
Agenda

- Compound Joins
- Types of Joins
- Cross Join
- USING
- NATURAL
- IMPLICIT JOIN
- Join with WHERE vs ON
- UNION

Compound Joins

Till now, whenever we did a join, we joined based on only 1 condition. Like in where clause we can combine multiple conditions, in Joins as well, we can have multiple conditions.

Let's see an example. For every film, name all the films that were released in the range of 2 years before or after that film and their rental rate was more than the rate of that movie.

```
SELECT f1.name, f2.name
FROM film f1
JOIN film f2
ON (f2.year BETWEEN f1.year - 2 AND f1.year + 2) AND f2.rental > f1.rental;
```

> *Note:*
>
> 1. *Join does not need to happen on equality of columns always.*
> 2. *Join can also have multiple conditions.*

A Compound Join is one where Join has multiple conditions on different columns.

Types of Joins

While we have pretty much discussed everything that is mostly important to know about joins, there are a few nitty gritties that we should know about.

Let's take the join query we had written a bit earlier:

```
SELECT s1.name, s2.name
FROM students s1
JOIN students s2
ON s1.buddy_id = s2.id;
```

Let's say there is a student that does not have a buddy, i.e., their `buddy_id` is null. What will happen in this case? Will the student be printed?

If you remember what we discussed about CRUD, is NULL equal to anything? Nope. Thus, the row will never match with anything and not get printed. The join that we discussed above is also called `inner join as discussed in Joins 1`. You could have also written that as:

```
SELECT s1.name, s2.name
FROM students s1
INNER JOIN students s2
ON s1.buddy_id = s2.id
```

The keyword INNER is optional. By default a join is INNER join.

As you see, an INNER JOIN doesn't include a row that didn't match the condition for any combination.

Opposite of INNER JOIN is OUTER JOIN. Outer Join will include all rows, even if they don't match the condition. There are 3 types of outer joins:

- Left Join
- Right Join
- Full Join

As the names convey, left join will include all rows from the left table, right join will include all rows from the right table and full join will include all rows from both the tables.

Let's take an example to understand these well:

Assume we have 2 tables: students and batches with following data:

`batches`

batch_id	batch_name
1	Batch A
2	Batch B
3	Batch C

`students`

student_id	first_name	last_name	batch_id
1	John	Doe	1
2	Jane	Doe	1
3	Jim	Brown	null
4	Jenny	Smith	null
5	Jack	Johnson	2

Now let's write queries to do each of these joins:

```
SELECT *
FROM students s
LEFT JOIN batches b
ON s.batch_id = b.batch_id;
```

```
SELECT *
FROM students s
RIGHT JOIN batches b
ON s.batch_id = b.batch_id;
```

```
SELECT *
FROM students s
FULL OUTER JOIN batches b
ON s.batch_id = b.batch_id;
```

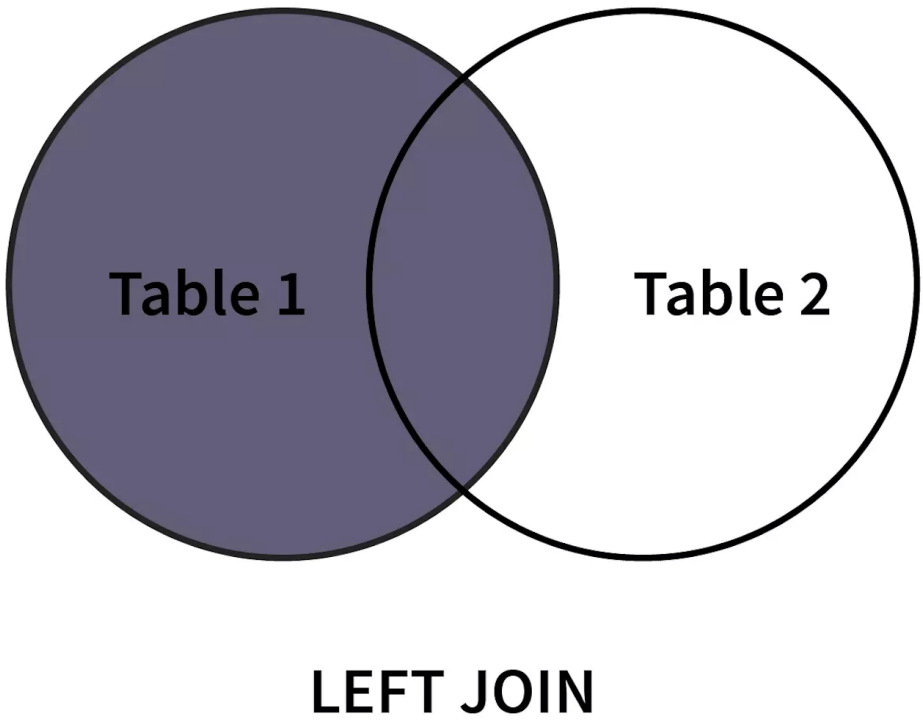
Now let's use different types of joins and tell me which row do you think will not be a part of the join.

