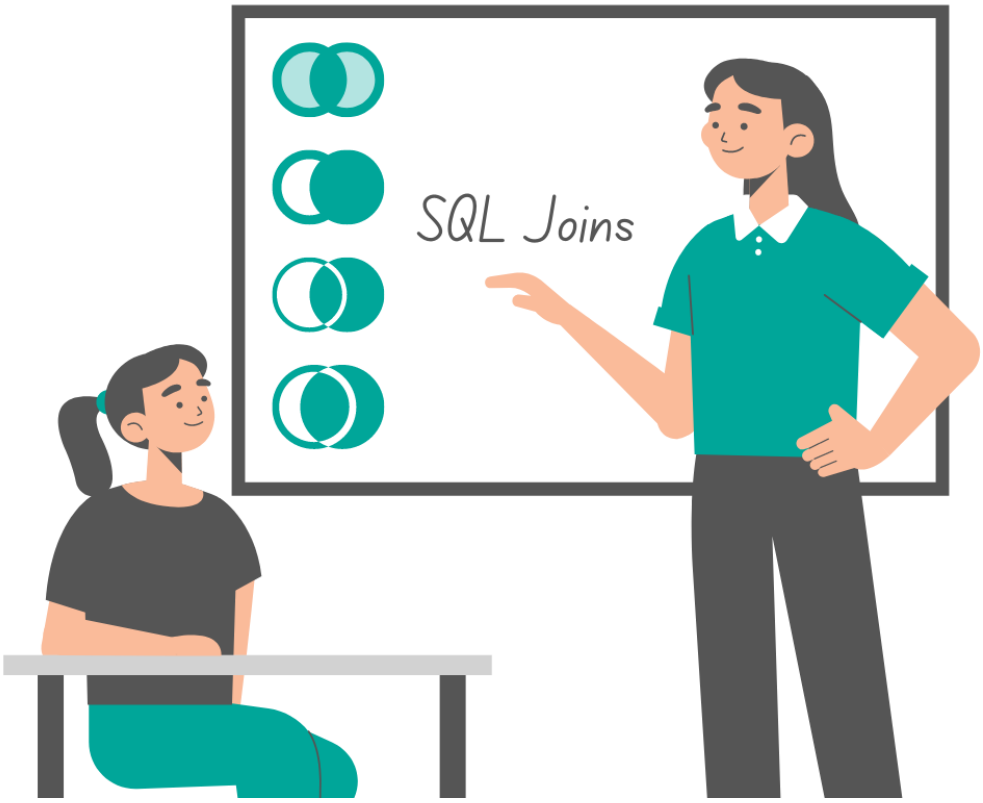
Different Types of SQL Joins that You Must Know

In this article, we will discuss different types of SQL JOINs and show practical examples to help you wrap your head around a variety of JOINs in SQL.



Relational databases store information about real world objects in many logically separate tables. SQL gives us the ability to define relationships and work with data from different tables.

JOINs are an essential feature of SQL that allow you to work with data from multiple tables. The importance of this feature can not be overstated, so employers often ask questions to test a candidate's knowledge of JOINs in SQL.

In this article, we will discuss the different types of SQL JOINs and show practical examples to help you wrap your head around a variety of JOINs in SQL.

Why Would Anyone Need JOINs?

In SQL, the most practical approach is to describe different objects in separate tables.

For example, it's possible to have one table that contains information about users as well as all their orders. However, this table will have too many columns and will be difficult to maintain.

It's easier to have a **users** table, which would contain essential user information, and have a separate **orders** table, that contains information about all the orders. One of the columns in the **orders** table could be used to identify the user who placed the order.

Not only is it easier, but also the database normalization would require this huge table to be split into two tables.

In a relational database, JOINs allow you SELECT and work with data from two tables. To do that, you have to specify the shared dimension (in simpler words, shared column) between two tables. If you look at two tables below, both tables contain columns that specify the identity of the user. In the **users** column that is **id,** and in the **orders** column that is **user_id**.

JOIN gives us easy access to both - information about the order and the user who placed that order. We can use this data to do analysis, aggregation, or use it for any other purpose.

users

id	email	full_name
1	george@gmail.com	George Harrison
2	paul@gmail.com	Paul McCartney

orders

order_id	user_id	order_size
1	1	10
2	2	21
3	2	25
4	3	50

This is how relational databases store information about complex real-world objects. Tables provide essential information about one object. You can use different types of JOINs in SQL to work with related tables.

Because of their importance, employers look for data scientists who are fluent in different types of JOINs. Interviewers often ask questions to test a candidate's knowledge of this important SQL feature. To find out whether or not you're up to the task, read this article about SQL JOIN Interview Questions.

When doing data analysis and aggregation, sometimes you need to work with data from two related tables. For that, you can use JOIN tables based on a common dimension.

In simple words, **Dimension** is a description of one specific characteristic of a thing. If we had a box, its dimensions would be: length, width, height, what's inside, and so on.

