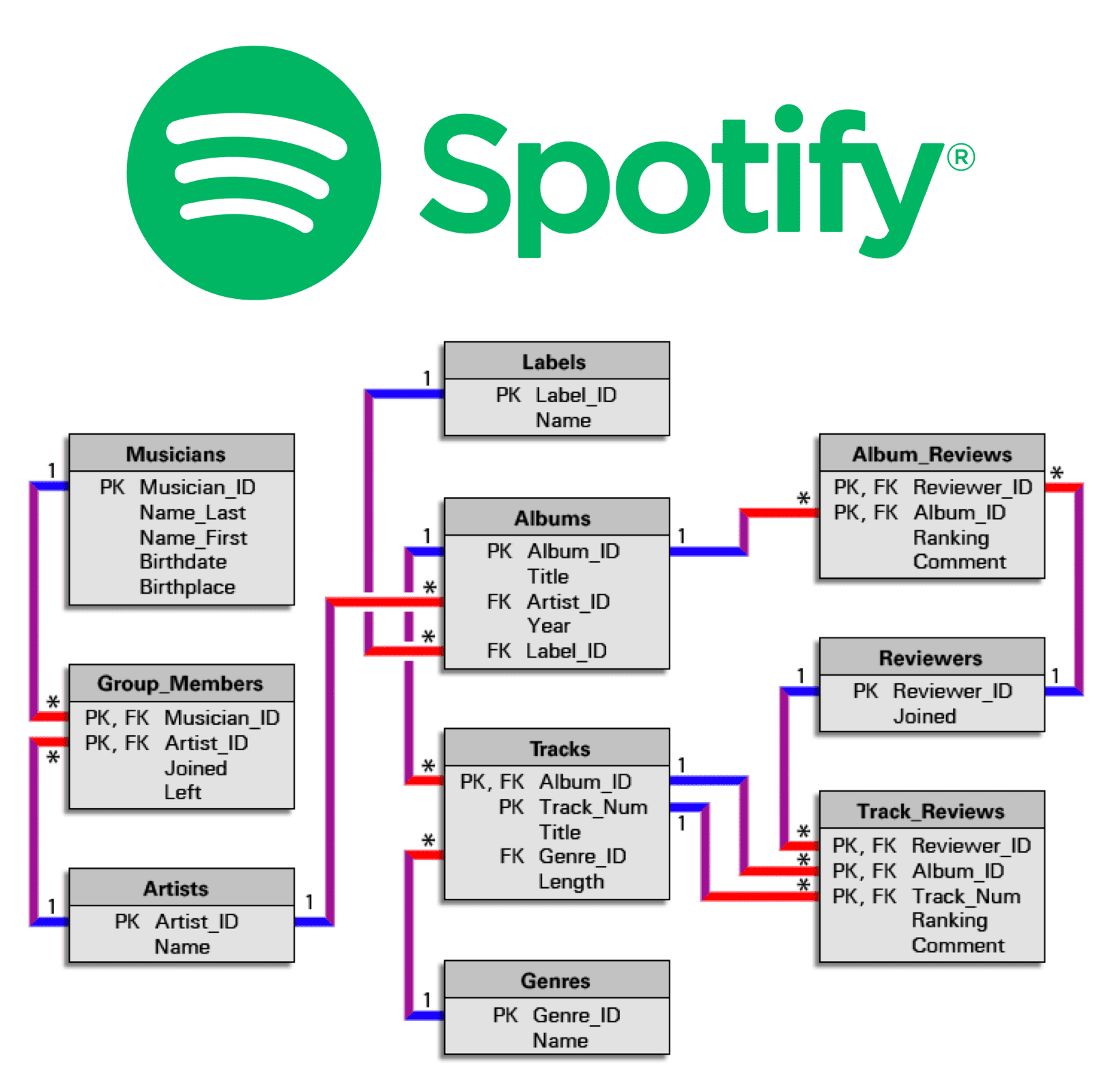
**SQL JOINS Tutorial With Practice Exercises**

So far in the SQL tutorial we've only been analyzing data from one table at a time. However, in the real world, companies have databases containing thousands of tables. To combine multiple tables, and analyze their data simultaneously, we can write a **JOIN** SQL query – the focus of this tutorial.