`Now let's try to understand each of Outer Joins in depth.`

Left Join

As the names convey, `left join` will include all rows from the `left` table, and include rows from `right table` which matches join condition. If there is any row for which there is no match on right side then it will be considered as `Null`.

> *Venn Diagram*



`General Syntax:`

```
SELECT column_name(s)
FROM table1 LEFT JOIN table2
```

```
ON table1.column_name = table2.column_name;

-- It's same as:

SELECT column_name(s)
FROM table1 LEFT OUTER JOIN table2
ON table1.column_name = table2.column_name;
```

`Example`

Let’s consider two tables of a supermarket set-up. The first table named Customers gives us information about different customers, i.e., their customer id, name, and phone number. Here, CustID is the primary key that uniquely identifies each row. The second table, named Shopping_Details gives us information about items bought by customers, i.e., item id, customer id (referencing the customer that bought the item), item name, and quantity.

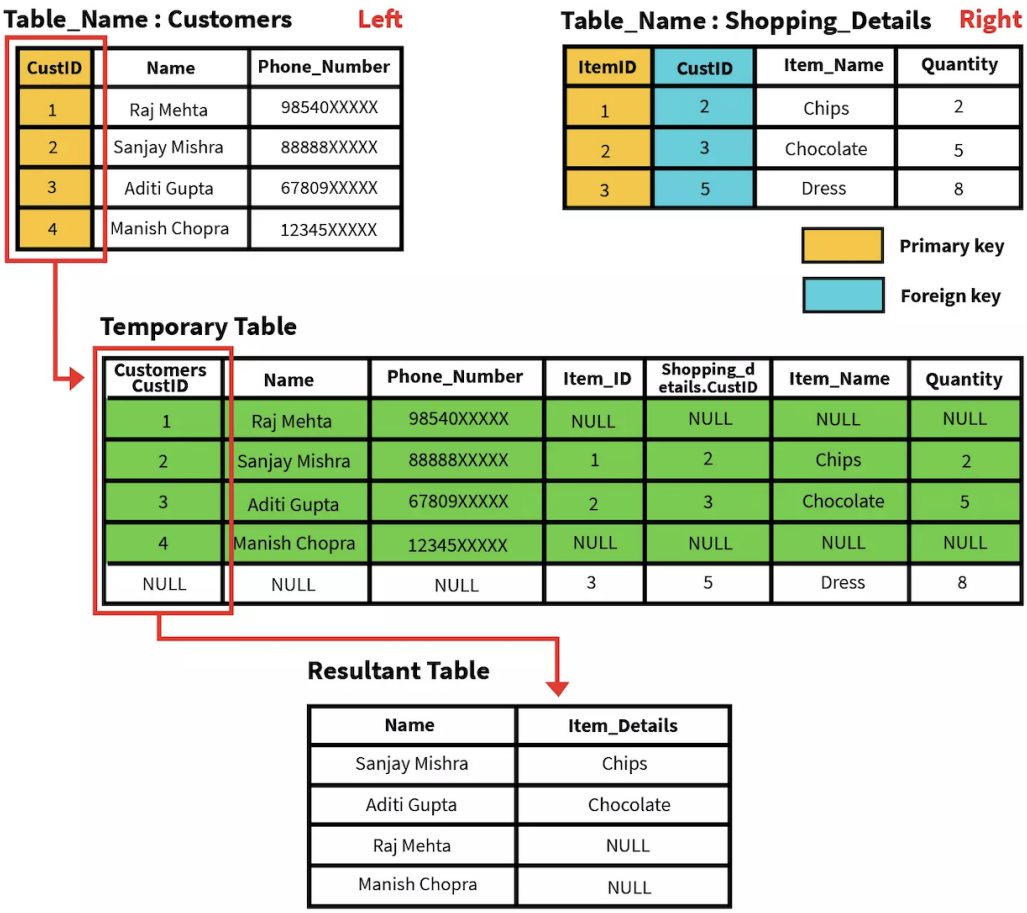
`Problem Statement`

Write a query to display all customers irrespective of items bought or not. Display the name of the customer, and the item bought. If nothing is bought, display NULL.

`Query:`

```
SELECT Customers. Name, Shopping_Details.Item_Name
FROM Customers LEFT JOIN Shopping_Details;
ON Customers.CustID = Shopping_Details.CustID;
```