How the JOIN Works in SQL

You can work with values from two tables by defining a relationship between them. In this case, you need to specify a common dimension (a shared column that contains the same values).

In this case, that is the **id** of users. In the **users** table, each record has its own **id**. In the **orders** table, the **user_id** column refers to the user who placed the order. Then you can use SELECT to take data from columns of two tables.

Here is a simple query to JOIN two tables:

```
SELECT *
FROM users
JOIN orders
ON users.id = orders.user_id
```

As you can see, we use the **ON** keyword to specify columns from two tables that can contain matching values.

This is the most basic example of how to perform a JOIN in SQL. If you want to work with more data, read the article which describes how to Join 3 or More Tables in SQL.

What Is a Primary Key and Foreign Key?

In this case, the identity of the user is the **dimension** that is featured in both tables.

Key that is used to identify each row in a table is called a **primary key.** Every row in the **users** table describes an individual user. Therefore, the **id** column that identifies each user (this value is unique for each row) is the primary key.

A value that specifies a connection between two tables is a **foreign key**. In the **orders** table, **user_id** is a **foreign key**, because it does not determine the identity of the order itself, but identifies the user who placed the order.

Different Types of JOINs in SQL overview

Now, let's look at different types of JOINs in SQL. The main difference between these is in how they combine data from two tables.

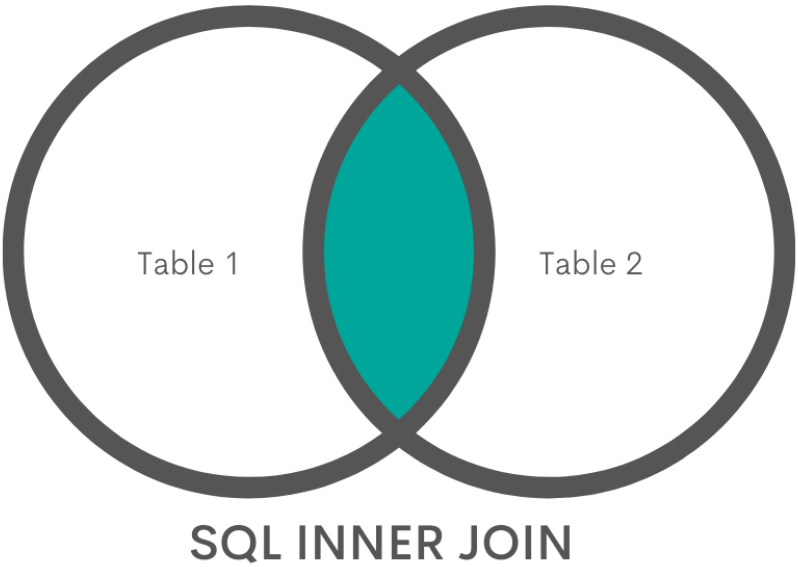
These are the 5 most important SQL join types you should know about:

- INNER JOIN
- LEFT JOIN
- RIGHT JOIN
- FULL OUTER JOIN
- CROSS JOIN

UNION operator allows you to work with sets of data (results of a query). UNION is fairly similar to JOINs, so we will discuss it as well.

INNER JOIN

INNER JOIN is the most commonly used type of JOINs in SQL. When writing JOIN, SQL will perform an inner JOIN by default. After performing an INNER JOIN, you will gain access to rows only when there is an overlap on a specified column.



In simple words, INNER JOIN will try to find if any of the users placed an order, by matching values from the **id** column to the values in the **user_id** column of the **orders** table.

If users can be identified as having placed an order, you will have access to columns from both tables. Rows that do not match will be discarded.

Let's look at an example query:

```
SELECT *
FROM users
JOIN orders
ON users.id = orders.user_id
```

The output will include orders placed by users that are listed in the table. Not orders placed by other individuals, or records of users who have not placed an order.

The **id** column contains an unique number to identify each user. It is the **primary key** for the **users** table. However, one user might place an order multiple times, so multiple orders can have the same value in the **user_id** column.

We use the ON statement, to specify the dimension on which two tables overlap. In other words, individual columns in two tables that contain the same value (id of the user).

The syntax for the ON statement is simple: specify the table, followed by a dot and name of the column that overlaps with a column from another table. We use the equation symbol to specify that we are looking for a match in the column of another table. Then we use the same syntax to specify the column.

id	email	full_name	order_id	order_size
1	**george@gmail.com**	George Harrison	1	10
1	**george@gmail.com**	George Harrison	2	50

