GIFT School of Engineering and Applied Sciences

Spring 2020

CS-115/217: Database Systems

Lab-5 Manual

Displaying Data from Multiple Tables- SQL Joins

Introduction to Lab

This lab introduces students to selecting data from multiple tables. A *join* is used to view information from multiple tables. Hence, you can *join* tables together to view information from more than one table. Various types and joins are explained and examined with the use of examples. The main topics of this lab include:

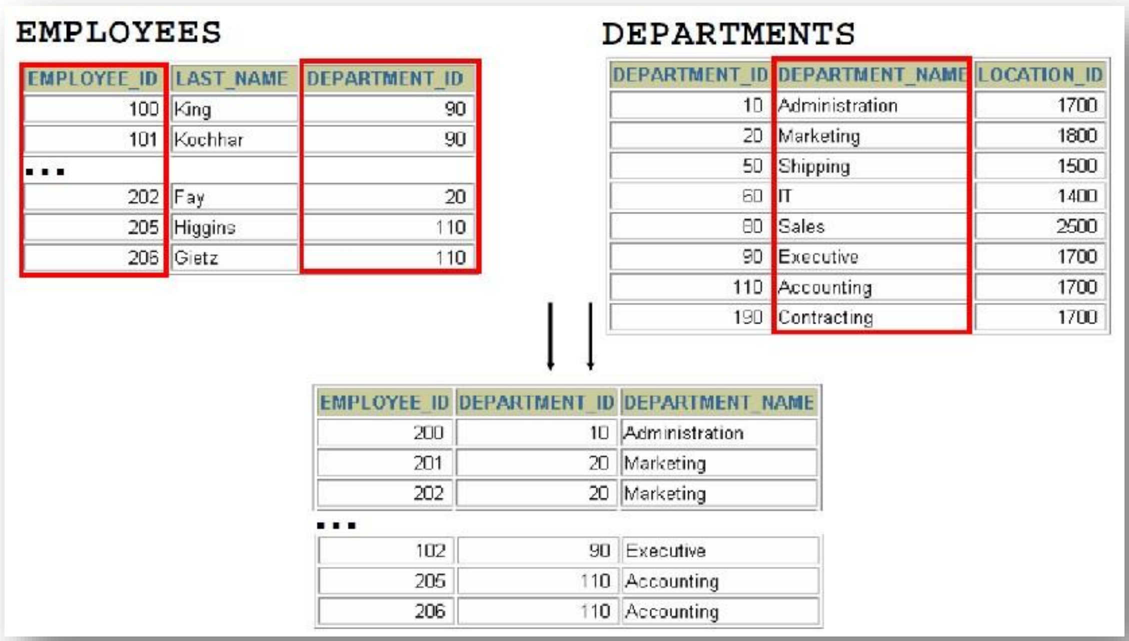
1. Obtaining Data from Multiple Tables
2. Cartesian Products
3. Types of Joins
4. Joining Tables Syntax
5. Creating Natural Joins
6. Creating Joins with the USING Clause
7. Joining Column Names
8. Qualifying Ambiguous Column Names
9. Using Table Aliases
10. Creating Joins with the ON Clause
11. Self-Joins Using the ON Clause
12. Applying Additional Conditions to a Join
13. Creating Three-Way Joins with the ON Clause
14. Non-Equijoins
15. Retrieving Records with Non-Equijoins
16. Outer Joins
17. INNER Versus OUTER Joins
18. LEFT OUTER JOIN
19. RIGHT OUTER JOIN
20. Creating Cross Joins
21. SQL Subqueries
22. Practice SQL Statements

Objectives of this Lab

At the end of this lab, students should be able to:

1. Write SELECT statements to access data from more than one table using SQL:1999 joins
2. Join a table to itself by using a self-join
3. View data that does not meet the join condition by using outer joins
4. Generate a Cartesian product of all rows from two or more tables

1. Obtaining data from Multiple Tables



EMPLOYEES

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

DEPARTMENTS

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

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

Sometimes you need to select data from more than one table. For example, to produce a report of Employee IDs, Department IDs, and Department Names, we need to select data from the EMPLOYEES (Employee ID) and DEPARTMENTS tables (Department ID, Department Name).
To produce this report, we need to link the EMPLOYEES and DEPARTMENTS tables together and access data from both of them.