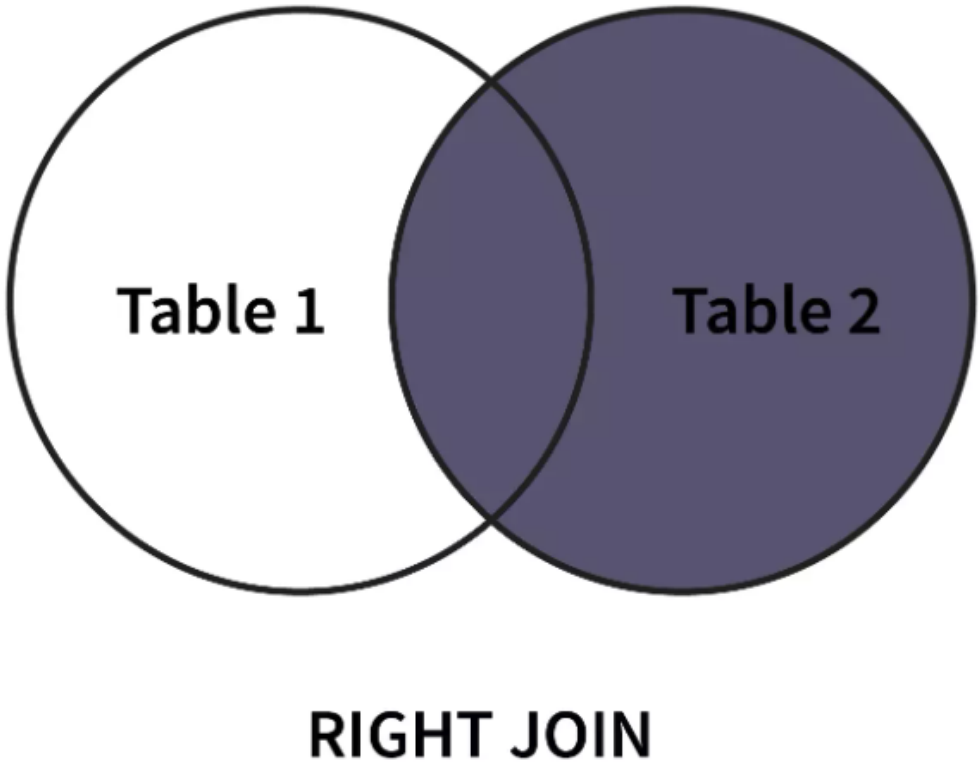
`Infographics:`



Right Join

As the names convey, `right join` will include all rows from the left table, and include rows from `left table` which matches join condition. If there is any row for which there is no match on left side then it will be considered as `Null value`.

> *Venn Diagram*



`General Syntax:`

```
SELECT column_name(s)
FROM table1 RIGHT JOIN table2
ON table1.column_name = table2.column_name;

-- It's same as:

SELECT column_name(s)
FROM table1 RIGHT OUTER JOIN table2
ON table1.column_name = table2.column_name;
```

`Example`

Let’s consider two tables of a supermarket set-up. The first table named Customers gives us information about different customers, i.e., their customer id, name, and phone number. Here, CustID is the primary key that uniquely identifies each row. The second table, named Shopping_Details gives us information about items bought by customers, i.e., item id, customer id (referencing the customer that bought the item), item name, and quantity.