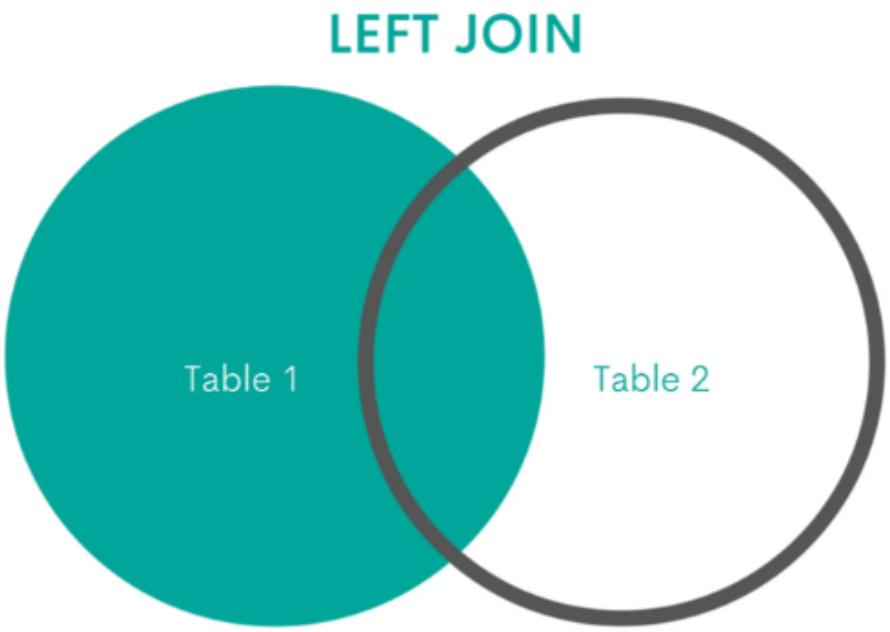
Writing SELECT *, like we do in this example, selects all columns. However, there is one exception - since **id** and **user_id** refer to the same value, the output will include only one - the **id** column.

The final output will contain **id, email, full_name, order_id,** and **order_size** columns. As an SQL developer, you are free to select as many or as few columns as you'd like.

Our output shows the case when one record in the **users** table is related to multiple records in the **orders** table. In simple words, when one user has placed multiple orders. The output of INNER JOIN can contain multiple rows that describe one user, but different orders.

Left JOIN

Left JOIN can be useful when you don't want to discard any rows from one table, but also want to have data from another table when it's possible. SQL is written from left to right, so the word LEFT refers to the first table, which is written on the left of the other table.



When there is an overlap between two tables, LEFT JOIN will return values from both tables. For example, if a user placed an order, LEFT JOIN will return the details from the **users** table as well as **orders** table.

If a certain user did not place an order (there is no overlap), LEFT JOIN will still keep rows to describe these users. However, since they do not have a corresponding value for columns of the second table, these columns' values will be null.

Here is the query:

```
SELECT id, order_size
FROM users
LEFT JOIN orders
ON users.id = orders.user_id
```

The output would be:

id	order_size
1	10
2	21

2	25
3	50
4	null

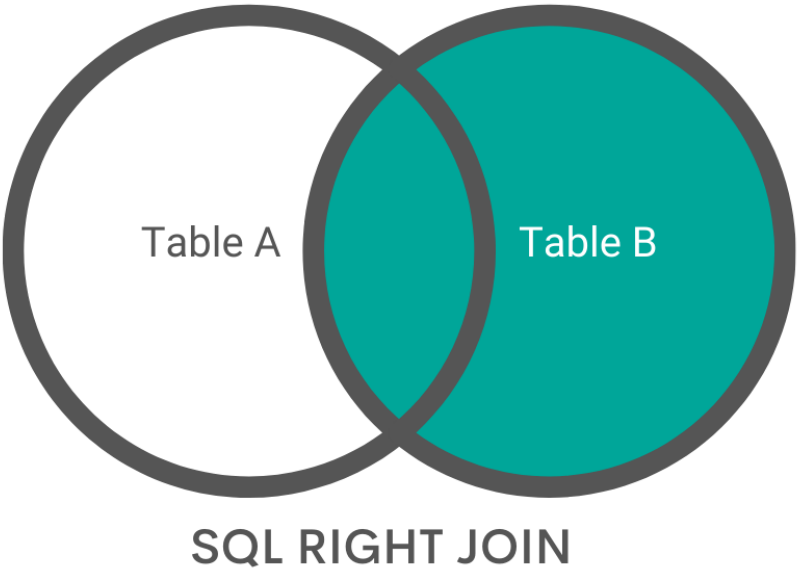
Let's say we LEFT JOIN two tables based on the same shared dimension - user's identity. In this scenario, the first, **users** table contains a row that represents the user with the **id** of 4, but there are no orders for this user.

Because the **users** table is considered the first, LEFT JOIN includes all users in the final output. But since some users have not placed an order, there will be no **order_size** value for their **id**, so its **order_size** column will be null. The same principle would apply to all columns from the second, **orders** table.

LEFT JOIN may be useful to include users with 0 sales to aggregate total number of users, or get the average order per user, counting even those who have not purchased anything.

RIGHT JOIN

SQL code is written from left to right, so the second table is on the right side of the query. For this reason this SQL join type always takes values from the second table.



In essence, this feature is the opposite of LEFT JOIN. For this reason, experienced developers sometimes change the order of tables to get the same result with LEFT JOIN.

