Database Systems
Lab 6
Joining multiple tables
Version 1.0

In this practical session, we will see how multiple table queries can be implemented. Obviously, this session will invoke a very important aspect of database systems- JOIN statement.

Run the script **join.sql-** it will create the tables. Then insert data into the tables by using **joindata.sql** script. At first, we will play with the following two tables named Department and Location.

DEPARTMENT TABLE

DEPT_ID	NAME	LOCATION_ID
10	ACCOUNTING	122
20	RESEARCH	124
30	SALES	123
40	OPERATIONS	167
12	RESEARCH	122
13	SALES	122
14	OPERATIONS	122
23	SALES	124
24	OPERATIONS	124
34	OPERATIONS	123
43	SALES	167

LOCATION TABLE

LOCATION_ID	REGIONAL_GROUP
122	NEW YORK
124	DALLAS
123	CHICAGO
167	BOSTON
144	SAN FRANCISCO

Say, ***you want to know the place where the departments of Department table are situated. From the above 2 tables, you can say Operations department with ID 14 is at location ID 122 means it is in New York.*** But how will you know that by SQL?

```
SELECT d.dept_id, d.name, l.regional_group
       FROM location l,department d
       WHERE l.location_id=d.location_id;
```

This query gives you the right answer.

In this case, ALIASING is used. "Location l" means you are representing the table "Location" with "l". It is for simplicity. ***REMEMBER- to get data from 2 tables, you must have a common field on both the tables.***

The same effect we now will get with JOIN operation. Try the following query. It gives you the same result.

```
SELECT d.dept_id, d.name, l.regional_group
       FROM department d JOIN location l
       ON d.location_id = l.location_id;
```

You see- the common column of the table is used in this case. SQL facilitates you to do this with USING clause as well. The following gives you the same result as the previous 2.

```
SELECT d.dept_id, d.name, l.regional_group
       FROM department d JOIN location l
       USING (location_id);
```

The USING clause affects the semantics of the SELECT clause. The USING clause tells Oracle that the tables in the join have identical names for the column in the USING clause. Oracle then merges those two columns, and recognizes only one such column with the given name. ***If you include a join column in the SELECT list, Oracle doesn't allow you to qualify that column with a table name (or table alias).*** If you attempt to qualify a join column name in the SELECT list using either an alias or a table name, you will get an error:

```
SELECT department.location_id, department.name, location.regional_group
       FROM department JOIN location
       USING (location_id);
```

Conditions using multiple columns

Quite often you will encounter a join condition that involves multiple columns from each table. If a join condition consists of multiple columns, you need to specify all the predicates in the ON clause. For example, if tables A and B are joined based on columns c1 and c2, the join condition would be:

```
SELECT  . . .
FROM A JOIN B
ON A.c1 = B.c1 AND A.c2 = B.c2;
```

If the column names are identical in the two tables, you can use the USING clause and specify all the columns in one USING clause, separated by commas. The previous join condition can be rewritten as:

```
SELECT  . . .
FROM A JOIN B
USING (c1, c2);
```

Natural Join

A natural join between two tables relates the rows from the two tables based on all pairs of columns, one column from each table, with matching names. You don't specify a join condition. The following example illustrates a natural join:

```
SELECT d.name, l.regional_group
       FROM department d NATURAL JOIN location l;
```

In this example, the two tables—department and location—have the same name for the column location_id. Therefore, the join takes place by equating the location_id from the department table to the location_id from the location table. The preceding query is equivalent to the following queries:

```
SELECT department.name, location.regional_group
       FROM department JOIN location
       ON department.location_id = location.location_id;
```

```
SELECT department.name, location.regional_group
       FROM department JOIN location
       USING (location_id);
```

While using a natural join, you are not allowed to qualify the common columns with table names or aliases (similar to the effect of the USING clause). For example, if you want to include the location_id column in the SELECT list, and you specify department.location_id, you will get an error:

```
SELECT department.location_id, department.name, location.regional_group
       FROM department NATURAL JOIN location;
```

You need to remove the department qualifier so the location_id column can include it in the SELECT list:

```
SELECT location_id, department.name, location.regional_group
       FROM department NATURAL JOIN location;
```

Implicit specification of join conditions can have some unwanted side affects. Let's take the example of join between the supplier and part tables to illustrate this:

Supplier Table

SUPPLIER_ID
NAME

Part Table

PART_NBR
NAME
SUPPLIER_ID
STATUS
INVENTORY_QTY
UNIT_COST
RESUPPLY_DATE

An inner join between these two tables, generates the following result:

```
SELECT supplier.supplier_id, part.part_nbr
       FROM supplier JOIN part
       ON supplier.supplier_id = part.supplier_id;
```

SUPPLIER_ID	PART_NBR
101	HD211
102	P3000

The following example illustrates a natural join between these two tables:

```
SELECT supplier_id, part.part_nbr
       FROM supplier NATURAL JOIN part;
```

No output. What happened? The reason lies in the fact that, aside from supplier_id, these two tables have another pair of columns with a common name. That column is *name*. So, when you ask for a natural join between the supplier and the part tables, the join takes place not only by equating the supplier_id column of the two tables, but the name column from the two tables is equated as well. Since, no supplier name is the same as a part name from that same

supplier, no rows are returned by the query. The equivalent inner join of the preceding natural join is:

```
SELECT supplier.supplier_id, part.part_nbr
       FROM supplier JOIN part
       ON supplier.supplier_id = part.supplier_id
       AND supplier.name = part.name;
```

or, expressed via the USING clause:

```
SELECT supplier_id, part.part_nbr
       FROM supplier JOIN part
       USING (supplier_id, name);
```

