL) for those adjustments is available for review. An Export DDL option is available if you want to create the full DDL for one or more objects in the schema.

The slide shows the creation of a table using the context menu. To open a dialog box for creating a new table, right-click Tables and select New Table. The dialog boxes for creating and editing the database objects have multiple tabs, each reflecting a logical grouping of properties for that type of object.
Creating a New Table: Example

In the Create Table dialog box, if you do not select the Advanced check box, you can create a table quickly by specifying the columns and some frequently used features.

If you select the Advanced check box, the Create Table dialog box changes to one with multiple options, in which you can specify an extended set of features while creating the table.

The example in the slide shows the creation of the DEPENDENTS table by selecting the Advanced check box.

To create a new table, perform the following steps:
1. In the Connections Navigator, right-click Tables.
2. Select Create TABLE.
3. In the Create Table dialog box, select Advanced.
4. Specify column information. In the column properties section, specify the column name, data types, DEFAULT value, and select the “Cannot be Null” check box to define a NOT NULL column. Click + to add a column and click x to remove a column. Note, in the screenshot, the LAST_NAME column is defined as VARCHAR2 (20) and as a NOT NULL column by selecting the “Cannot be Null” check box.
5. Click OK.

Although it is not required, you should also specify a primary key using the Primary Key tab in the dialog box. Sometimes, you may want to edit the table that you have created. To edit a table, right-click the table in the Connections Navigator and select Edit.
Defining Constraints

From the same Create Table dialog box, you can define all types of constraint on the table. In the slide, the Primary Key option is selected. A default constraint name, DEPENDENTS_PK is assigned. From the Available Columns list, you can select the appropriate column and click > to move it to the Selected Columns list. In the slide, ID is specified as the primary key. Similarly, click the Unique Constraints option to define a unique constraint, or click Foreign Keys to define a foreign key constraint. Click OK.

If you want to define constraints on an existing table, in the Connections Navigator, right-click the table, and select Edit. A similar dialog box as shown in the slide above will appear. You can also click the pencil icon to edit.
Modifying a Table

When you create a table, it gets added under the Tables node in the Connections Navigator. You can view the table structure and even modify the column definition of the table. In the Connections Navigator, click the table that you want to modify. On the right side, you get a set of tabs that gives you all the details about the table. The column tab shows the table structure and the Data tab enables you to view the table’s data.

With the Columns tab selected, perform the following:
1. Click Actions… A sub menu appears.
2. Select Table and a set of menu options appears. If you want to rename the table, select Rename.
   Or if you want to delete all the rows from the table, select Truncate. To drop the table, select Drop.
3. If you want to add a check constraint on the GENDER column such that it accepts only “M” and “F” as values, make sure that the GENDER column is selected, then select Constraint, and click Add Check.
4. If you want to drop a column, select Column from the Actions submenu, and click Drop.

Note: You also get the same menu options when you right-click the table in the Connections Navigator.
Adding Data to a Table

To add rows in the table, perform the following:

1. In the Connections Navigator, select the table that you want to add data into. Click the Data tab.
2. Click the Insert Row icon. Insert Row adds an empty row after the selected row for you to enter new data.
3. Enter the field values for the record. To add another row, click the Insert Row icon again.
Updating and Deleting Rows

1. To update data in a table, make the changes directly in the grid of data values in the Data tab. When you enter a cell in the grid, you can directly edit the data for many data types, and for all data types, you can click the ellipsis (...) button to edit the data.
2. Click Delete Selected Row(s) to mark the selected rows for deletion. The actual deletion does not occur until you commit the changes.
Commit and Rollback

After you have inserted new rows or updated any rows, you can either save the changes or undo the changes.

1. To save the changes, click the Commit Changes icon. Commit Changes ends the current transaction and makes permanent all changes performed in the transaction.
2. To undo the changes, click the Rollback Changes icon. Rollback Changes “undoes” any work done in the current transaction.
Creating a View

Views are virtual tables (analogous to queries in some database products) that select data from one or more underlying tables. To create a view, perform the following:

1. In the Connections Navigator, right-click Views and select New View.
2. In the Create View dialog box, select the Advanced check box. If this option is checked, the dialog box changes to include a pane that provides an extended set of features for creating the view.
3. In the SQL Query box, you can directly enter the SQL query. Or you can expand the SQL Query node and use its options individually to define the view step by step. Click OK. The view gets added in the Views node in the Connections Navigator.
Retrieving Data from a View

To retrieve data from a view, perform the following:

1. In the Connections Navigator, expand Views. Select the view that you want to view the data of.
2. Click the Data tab to view its data.
Creating a Sequence

Sequences are used to generate unique integers. You can use sequences to automatically generate primary key values. To create a sequence, perform the following:

1. In the Connections Navigator, expand Sequences. Select New Sequence.
2. In the Create Database Sequence dialog box, specify the schema name and the sequence name. In the Properties tab page, specify the increment value, the minimum value, the start with value, the maximum value, and so on. You can review and make changes to the code for this sequence by clicking the DDL tab. Click OK.