Let's take a look at a query that uses RIGHT JOIN:

```
SELECT id, order_size, email
FROM users
RIGHT JOIN orders
ON users.id = orders.user_id
```

If one of the orders was placed by the user with the id of 4, and such user was not found in the **users** table, the output would be:

id	order_size	email
1	10	**george@gmail.com**
2	21	**paul@gmail.com**
2	25	**paul@gmail.com**

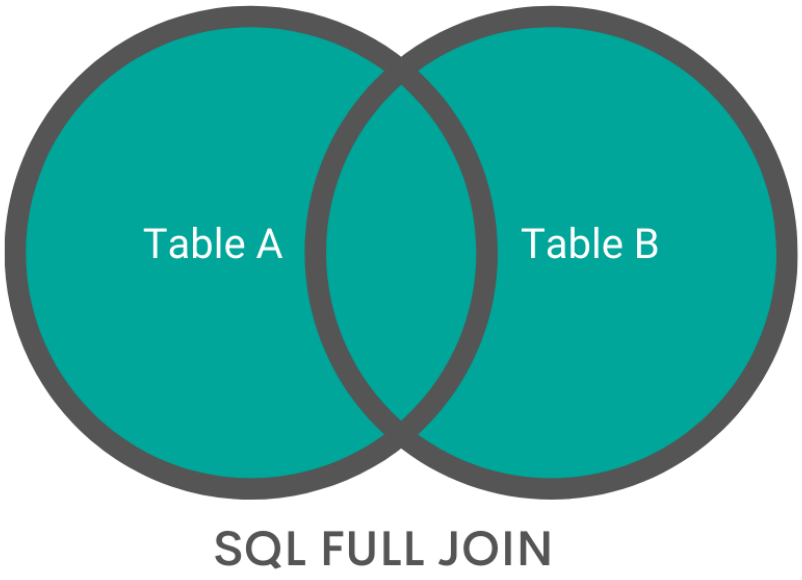
3	50	**john@gmail.com**
4	50	null

Rows that match between two tables will have all columns with full information. Rows from the second table that did not have a match in the first table will have some columns with a null value.

Performing this SQL JOIN type checks if the values from the specified column of the second table are available in the first. Some records in the second table will be absent from the first. In this case, columns will be filled with a null value.

FULL OUTER JOIN

One of the misunderstood types of JOINs in SQL. It takes columns and rows from both tables, even those that do not match. When two tables overlap, FULL OUTER JOIN output values from both columns.



Here's an example of a query that performs FULL OUTER JOIN:

```
SELECT id, order_size, email
FROM users
RIGHT JOIN orders
ON users.id = orders.user_id
```

You can think of an OUTER JOIN as both LEFT JOIN and RIGHT JOIN performed together.

Let's imagine in both these tables we have one record that is not found in the other. The output would be:

The result of FULL OUTER JOIN operation:

id	email	full_name	order_id	order_size
1	george@gmail.com	George Harrison	1	10
2	paul@gmail.com	Paul McCartney	2	21
2	paul@gmail.com	Paul McCartney	3	25
3	john@gmail.com	John Lennon	4	50
4	ringo@gmail.com	Ringo Starr	null	null
5	null	null	6	200

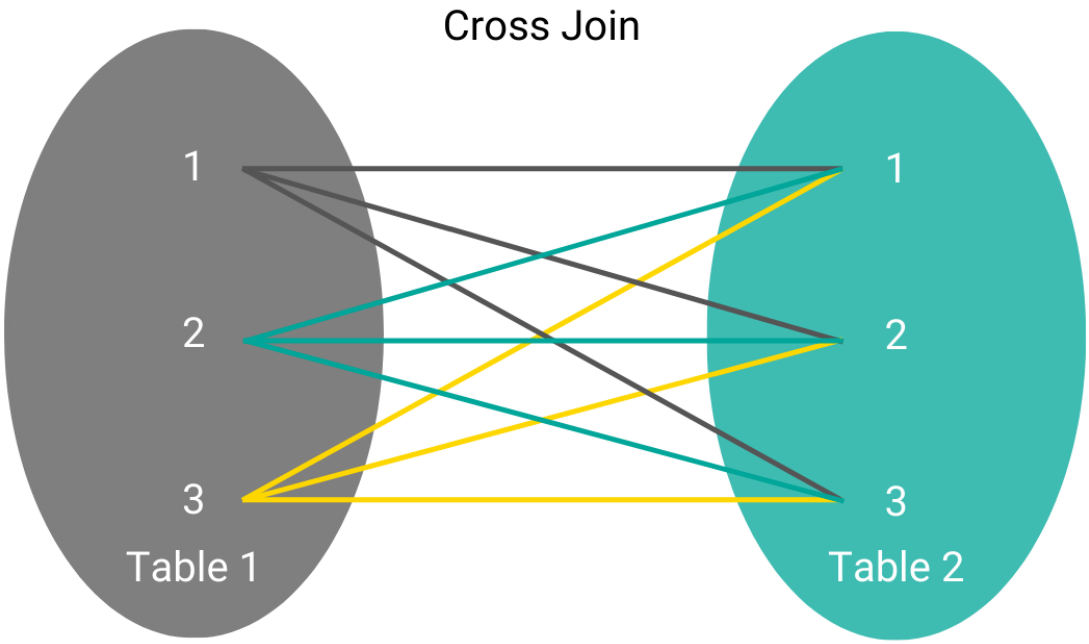
The records that are featured in both tables will contain information in columns from both tables. Rows that can be found in one table but not another, will be in the combined output, but columns from the 'other' table will have a *null* value.