Cross Joins/ Cartesian Products

Now, take a look at the employee table by
`SELECT * FROM employee;`

And then take a look at the department table by
`SELECT * FROM department;`

Take a note at the individual number of records of each table. One has 14 rows and the other has 13. So, what will be their Cartesian product? 14 multiplied by 13 equals 182 right? Try to execute the following.

```
SELECT e.lname, d.name
       FROM employee e CROSS JOIN department d;
```

Since the query didn't specify a join condition, each row from the employee table is combined with each row from the department table. Needless to say, this result set is of little use. More often than not, a cross join produces a result set containing misleading rows. Therefore, ***unless you are sure that you want a Cartesian product, don't use a cross join.***

Notice the use of the keyword CROSS before the JOIN keyword in the previous example. ***If you omit the CROSS keyword, and don't specify a join condition, Oracle will throw an error, because it thinks that you are attempting a regular join and have inadvertently omitted the join condition.***

What happens when you specify the CROSS keyword as well as a join condition through an ON or USING clause? Oracle rejects your query with an error, and rightly so, because cross joins are joins without join conditions.

Inner Join

Inner joins are the most commonly used joins. When people refer simply to a "join," they most likely mean an "inner join." An inner join relates the rows from the source tables based on the join condition, and returns the rows that satisfy it. For example, to list the name and department for each employee, you would use the following SQL statement:

```
SELECT e.lname, d.name
       FROM employee e JOIN department d
       ON e.dept_id = d.dept_id;
```

And now try this-

```
SELECT e.lname, d.name
       FROM employee e INNER JOIN department d
       ON e.dept_id = d.dept_id;
```

They are the same! The JOIN keyword, unless prefixed with another keyword, means an inner join. Optionally, you can use the INNER keyword before the JOIN keyword to explicitly indicate an inner join.

Outer Join

To list all departments even if they are not related to any particular location, you can perform a LEFT OUTER JOIN between the department and the location tables. For example:

```
SELECT d.dept_id, d.name, l.regional_group
       FROM department d LEFT OUTER JOIN location l
       ON d.location_id = l.location_id;
```

Likewise, to list all the locations even if they are not related to any particular department, you can perform a RIGHT OUTER JOIN between the location and the department tables. For example:

```
SELECT d.dept_id, d.name, l.regional_group
       FROM department d RIGHT OUTER JOIN location l
       ON d.location_id = l.location_id;
```

The LEFT and RIGHT keywords in an outer join query are relative to the position of the tables in the FROM clause. The same result can be achieved using either a LEFT OUTER JOIN or a RIGHT OUTER JOIN, by switching the position of the tables. For example, the following two queries are equivalent:

```
SELECT d.dept_id, d.name, l.regional_group
       FROM department d LEFT OUTER JOIN location l
       ON d.location_id = l.location_id;

SELECT d.dept_id, d.name, l.regional_group
       FROM location l RIGHT OUTER JOIN department d
       ON d.location_id = l.location_id;
```

In each case, the directional word, either LEFT or RIGHT, points toward the anchor table, the table that is required. The other table is then the optional table in the join.

Occasionally, you may need the effect of an outer join in both directions, which you can think of as a combination of LEFT and RIGHT outer joins. For example, you may need to list all the departments (with or without a location), as well as all the locations (with or without a department). Use a FULL OUTER JOIN to generate such a result set:

```
SELECT d.dept_id, d.name, l.regional_group
       FROM department d FULL OUTER JOIN location l
       ON d.location_id = l.location_id;
```

Equi-joins versus non-equi joins

The join condition determines whether a join is an equi-join or a non-equi-join. When a join condition relates two tables by equating the columns from the tables, it is an equi-join. When a join condition relates two tables by an operator other than equality, it is a non-equi-join. A query may contain equi-joins as well as non-equi-joins.
Equi-joins are the most common join type. For example, if you want to list all the parts supplied by all the suppliers, you can join the supplier table with the part table by equating the supplier_id from one table to that of the other:

```
SELECT s.name supplier_name, p.name part_name
       FROM supplier s JOIN part p
       ON s.supplier_id = p.supplier_id;
```

However, there are situations in which you need non-equi-joins to get the required information. For example, if you want to list the inventory_class of each part, and the inventory_class is based on a range of unit costs, you need to execute the following query:

```
SELECT p.name part_name, c.inv_class inv_class
       FROM part p JOIN inventory_class c
       ON p.unit_cost BETWEEN c.low_cost AND c.high_cost;
```

Self Joins

There are situations in which one row of a table is related to another row of the same table. The employee table is a good example. The manager of one employee is also an employee. The rows for both are in the same employee table. This relationship is indicated in the manager_emp_id column.

To get information about an employee and his manager, you have to join the employee table with itself. You can do that by specifying the employee table twice in the FROM clause and using two different table aliases, thereby treating employee as if it were two separate tables. The following example lists the name of each employee and his manager:

```
SELECT e.lname employee, m.lname manager
       FROM employee e JOIN employee m
       ON e.manager_emp_id = m.emp_id;
```

Even though the employee table has 14 rows, the previous query returned only 13 rows. This is because there is an employee without a manager_emp_id. Oracle excludes that employee's row from the result set while performing the self inner join. To include employees without manager_emp_id values, in other words, without managers, you need an outer join:

```
SELECT e.lname employee, m.lname manager
       FROM employee e LEFT OUTER JOIN employee m
       ON e.manager_emp_id = m.emp_id;
```

Be careful when using a LEFT or RIGHT outer join to join a table to itself. If you choose the wrong direction, you may get an absurd result set that makes no sense. In this case, we want to list all the employees irrespective of whether they have a manager or not. Therefore, the employee table we need to make optional is the one from which we are drawing manager names.