Note: You can edit, drop, or alter a sequence by using the menu options that appear when you right-click the sequence.
Creating Indexes

An index is a database object that contains an entry for each value that appears in the indexed column(s) of the table or cluster, and provides direct, fast access to rows. Indexes are automatically created on primary key columns; however, you must create indexes on other columns to gain the benefits of indexing.

To create an index, perform the following:

1. Right-click Indexes in the Connections Navigator. Select New Index.
2. In the Create Index dialog box, select the schema that will own the index. Specify the name of the index.
3. On the Properties tabbed page, specify the schema that owns the table that you want to index. Select the table that will be associated with the index. Select the type of index. Add a list of index expressions, that is, the table columns or column expressions in the index. To add an index expression, click the Add Column Expression (+) icon; this adds a column name here and in the Column Expression, where you can edit it. You can also specify the order of the index.

Note: You right-click an index and then use the menu options to edit, drop, rebuild, rename it.
Creating a Synonym

Synonyms provide alternative names for tables, views, sequences, or other synonyms. The Connections Navigator has a Synonyms node for all synonyms (public and private) owned by the user associated with the specified connection, and a Public Synonyms node for all public synonyms on the database associated with the connection.

To create a synonym, perform the following:
1. In the Connections Navigator, right-click Synonyms. Select New Synonym.
2. In the Create Database Synonym dialog box, select the Public check box if you want the synonym to be accessible to all users. Private synonyms are accessible only within its schema.
3. Enter the name of the synonym.
4. On the Properties tabbed page, select the schema that contains the object for which this synonym is being defined. You can enter the name of the object directly (if Name based is selected) or, you can select Object based to get a drop-down list of all the objects in the schema. Select the object from the list. Click OK.

Note: Public synonyms get added in a different node in the Connections Navigator. So, look for your public synonyms in the Public Synonyms node. To drop a synonym, simply right-click the synonym and select Drop from the menu.
Using Snippets

Snippets are code fragments that may be just syntax or examples.

When you place your mouse pointer here, it shows the Snippets window. From the drop-down list you can select the functions category you want.

Using Snippets

You may want to use certain code fragments when you are using the SQL Worksheet or creating or editing a PL/SQL function or procedure. SQL Developer has the feature called Snippets. Snippets are code fragments, such as SQL functions, Optimizer hints, and miscellaneous PL/SQL programming techniques. You can drag snippets into the Editor window.

To display Snippets, select View > Snippets.

The Snippets window is displayed on the right side. You can use the drop-down list to select a group. A Snippets button is placed in the right window margin, so that you can display the Snippets window if it becomes hidden.
Using Snippets: Example

To insert a Snippet into your code in a SQL Worksheet or in a PL/SQL function or procedure, drag the snippet from the Snippets window into the desired place in your code. Then you can edit the syntax so that the SQL function is valid in the current context. To see a brief description of a SQL function in a tool tip, place the cursor over the function name.

The example in the slide shows that `CONCAT(char1, char2)` is dragged from the Character Functions group in the Snippets window. Then the `CONCAT` function syntax is edited and the rest of the statement is added such as in the following:

```sql
SELECT CONCAT(first_name, last_name) FROM employees;
```
Search Engines and External Tools

To enhance the productivity of the SQL developers, SQL Developer has added quick links to popular search engines and discussion forums such as AskTom, Google, and so on. Also, you have shortcut icons to some of the frequently used tools such as Notepad, Microsoft Word, Dreamweaver, and so on available to you.

You can add external tools to the existing list or even delete shortcuts to the tools that you do not use frequently. To do so, perform the following:

1. From the Tools menu, select External Tools.
2. In the External Tools dialog box, select New to add new tools. Select Delete to remove any tool from the list.
Setting Preferences

• Customize the SQL Developer interface and environment.
• In the Tools menu, select Preferences.

You can customize many aspects of the SQL Developer interface and environment by modifying the SQL Developer preferences according to your preferences and needs. To modify the SQL Developer preferences, select Tools, then Preferences.

The preferences are grouped in the following categories:

• Environment
• Accelerators (Keyboard shortcuts)
• Code Editors
• Database
• Debugger
• Documentation
• Extensions
• File Types
• Migration
• PL/SQL Compilers
• PL/SQL Debugger, and so on.
SQL Developer Tutorial

- To learn how to use Oracle SQL Developer to perform common database development tasks, take the Oracle SQL Developer tutorial.
- Available at: http://st-curriculum.oracle.com/tutorial/SQLDeveloper/index.htm

SQL Developer Tutorial

The SQL Developer tutorial is an extensive resource for learning how to perform common database development tasks in SQL Developer. It has embedded demos that give a visual step-by-step view of how to perform specific tasks.

Read through the following topics/sections on SQL:
- Working with Database Objects
- Accessing Data
- Manipulating Data
Summary

In this appendix, you should have learned how to use SQL Developer to do the following:

- Browse, create, and edit database objects using the SQL Developer GUI interface
- Insert, update, delete rows from a table using the SQL Developer GUI interface

Summary

SQL Developer is a free graphical tool to simplify the database development tasks. By using SQL Developer, you can browse, create, and edit database objects. By using the quick menu options and the graphical interface, you can perform the same database development tasks that you can, using SQL statements.