In this case, the fourth record is present in the **users** table, but not in the **orders** table. For this reason, columns from this table have a value of **null**. Then we have the opposite - a row from the **orders** table that is absent from the **users** table.

If you need to include users and orders that do not appear in both tables, you need to use FULL OUTER JOIN.

Cross JOIN

CROSS JOIN generates all possible combinations between all the rows in two tables. They are typically used to get the number of possible combinations of rows.



Let’s imagine our previous example of two tables. If we ran a query to perform a cross JOIN, it would look like this:

```
SELECT *
FROM users
CROSS JOIN orders
```

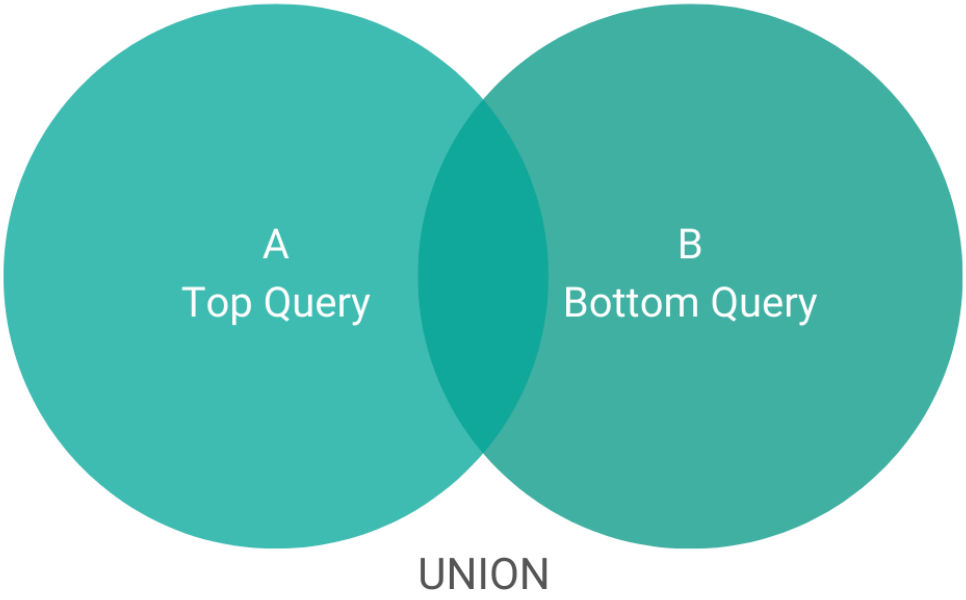
And would return:

id	email	full_name	user_id	order_size	order_id
1	george@gmail.com	George Harrison	1	10	1
1	george@gmail.com	George Harrison	2	21	2
1	george@gmail.com	George Harrison	2	25	3
2	paul@gmail.com	Paul McCartney	1	10	1
2	paul@gmail.com	Paul McCartney	2	21	2
2	paul@gmail.com	Paul McCartney	2	25	3
3	john@gmail.com	John Lennon	1	10	1
3	john@gmail.com	John Lennon	2	21	2
3	john@gmail.com	John Lennon	2	25	3

We get a larger amount of rows because CROSS JOIN generates all possible combinations of rows between two tables. To help you differentiate, rows from the first table are colored in red, yellow and green. The ones from the **orders** table are colored in blue, purple and pink.

Unions

UNION is a set operator that works similarly to JOINs. It stacks two query results on top of each other. If you're selecting multiple columns from each table, then the order of columns and their value types must be similar.



In unions, the order of columns defines how columns are combined. Let's look at this query as an example:

```
SELECT id, balance
FROM users
UNION ALL
SELECT user_id, order_size
FROM orders
```