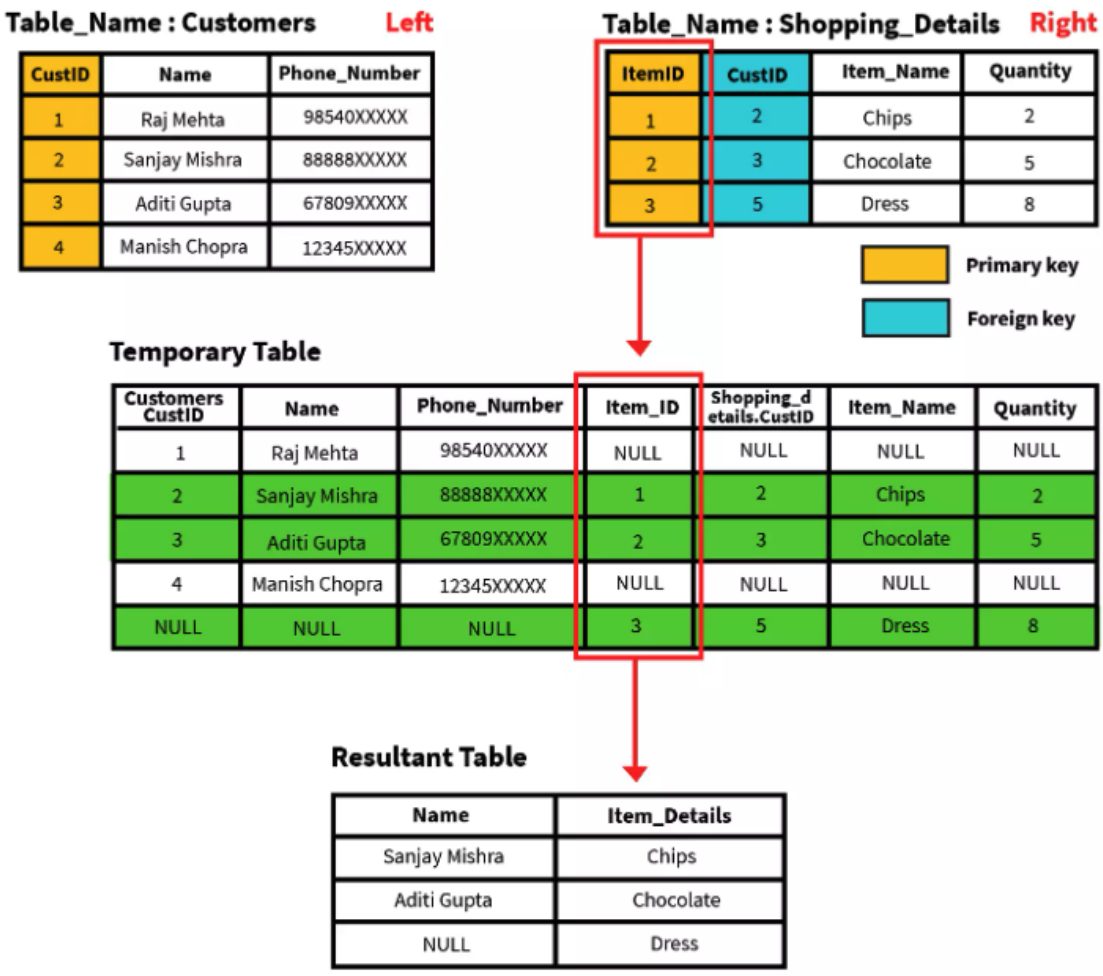
`Problem Statement`

Write a query to get all the items bought by customers, even if the customer does not exist in the Customer database. Display customer name and item name. If a customer doesn’t exist, display NULL.

`Query:`

```
SELECT Customers.Name, Shopping_Details.Item_Name
FROM Customers RIGHT JOIN Shopping_Details;
ON Customers.CustID = Shopping_Details.CustID;
```

`Infographics:`



CustID	Name	Phone_Number
1	Raj Mehta	98540XXXXX
2	Sanjay Mishra	88888XXXXX
3	Aditi Gupta	67809XXXXX
4	Manish Chopra	12345XXXXX

ItemID	CustID	Item_Name	Quantity
1	2	Chips	2
2	3	Chocolate	5
3	5	Dress	8

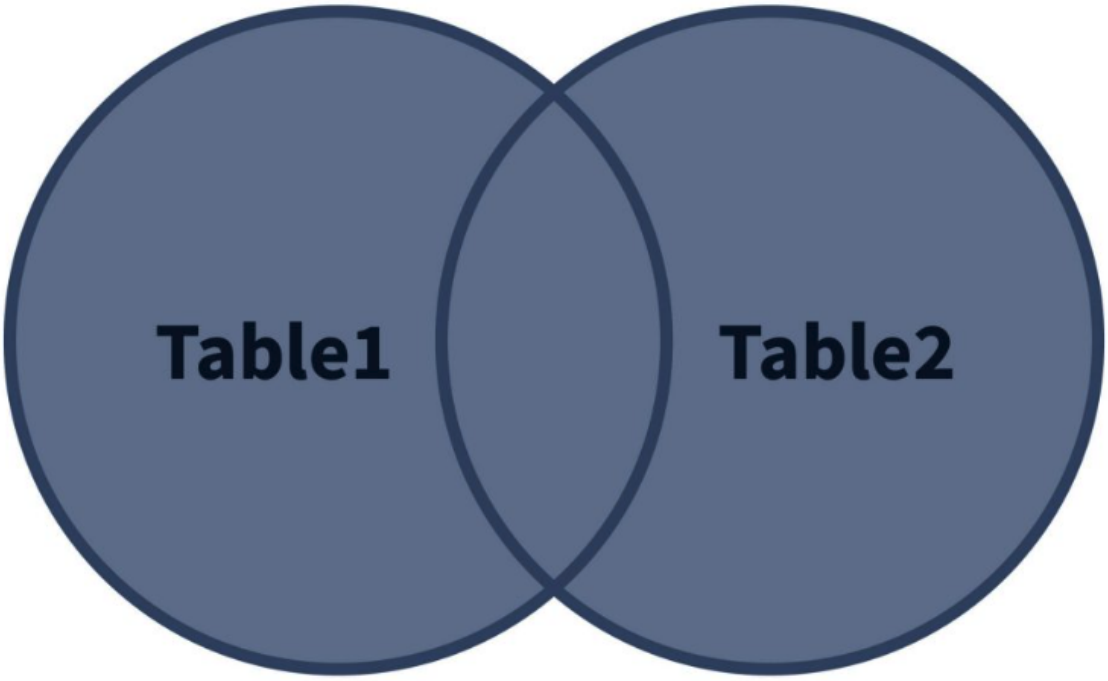
Customers CustID	Name	Phone_Number	Item_ID	Shopping_details.CustID	Item_Name	Quantity
1	Raj Mehta	98540XXXXX	NULL	NULL	NULL	NULL
2	Sanjay Mishra	88888XXXXX	1	2	Chips	2
3	Aditi Gupta	67809XXXXX	2	3	Chocolate	5
4	Manish Chopra	12345XXXXX	NULL	NULL	NULL	NULL
NULL	NULL	NULL	3	5	Dress	8

Name	Item_Details
Sanjay Mishra	Chips
Aditi Gupta	Chocolate
NULL	Dress

Full Outer Join

As the names convey, `Full join` will include all rows from the left table as well as right table, If there is any row for which there is no match on either of the sides then it will be considered as `Null value`.

> *Venn Diagram*



`General Syntax:`

```
SELECT column_name(s)
FROM table1 FULL OUTER JOIN table2
ON table1.column_name = table2.column_name;
```

`Example`

Let’s consider two tables of a supermarket set-up. The first table named Customers gives us information about different customers, i.e., their customer id, name, and phone number. Here, CustID is the primary key that uniquely identifies each row. The second table, named Shopping_Details gives us information about items bought by customers, i.e., item id, customer id (referencing the customer that bought the item), item name, and quantity.