2. Cartesian Products

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

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

To see an example of a Cartesian product, try executing the following query:

```
SELECT empno, ename, dname, loc FROM emp, dept;
```

The result will be a combination of all rows from the EMP table with all rows from the DEPT table, i.e., 14 x 4 = 56 rows!

3. Types of Joins

To join tables, you can use join syntax that is compliant with the SQL:1999 standard.

1. Cross joins
2. Natural joins
3. USING clause
4. Full (or two-sided) outer joins
5. Arbitrary join conditions for outer joins

4. Joining Tables Syntax

Use the following syntax to join two or more tables:

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

Write the join condition in the WHERE clause. Always prefix the column name with the table name if the same column name appears in more than one joined table.

In the syntax:

- ***table1.column*** denotes the table and column from which data is retrieved
- **NATURAL JOIN** joins two tables based on the same column name
- **JOIN *table* USING *column_name*** performs an equijoin based on the column name
- **JOIN *table ON table1.column_name*** performs an equijoin based on the condition in the ON clause, = *table2.column_name LEFT/RIGHT/FULL OUTER* is used to perform outer joins
- **CROSS JOIN** returns a Cartesian product from the two tables

Important Note: To join *n* tables together, you will need a minimum of *n-1* join conditions. For example, to join four tables together, a minimum of three joins is required.

5. Creating Natural Joins

You can join tables automatically based on columns in the two tables that have matching data types and names. You do this by using the keywords NATURAL JOIN.

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

Example:

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

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

Natural Joins with a WHERE Clause

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

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

6. Creating Joins with a USING Clause

Natural joins use all columns with matching names and data types to join the tables. The USING clause can be used to specify only those columns that should be used for an equijoin.

The columns that are referenced in the USING clause should not have a qualifier (table name or alias) anywhere in the SQL statement.

For example, the following statement is valid:

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

The following statement is invalid because the LOCATION_ID is qualified in the WHERE clause:

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

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

Note: The NATURAL JOIN and USING clauses are mutually exclusive.

7. Joining Column names

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. Frequently, this type of join involves primary and foreign key complements.

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

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

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

8. Qualifying Ambiguous Column Names

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. It is necessary to add the table prefix to execute your query:

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

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

Note: When joining with the USING clause, you cannot qualify a column that is used in the USING clause itself. Furthermore, if that column is used anywhere in the SQL statement, you cannot alias it.

9. Using Table Aliases

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

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

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

Guidelines

1. Table aliases can be up to 30 characters in length, but shorter aliases are better than longer ones.
2. 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.
3. The table alias is valid for only the current SELECT statement.

10. Creating Joins with the ON Clause

Use the ON clause to specify a join condition. This lets you specify join conditions separate from any search or filter conditions in the WHERE clause.

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

In this example, the DEPARTMENT_ID columns in the EMPLOYEES and DEPARTMENTS table are joined using the ON clause. Wherever a department ID in the EMPLOYEES table equals a department ID in the DEPARTMENTS table, the row is returned.

You can also use the ON clause to join columns that have different names.

11. Self-Joins Using the ON Clause

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 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.

The below example is a self-join of the EMPLOYEES table, based on the EMPLOYEE_ID and MANAGER_ID columns.

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

The ON clause can also be used to join columns that have different names, within the same table or in a different table.

12. Applying Additional Conditions to a Join

You can apply additional conditions to the join.

This example performs a join on the EMPLOYEES and DEPARTMENTS tables and, in addition, displays only employees who have a manager ID of 149. To add additional conditions to the ON clause, you can add AND clauses. Alternatively, you can use a WHERE clause to apply additional conditions:

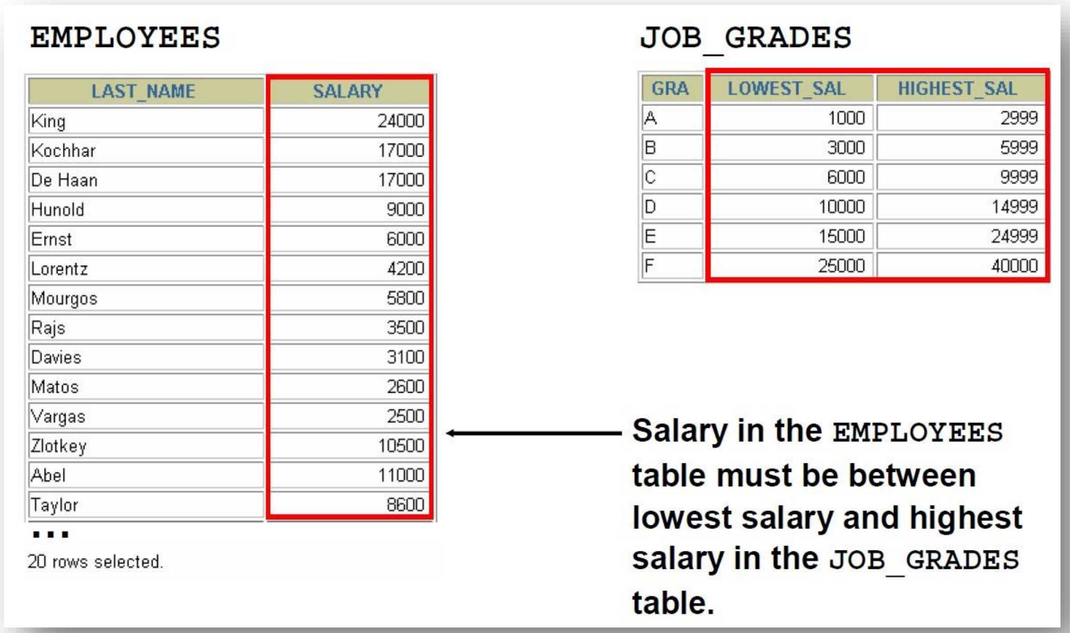
```
SELECT e.employee_id, e.last_name, e.department_id, d.department_id,
d.location_id FROM employees e JOIN departments d ON (e.department_id
= d.department_id) WHERE e.manager_id = 149;
```

13. Creating Three-Way Joins with the ON Clause

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

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

14. Non-Equijoins



EMPLOYEES

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

...

20 rows selected.

JOB_GRADES

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

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

A non-equijoin is a join condition containing something other than an equality operator.

The relationship between the EMPLOYEES table and the JOB_GRADES table is an example of a non-equijoin. A relationship between the two tables is that the SALARY column in the

EMPLOYEES table must be between the values in the LOWEST_SALARY and

HIGHEST_SALARY columns of the JOB_GRADES table. The relationship is obtained using an operator other than equality (=).

15. Retrieving Records with Non-Equijoins

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

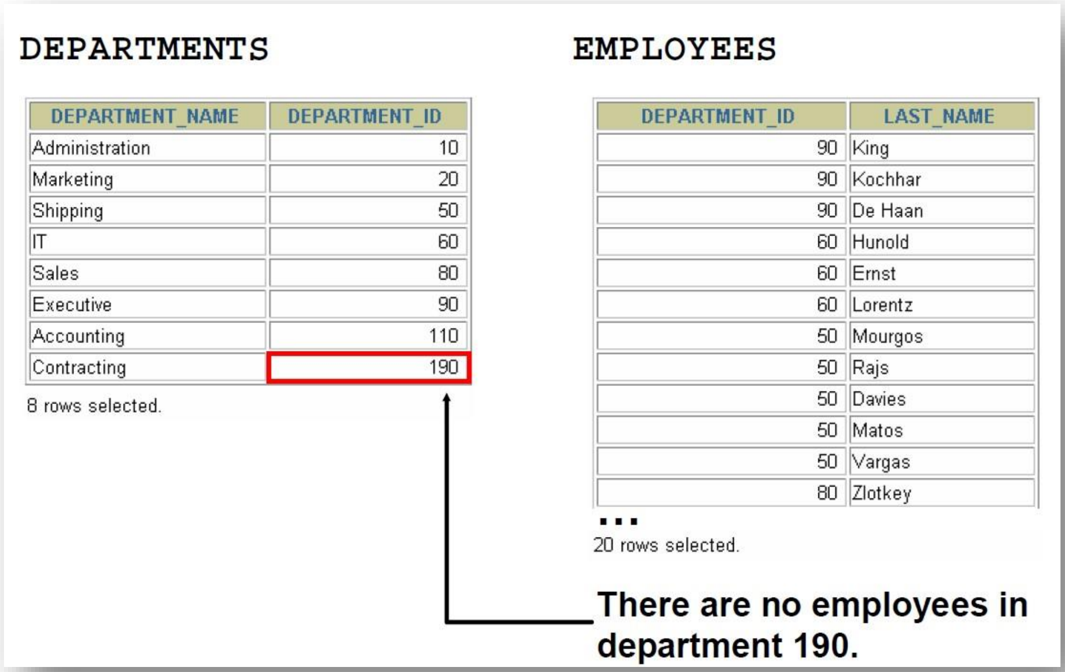
The above example creates a non-equijoin to evaluate an employee's salary grade. The salary must be *between* any pair of the low and high salary ranges.