The output would be:

id	balance
1	200
2	100
3	400
1	50

2	20

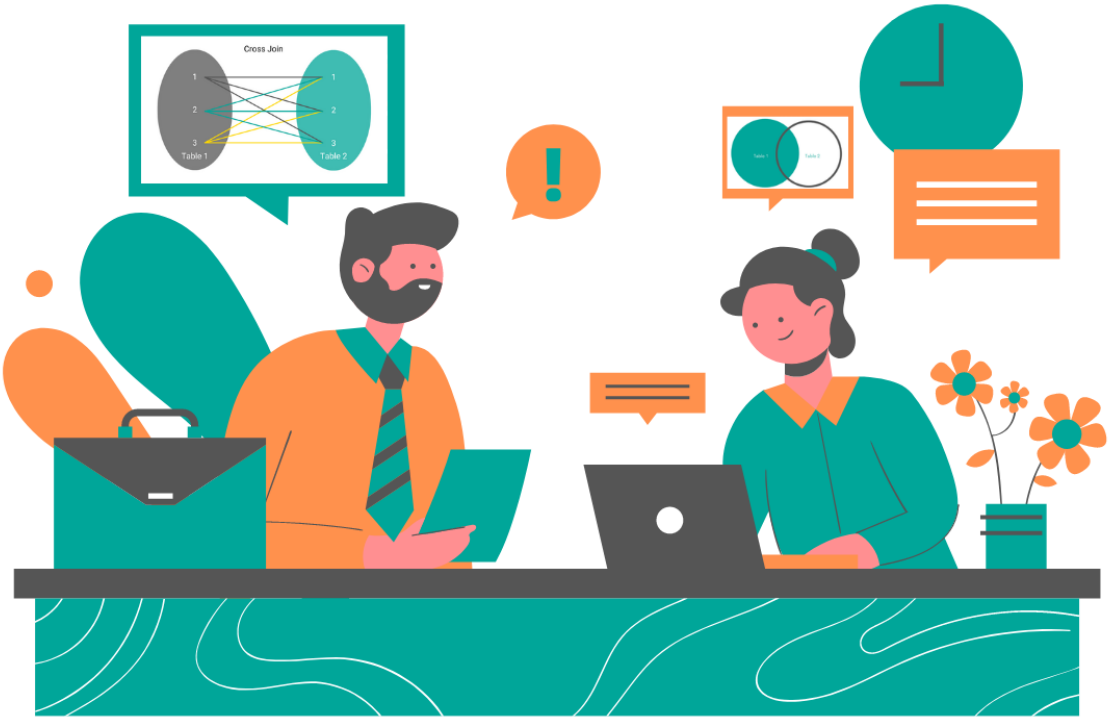
Here, we perform an UNION on two columns. The first pair - **id** from the **users** table, and **user_id** from **orders** table, contain the same values. However, second columns - **balance** and **order_size** describe different things.

For example, if one column in the first table is a text, and a column from the second table is a number, and you try to perform an UNION on queries that return these columns, it will throw an error. This happens because you're trying to combine different data types.

If you try to union multiple columns from two tables, it's important to pay attention to the order of columns. Even if columns have matching data types, the order of columns has to match as well.

UNIONs are ideal when you have two similar tables that you need to combine into one.

How to Use These SQL JOIN Types in the Interview Questions



Now that you understand what each SQL JOIN type will do; let's see how that works in the interview questions.

INNER JOINs

Let’s go through the solution to a Dropbox interview question, where you have to JOIN two tables based on shared values in a certain column.

Salaries Differences

Dropbox Easy Interview Questions ID 10308 70 8

Write a query that calculates the difference between the highest salaries found in the marketing and engineering departments. Output just the absolute difference in salaries.

Link to the question: https://platform.stratascratch.com/coding/10308-salaries-differences

Sounds simple enough, except when you look at available data and discover that there are two tables:

db_employee

id:	int
first_name:	varchar
last_name:	varchar
salary:	int
department_id:	int
email:	datetime

db_dept

id:	int
department:	varchar

The **department_id** column of the **db_employee** table only references the **id** of the department. According to the description of the question, we need to specify departments by name - ‘marketing’ and ‘engineering’.

When solving questions that require you to JOIN two tables, your first step should be to identify the shared dimension. In this case, the **department_id** and **id** columns both contain values that identify a department. Fortunately, the **department** column of the **db_dept** table also contains the name of the apartment.

With that in mind, let’s look at the code:

```
SELECT
  ABS((SELECT max(salary)
   FROM db_employee emp
   JOIN db_dept dept ON emp.department_id = dept.id
   WHERE department = 'marketing') -
  (SELECT max(salary)
   FROM db_employee emp
   JOIN db_dept dept ON emp.department_id = dept.id
   WHERE department = 'engineering')) AS salary_difference
```

The question asks us to find the absolute difference between the highest salaries of two departments. We have to simply find the highest salary from each department, subtract them and return the absolute value of the result.

Finding the highest salary in one department requires us to use INNER JOIN. We use it to return only rows that refer to employees in the specified department.

After we combine data from two tables, we can use the additional WHERE statement to only keep records of employees that work in the ‘marketing’ department.

We perform the same operation to find the highest salary for the second department. However, this time the WHERE statement keeps records of employees from the ‘engineering’ department.

Once we find the difference, we output it as **salary_difference**:

Output

salary_difference
2400

LEFT JOIN

In this Yelp interview question, we use LEFT JOIN to find the number of businesses open on specific time-slots of Sunday.