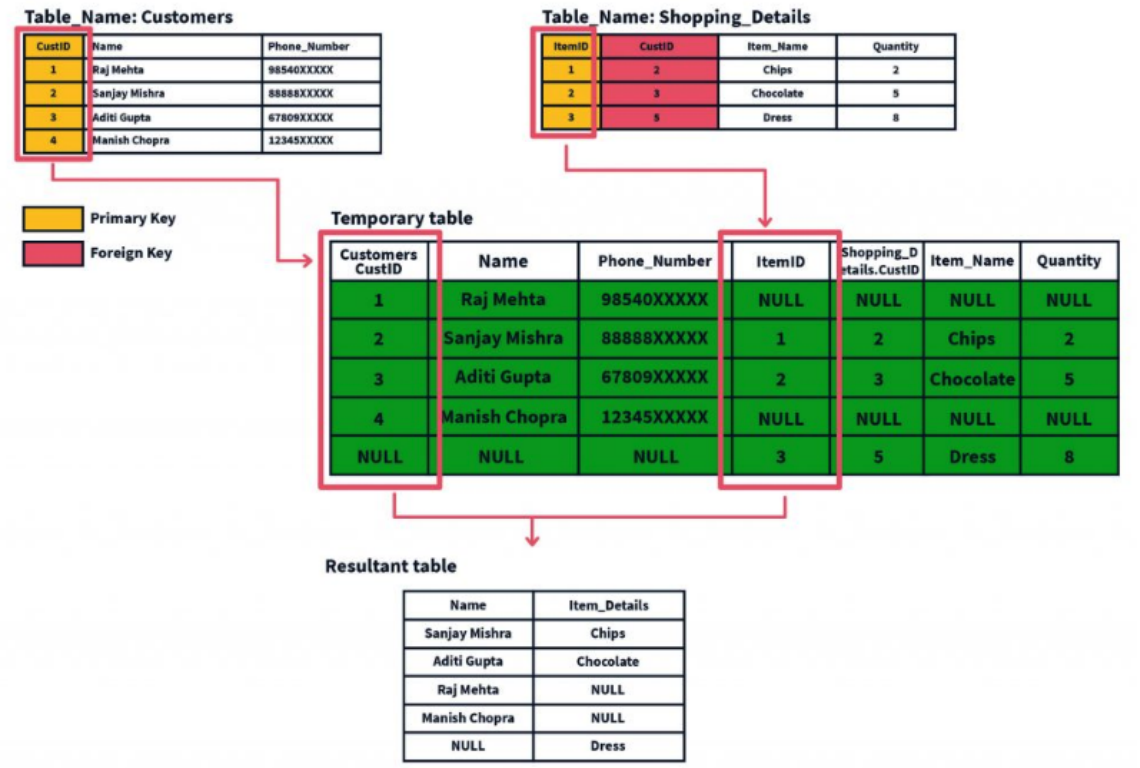
`Problem Statement`

Write a query to provide data for all customers and items ever bought from the store. Display the name of the customer and the item name. If either data does not exist, display NULL.

`Query:`

```
SELECT Customers.Name, Shopping_Details.Item_Name
FROM Customers FULL OUTER JOIN Shopping_Details
WHERE Customer.CustID = Shopping_Details.CustID;
```

Infographics:



When to Use What?

SQL is an essential skill for people looking for Data Engineering, Data Science, and Software Engineering Roles. Joins in SQL is one of the advanced SQL concepts and is often asked in interviews. These questions do not directly state what SQL join to use. Hence, we need to use a four-step analysis before we start forming our SQL query.

1. Identification: Identify tables relating to the problem statement. We also need to identify relations between these tables, the order in which they are connected, and primary and foreign keys.
   - Example: Let’s say we have Tables A and B. Table A and Table B share a relation of Employee Details – Department Details. Table A has three fields – ID, Name, and DeptID. Table B has two fields – DeptID and DeptName. Table A has a primary key ID, and Table B’s primary key is DeptID. Table A and Table B are connected with the foreign key in Table A, i.e., Table B’s primary key, DeptID.
2. Observe: Observe which join will be most suitable for the scenario. This means it should be able to retrieve all the required columns and have the least number of columns that need to be eliminated by the condition.
   - Example: If all values of Table A are required irrespective of the condition depending on Table C, we can use a left outer join on A and C.
3. Deconstruction: Now that we have all requirements to form our query, firstly, we need to break it into sub-parts. This helps us form the query quicker and make our understanding of the database structure quicker. Here, we also form the conditions on the correctly identified relationships.

- Example: You need to present data from Table A and Table B. But Table A's foreign key is Table C's primary key which is Table B's foreign key. Hence breaking down the query into results from Table B and C (let's say Temp) and then common results between its Temp and Table A will give us the correct solution.

4. Compilation: Finally, we combine all the parts and form our final query. We can use query optimization techniques like heuristic optimization, resulting in quicker responses.