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

- None of the rows in the job grade table contain grades that overlap. That is, the salary value for an employee can lie only between the low salary and high salary values of one of the rows in the salary grade table.
- All of the employees' salaries lie within the limits 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 BETWEEN. Table aliases have been specified in the slide example for performance reasons, not because of possible ambiguity.

16. Outer Joins



DEPARTMENTS

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

8 rows selected.

EMPLOYEES

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

...

20 rows selected.

There are no employees in department 190.

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 `EMPLOYEES` and `DEPARTMENTS` tables, department ID 190 does not appear because there are no employees with that department ID recorded in the EMPLOYEES table. Instead of seeing 20 employees in the result set, you see 19 records.

To return the department record that does not have any employees, you can use an outer join.

17. INNER Versus OUTER Joins

Joining tables with the NATURAL JOIN, USING, or ON clauses results in an inner join.

Any unmatched rows are not displayed in the output. To return the unmatched rows, you can use an outer join. An outer join returns all rows that satisfy the join condition and also returns some or all of those rows from one table for which no rows from the other table satisfy the join condition.

There are three types of outer joins:

- LEFT OUTER
- RIGHT OUTER
- FULL OUTER

18. LEFT OUTER JOIN

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

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

19. RIGHT OUTER JOIN

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

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

20. Creating Cross Joins

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

The above example produces a Cartesian product of the EMPLOYEES and DEPARTMENTS tables.

21. SQL Subqueries

A Subquery or Inner query or a Nested query is a query within another SQL query and embedded within the WHERE clause.

A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.

Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc.

There are a few rules that subqueries must follow –

- Subqueries must be enclosed within parentheses.
- A subquery can have only one column in the SELECT clause, unless multiple columns are in the main query for the subquery to compare its selected columns.
- An ORDER BY command cannot be used in a subquery, although the main query can use an ORDER BY. The GROUP BY command can be used to perform the same function as the ORDER BY in a subquery.
- Subqueries that return more than one row can only be used with multiple value operators such as the IN operator.
- A subquery cannot be immediately enclosed in a set function.
- The BETWEEN operator cannot be used with a subquery. However, the BETWEEN operator can be used within the subquery.

Subqueries with the SELECT Statement

Subqueries are most frequently used with the SELECT statement. The basic syntax is as follows –

```
SELECT column_name [, column_name ]
FROM   table1 [, table2 ]
WHERE  column_name OPERATOR
   (SELECT column_name [, column_name ]
   FROM table1 [, table2 ]
   [WHERE])
```

Example:

Write a query to display the name (first name and last name) for those employees who gets more salary than the employee whose ID is 112

```
SELECT FIRST_NAME, LAST_NAME FROM employees WHERE SALARY >
(SELECT SALARY FROM employees WHERE EMPLOYEE_ID = 112);
```

22. Practice SQL Statements

1. Write a query to display the name, department number, and department name for all employees.
2. Create a unique listing of all jobs that are in department 30.
3. Write a query to display the employee name, department name, location ID, and city of all employees who earn a commission.
4. Display the employee name and department name for all employees who have an *A* in their last names.
5. Write a query to display the name, department number, and department name for all employees who work in Seattle.
6. Display the employee name and employee number along with their manager's name and manager number. Label the columns **Employee, Emp#, Manager,** and **Mgr#,** respectively.
7. Modify query # 6 to display all employees including King, who has no manager. Order the results by the employee number.
8. Create a query that displays employee name, department numbers, and all the employees who work in the same department as a given employee. Label the columns **Department, Employee, Colleague.**
9. Create a query that displays the name, job title for all employees
10. Create a query to display the name, department_name and hire date of all employees
11. Write a query to display the name (first name and last name), salary, department id, job id for those employees who works in the same designation as the employee works whose id is 169