Businesses Open On Sunday

Yelp Medium General Practice ID 10178 3 4

Find the number of businesses that are open on Sundays. Output the slot of operating hours along with the corresponding number of businesses open during those time slots. Order records by total number of businesses opened during those hours in descending order.

Link to the question:
https://platform.stratascratch.com/coding/10178-businesses-open-on-sunday

For this question, available data is split between two tables. We need to use JOINs to work with data from both tables. Let’s take a look:

yelp_business_hours

business_id:	varchar
monday:	varchar
tuesday:	varchar
wednesday:	varchar
thursday:	varchar
friday:	varchar
saturday:	varchar
sunday:	datetime

yelp_business

business_id:	varchar
name:	varchar
neighborhood:	varchar
address:	varchar
city:	varchar
state:	varchar
postal_code:	varchar
latitude:	float
longitude:	float
stars:	float
review_count:	int
is_open:	int
categories:	varchar

Our final output is going to be the list of business hours open on Sunday. For this reason, we use LEFT JOIN to keep all possible business hours on Sunday.

```
SELECT
    sunday,
    count(*) as total_business
FROM yelp_business_hours business_hours
LEFT JOIN yelp_business business
ON business_hours.business_id = business.business_id
WHERE sunday is NOT NULL
```

```
            and is_open = 1
GROUP BY sunday
ORDER BY total_business DESC
```

The most important section of this query is the one where we JOIN two tables. We leave a space between the current name of the table and its alias.

We use the ON statement to merge data based on shared dimension - identity of the business. LEFT JOIN keeps all rows from the **yelp_business_hours** table. We need to do this to count the number of all businesses.

We use the WHERE statement to filter rows based on their value in the **sunday** column. Our conditions are that the values in this column must not be **NULL** and the **is_open** column should be 1, which indicates that the business is operational.

To illustrate the purpose of the WHERE statement, let's preview the **yelp_business_hours** table:

Output View the output in a separate browser tab

	monday	tuesday	wednesday	thursday	friday	saturday	sunday
NNF4Eh7_mlRg	10:30-22:00	10:30-22:00	10:30-22:00	10:30-22:00	10:30-22:00	10:30-22:00	X
:dMgOUXgr6rA	08:00-16:00	08:00-16:00	08:00-16:00	08:00-16:00	08:00-16:00	08:00-12:00	X
‹CfZPdnmU9tNA							X
dTJqvY7ksGCA	9:00-23:00	9:00-23:00	9:00-23:00	9:00-23:00	9:00-23:00	11:00-22:00	11:00-22:00
ınZ1PaGVRGBfQ	18:00-23:00	18:00-23:00	18:00-23:00	18:00-23:00	18:00-23:00	17:00-23:00	X
pSR4iprqTw8A							X

As you can see, for some businesses, the value of columns (the days of week) are empty (null). Our WHERE statement is going to weed out businesses whose **sunday** column is null (empty).

At this point, we have separate rows for every business that operates on a specific time-slot on sunday. We use the GROUP BY statement to aggregate the number of businesses for each time slot. Finally, we use the ORDER BY statement to arrange them in a descending order.

If we run the query, this is the final output we are going to get:

Output

sunday	total_business
00:00-00:00	4
9:00-21:00	3
10:00-19:00	2
11:00-22:00	2
11:00-00:00	2
11:00-18:00	2
11:30-22:30	2
11:00-23:00	2
06:00-20:30	1
13:00-17:00	1
9:00-18:00	1
10:00-16:00	1
06:30-10:00	1
11:30-20:00	1
9:00-23:00	1

RIGHT JOIN

Next, we have a question from Amazon.

Find the number of customers without an order

Link to the question:
https://platform.stratascratch.com/coding/10089-find-the-number-of-customers-without-an-order

We need to work with data from two tables:

orders

id:	int
cust_id:	int
order_date:	datetime
order_details:	varchar
total_order_cost:	int

and **customers**

id:	int
first_name:	varchar
last_name:	varchar
city:	varchar
address:	varchar
phone_number:	varchar

We use the RIGHT JOIN to merge data from two tables: **orders** (with an alias of **o**) and **customers** (with an alias of **c**), on a shared dimension - **id**.

The question asks us to find users who have not placed an order. RIGHT JOIN keeps all rows from the **customers** table.

```
SELECT
    COUNT(*) AS n_customers_without_orders
```

```
FROM
    orders o
RIGHT OUTER JOIN
    customers c
ON
    o.cust_id = c.id
WHERE
    o.cust_id IS NULL
```

As we explained above, even though RIGHT JOIN keeps rows from the second table (in this case, **customers**), if customer is not found in the **orders** table, columns from this table (such as **cust_id**, will have a value of NULL).

Once data from two tables is combined, we use the WHERE statement to find instances when the **cust_id** column has a NULL value. This indicates that the customer has not placed an order.

Finally, we use the COUNT() aggregate function to get the number of all filtered rows.

Output

n_customers_without_orders
9

FULL OUTER JOIN

Here, we have a question asked during an interview at the software giant - Salesforce.

We have a list of companies and their yearly release of products.