> ***Please refer to this link for more practice:***
> *https://www.scaler.com/topics/sql/joins-in-sql/*

Quiz 1

Which of the following rows will NOT be a part of the result set in a `RIGHT` JOIN of the students table on the batches table on batch_id?

Choices

- [ ] [1, John, Doe, 1]
- [ ] [3, Jim, Brown, null]
- [ ] [5, Jack, Johnson, 2]
- [ ] None of the above

Quiz 2

If we perform a RIGHT JOIN of the students table on the batches table on batch_id, which row from the students table will NOT be included in the result set?

Choices

- [ ] [1, John, Doe, 1]
- [ ] [3, Jim, Brown, null]
- [ ] [5, Jack, Johnson, 2]
- [ ] None of the above

Quiz 3

For an INNER JOIN of the students table on the batches table on batch_id, which of the following rows will NOT be included in the resulting set?

Choices

- [ ] [1, John, Doe, 1]
- [ ] [3, Jim, Brown, null]
- [ ] [5, Jack, Johnson, 2]
- [ ] None of the above

Quiz 4

Which row will NOT appear in the resulting set when we perform a FULL OUTER JOIN of the students table on the batches table on batch_id?

Choices

- [ ] [1, John, Doe, 1]
- [ ] [3, Jim, Brown, null]
- [ ] [5, Jack, Johnson, 2]
- [ ] None of the above

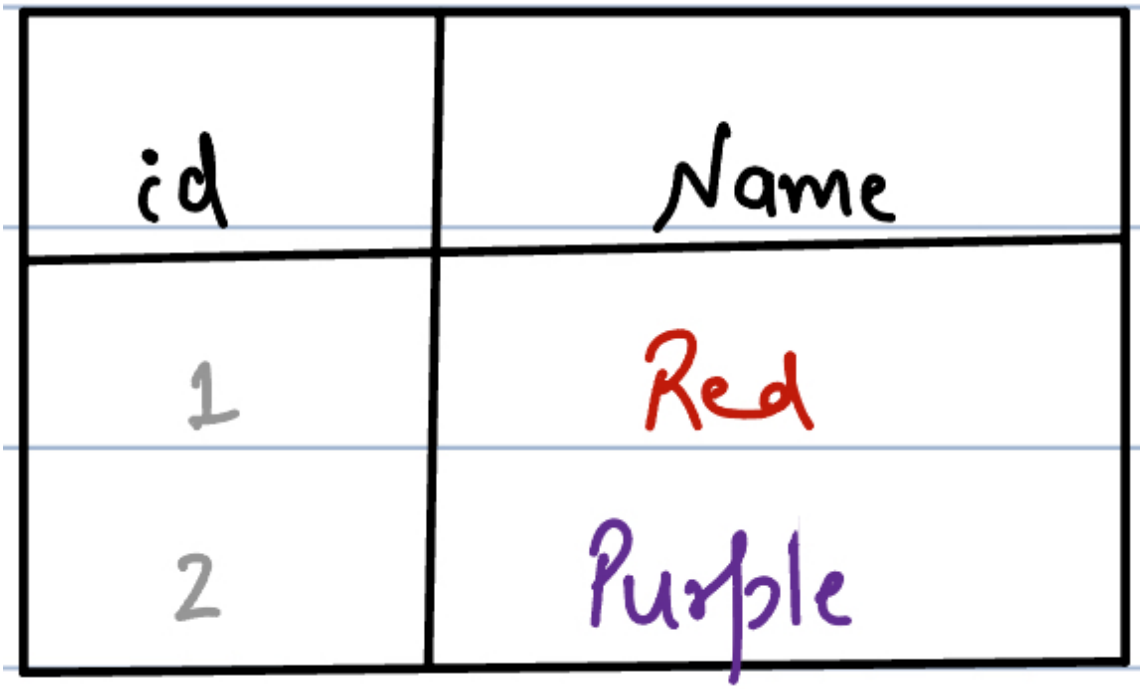
CROSS JOIN

There is one more type of join that we haven't discussed yet. It is called cross join. Cross join is a special type of join that doesn't have any condition. It just combines every row of the first table with every row of the second table. Let's see an example:

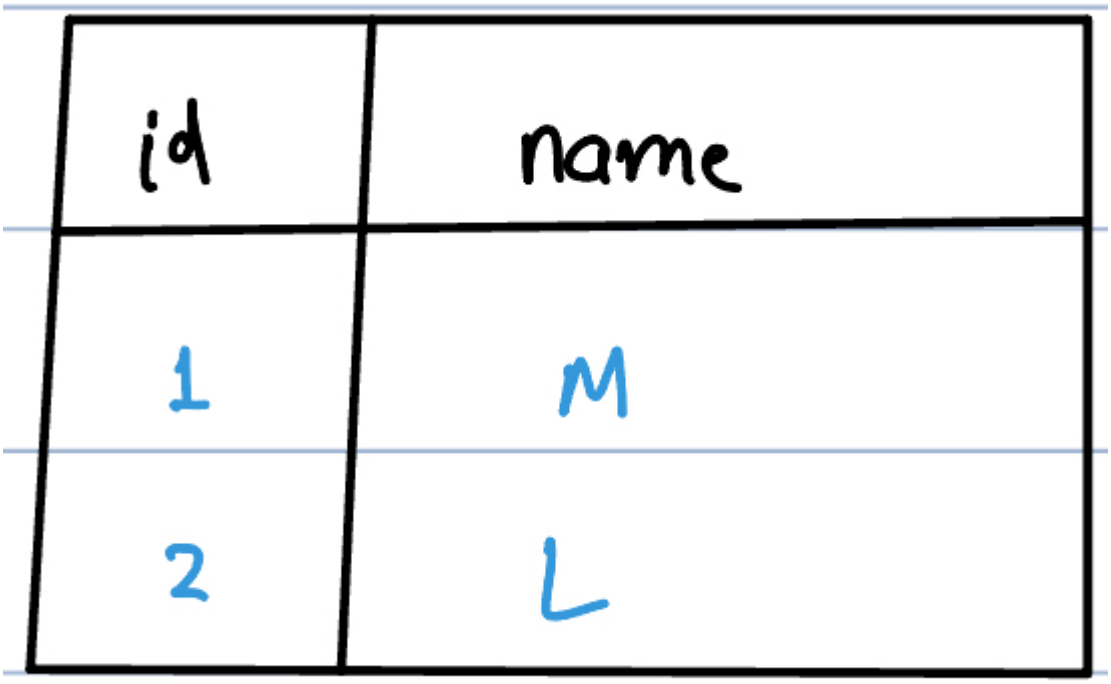
```
SELECT *
FROM students s
CROSS JOIN batches b;
```

Now you may wonder why might someone need this join? For example, in a clothing store's database, one table might have a list of colors, and another table might have a list of sizes. A cross join can generate all possible combinations of color and size.

`colors:`



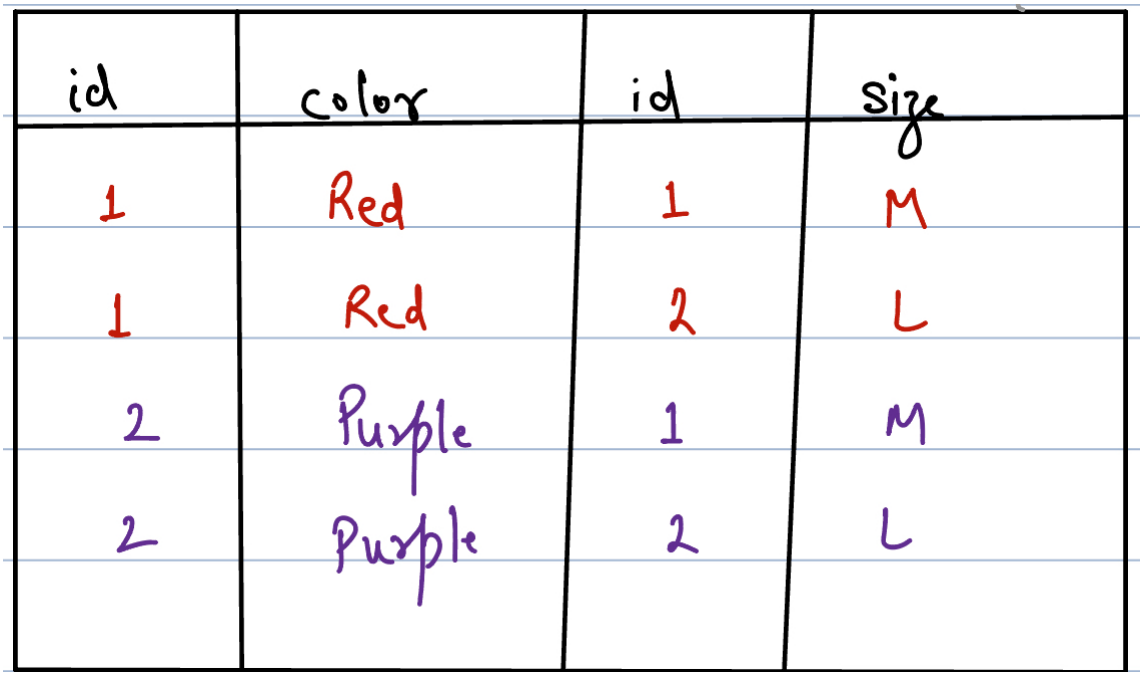
Sizes:



Query:

```
SELECT *
FROM COLORS
CROSS JOIN SIZES;
```

`RESULTANT TABLE:`



id	color	id	size
1	Red	1	M
1	Red	2	L
2	Purple	1	M
2	Purple	2	L

Cross join produces a table where every row of one table is joined with all rows of the other table. So, the resulting table has `N*M` rows given that the two tables have `N` and `M` rows.

That's pretty much all different kind of joins that exist. There are a few more syntactic sugars that we can use to write joins. Let's see them:

Now, let's understand some syntactical sugars and tiny topics:

USING

Let's say we want to join 2 tables on a column that has the same name in both the tables. For example, in the students and batches table, we want to join on the column `batch_id` . We can write the join as:

```
SELECT *
FROM students s
JOIN batches b
ON s.batch_id = b.batch_id;
```

But there is a shorter way to write this. We can write this as:

```
SELECT *
FROM students s
JOIN batches b
USING (batch_id);

-- Here the above tables will be joined based on equality of batch_id
```

> *Note: Using is a syntactical sugar used to write queries with ease.*

NATURAL JOIN

Many times it happens that when you are joining 2 tables, they are mostly on the columns with same name. If we want to join 2 tables on all the columns that have the same name, we can use NATURAL JOIN. For example, if we want to join students and batches table on all the columns that have the same name on both sides, we can write:

```
SELECT *
FROM students s
NATURAL JOIN batches b;

-- In above tables we have only batch_id as common column in both of the tables with same name.
```

IMPLICIT JOIN

There is one more way to write joins. It is called implicit join. In this, we don't use the JOIN keyword. Instead, we just write the table names and the condition. For example, if we want to write the join query that we wrote earlier as implicit join, we can write:

```
SELECT *
FROM students s, batches b;

-- Above query will work as cross joins behind the scenes.
```

> Note: Behind the scenes, this is same as a cross join.

Join with WHERE vs ON

Let's take an example to discuss this. If we consider a simple query:

```
SELECT *
FROM A
JOIN B
ON A.id = B.id;
```

In pseudocode, it will look like:

```
ans = []

for row1 in A:
    for row2 in B:
        if (ON condition matches):
            ans.add(row1 + row2)

for row in ans:
    print(row.id, row.id)
```

Here, the size of intermediary table (`ans`) will be less than `n*m` because some rows are filtered.

We can also write the above query in this way:

```
SELECT *
FROM A, B
WHERE A.id = B.id;
```

The above query is nothing but a CROSS JOIN behind the scenes which can be written as:

```
SELECT *
FROM A
CROSS JOIN B
WHERE A.id = B.id;
```

Here, the intermediary table `A CROSS JOIN B` is formed before going to WHERE condition.

In pseudocode, it will look like:

```
ans = []

for row1 in A:
    for row2 in B:
        ans.add(row1 + row2)

for row in ans:
    if (WHERE condition matches):
        print(row.id, row.id)
```

The size of `ans` is always `n*m` because table has cross join of A and B. The filtering (WHERE condition) happens after the table is formed.

From this example, we can see that:

1. The size of the intermediary table (`ans`) is always greater or equal when using WHERE compared to using the ON condition. Therefore, joining with ON uses less internal space.
2. The number of iterations on `ans` is higher when using WHERE compared to using ON. Therefore, joining with ON is more time efficient.

In conclusion,

1. The ON condition is applied during the creation of the intermediary table, resulting in lower memory usage and better performance.
2. The WHERE condition is applied during the final printing stage, requiring additional memory and resulting in slower performance.
3. Unless you want to create all possible pairs, avoid using CROSS JOINS.

UNION

Sometimes, we want to print the combination of results of multiple queries. Let's take an example of the following tables:

`students` | id | name | |----|------|

`employees` | id | name | |----|------|

`investors` | id | name | |----|------|

You are asked to print the names of everyone associated with Scaler. So, in the result we will have one column with all the names.

We can't have 3 SELECT name queries because it will not produce this singular column. We basically need SUM of such 3 queries. Join is used to stitch or combine rows, here we need to add the rows of one query after the other to create final result.

UNION allows you to combine the output of multiple queries one after the other.

```
SELECT name FROM students
UNION
SELECT name FROM employees
UNION
SELECT name FROM investors;
```

Now, as the output is added one after the other, there is a constraint: Each of these individual queries should output the same number of columns.

Note that, you can't use ORDER BY for the combined result because each of these queries are executed independently.

UNION outputs distinct values of the combined result. It stores the output of individual queries in a set and then outputs those values in final result. Hence, we get distinct values. But if we want to keep all the values, we can use UNION ALL. It stores the output of individual queries in a list and gives the output, so we get all the duplicate values.

Difference Between JOIN and UNION in SQL:

SQL JOIN	SQL UNION
The SQL JOIN is used when we have to extract data from more than one table.	The SQL UNION is used when we have to display the results of two or more SELECT statements.
In the case of SQL JOINS, the records are combined into new columns.	In the case of SQL UNION, the records are combined into new rows.
The SQL JOINS facilitates the joining of tables vertically.	The SQL UNION facilitates the connection of tables vertically.
The SQL JOINS are used to produce the given table's intersection.	The SQL UNION is used to produce the given table's conjunction.
The duplicate values can exist in SQL JOINS.	The duplicate values are removed by default in SQL UNION
To use SQL JOINS the two given tables need to have at least one column present within them.	To use SQL UNION the domain of the columns along with their attributes needs to be the same.

> *Conclusion*

1. The SQL JOIN is used to combine two or more tables.
2. The SQL UNION is used to combine two or more SELECT statements.
3. The SQL JOIN can be used when two tables have a common column.
4. The SQL UNION can be used when the columns along with their attributes are the same.

That's all about Union and Joins! See you next time. Thanks.

Solution to Quizzes:

> *-- Quiz1: Option D (None of the above) Quiz2: Option B [3, Jim, Brown, null] Quiz3: Option B [3, Jim, Brown, null] Quiz4: Option D (None of the above) --*