We have to find how many products companies introduced in 2020 for the first time, and the difference between this value and the number of new launches in 2019.

Our output is going to be the name of the companies and net difference between two numbers.

Let’s look at the question description:

New Products

Salesforce Medium Interview Questions ID 10318 14 6

You are given a table of product launches by company by year. Write a query to count the net difference between the number of products companies launched in 2020 for the first time with the number of products companies launched in the previous year. Output the name of the companies and a net difference of net products released for 2020 compared to the previous year. If a company is new or had no products in 2019, then any product released in 2020 would be considered as new.

Table: car_launches

Link to the question: https://platform.stratascratch.com/coding/10318-new-products

This is the case when we use OUTER JOIN to combine two subqueries that filter the same table.

In each subquery, we filter the table to find new launches for 2020 and 2019. And we combine data from two tables using OUTER JOIN. The shared dimension is going to be the name of the company.

```
SELECT a.company_name,
       (count(DISTINCT a.brand_2020)-count(DISTINCT b.brand_2019))
net_products
FROM
  (SELECT company_name,
          product_name AS brand_2020
   FROM car_launches
   WHERE YEAR = 2020) a
FULL OUTER JOIN
  (SELECT company_name,
          product_name AS brand_2019
   FROM car_launches
   WHERE YEAR = 2019) b ON a.company_name = b.company_name
GROUP BY a.company_name
ORDER BY company_name
```

FULL OUTER JOIN returns all rows from both subqueries. This might lead to duplicate values.

We solve this problem in the **SELECT** statement. Like the question specifies, for our final output, we are going to **SELECT** the company name, and the difference between unique values (every release will count only once) from two subqueries.

The use of the **DISTINCT** keyword ensures that our aggregate function COUNT() will only return the number of unique values.The COUNT() aggregate function gets the number of unique rows. Finally, we perform a normal subtraction operation (-) to find the difference.

The output of this query would be:

Output

company_name	net_products
Chevrolet	2
Ford	-1
Honda	-3
Jeep	1
Toyota	-1

Cross JOINs

Finally, we have a question from consultancy company Deloitte.

In this question, we are tasked to find all possible permutations (combinations) of any two numbers from the list.

The output is going to be the first number, the second number, and the higher between the two.

Let's look at the question:

Maximum of Two Numbers

Deloitte Medium Active Interview ID 2101 4 0

Given a single column of numbers, consider all possible permutations of two numbers assuming that pairs of numbers (x,y) and (y,x) are two different permutations. Then, for each permutation, find the maximum of the two numbers.
Output three columns: the first number, the second number and the maximum of the two.

Link to the question: https://platform.stratascratch.com/coding/2101-maximum-of-two-numbers

We already mentioned that CROSS JOINs are the right solution for finding the maximum number of combinations between two tables. In this case, we have only one table, but we can CROSS JOIN it with itself to get all possible combinations of two numbers.

```
SELECT dn1.number AS number1,
       dn2.number AS number2,
       CASE
           WHEN dn1.number > dn2.number THEN dn1.number
           ELSE dn2.number
       END AS max_number
FROM deloitte_numbers AS dn1
CROSS JOIN deloitte_numbers AS dn2
```

Finally, we can write a CASE clause to output the number that is higher of the two.

Output

number1	number2	max_number
-2	-2	-2
-2	-1	-1
-2	0	0
-2	1	1
-2	2	2
-1	-2	-1
-1	-1	-1
-1	0	0
-1	1	1
-1	2	2
0	-2	0
0	-1	0
0	0	0

UNION

This time, we will go through the question from ride-hailing giant, Uber.

We have to find an aggregate sum of numbers two times. Once when the value of index is under 5, and once when it is over 5. Then we have to stack these sums on top of one another.

Here is how Uber interviewers formulate the question:

Sum of Numbers

Uber Medium General Practice ID 10008 3 0

Find the sum of numbers whose index is less than 5 and the sum of numbers whose index is greater than 5. Output each result on a separate row.

Link to the question: https://platform.stratascratch.com/coding/10008-sum-of-numbers

We will use the SUM() aggregate function to calculate the total of numbers that meet the condition, which can be specified using the WHERE statement.

```
SELECT SUM(number)
FROM transportation_numbers
WHERE INDEX < 5
UNION ALL
SELECT SUM(number)
FROM transportation_numbers
WHERE INDEX > 5
```

We can use **UNION** to append columns from one table to another. The final table will be the sum of numbers that satisfy the first condition, followed by the sum of numbers that satisfy the second condition.

Output

sum
16
16

In practice, it is very common to write UNION ALL to combine the output of two SELECT statements.

UNION ALL only works if the output of the SELECT statement has the same number of columns, in the same order, that contain the same data types.

Summary

In this article, we tried to explain how to use JOINs to work with data from two tables. Understanding differences between different types of SQL JOINs is necessary to be a productive SQL developer. A deep knowledge of SQL JOIN types can distinguish you from other candidates and help you land a job.