**Spotify SQL JOIN Case Study**

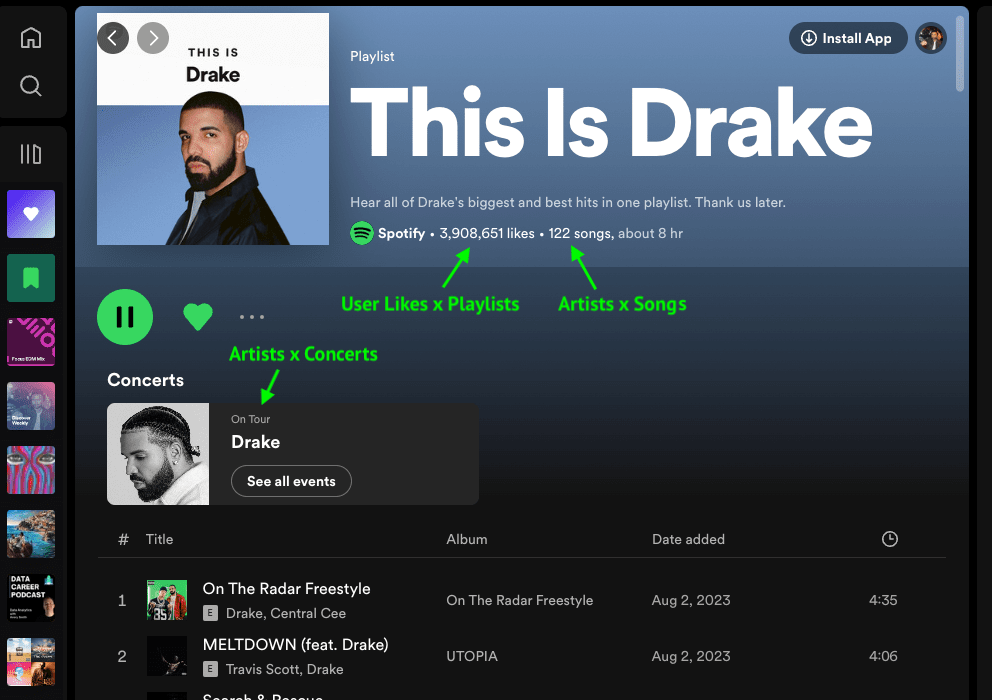
For example, let's pretend we work at Spotify where we have a table for albums, a table for artists, a table to represent users, and the list 🎵 *goes on and on and on*🎵 (sung to the tune of [Don't Stop Believin'](https://open.spotify.com/track/4bHsxqR3GMrXTxEPLuK5ue)).



As a Data Analyst or Data Scientist, you'll almost always be pulling data from multiple tables, which is why the SQL JOIN is crucial.

For example, at Spotify you might analyze Drake 🐐 by writing queries that combine two tables like:

* **JOIN** the **artists** and **concerts** table to analyze upcoming Drake concerts.
* **JOIN** the **user\_likes** and **playlists** table to count the # of likes on the *This Is Drake* playlist.
* **JOIN** the **artists** and **songs** table to rank Drake's top songs by play count.



Now that you understand the real-world use case of a **JOIN**, let's write a specific SQL join query using Spotify data.

**SQL JOIN Example**

Suppose you work at Spotify and want to join information in the **artists** table with information in the **songs** table. Here's the schema for the two tables:

**artists Example Input:**

| **artist\_id** | **artist\_name** | **label\_owner** |
| --- | --- | --- |
| 101 | Ed Sheeran | Warner Music Group |
| 120 | Drake | Warner Music Group |
| 125 | Bad Bunny | Rimas Entertainment |

**songs Example Input:**

| **song\_id** | **artist\_id** | **name** |
| --- | --- | --- |
| 55511 | 101 | Perfect |
| 45202 | 101 | Shape of You |
| 22222 | 120 | One Dance |
| 19960 | 120 | Hotline Bling |

Notice how the two tables have the **artist\_id** column in common:

* In the **artists** table, you have information about each **artist\_id**, like their artist name and what music label they work with.
* in the **songs** table, you have information about each song, which includes which **artist\_id** made that particular song.

Now, let's actually build our SQL JOIN query to combine the **artists** and **songs** table.

**How To Write A SQL JOIN**

There are three key parts to get right when writing a **JOIN** in SQL. First, you need to specify what columns to show. For our **artists** x **songs** example, we'll just use **SELECT \*** for that.

Secondly, you need to specify the names of the two tables we are joining. Since we are joining the **artists** with **songs** table, our query so far is as follows:

**SELECT \***

**FROM artists**

**JOIN songs;**

The third and final in a SQL JOIN query is the **ON** clause, which explains to the RDBMS (Relational Database Management System) how the two tables relate to each other.

We need to explicitly write in the SQL query that the **artists\_id** column in the **artists** table matches up against the **artist\_id** column in the **songs** table with this clause:

**ON artists.artist\_id = songs.artist\_id**

Putting these three parts together, we get the following SQL JOIN query:

**SELECT \***

**FROM artists**

**JOIN songs**

**ON artists.artist\_id = songs.artist\_id;**

The above query combines info from the **artists** and **songs** table into the result below:

| **artist\_id** | **artist\_name** | **label\_owner** | **song\_id** | **artist\_id** | **name** |
| --- | --- | --- | --- | --- | --- |
| 101 | Ed Sheeran | Warner Music Group | 55511 | 101 | Perfect |
| 101 | Ed Sheeran | Warner Music Group | 45202 | 101 | Shape of You |
| 120 | Drake | Warner Music Group | 22222 | 120 | One Dance |
| 120 | Drake | Warner Music Group | 19960 | 120 | Hotline Bling |
| 125 | Bad Bunny | Rimas Entertainment | 12636 | 125 | Mia |

Now, let's practice writing a **JOIN** on a different dataset.

**Easy SQL JOIN Practice Exercise**

Suppose you work as a Data Scientist at the stock-trading app Robinhood. Assume you're given access to a table called **trades** which contains information about trades placed on the platform, and a table called **users** which has information about a specific user.

Here's what the data looks like in both tables:

**trades Example Input:**

| **order\_id** | **user\_id** | **price** | **quantity** | **status** | **timestamp** |
| --- | --- | --- | --- | --- | --- |
| 100101 | 111 | 9.80 | 10 | Cancelled | 08/17/2022 12:00:00 |
| 100102 | 111 | 10.00 | 10 | Completed | 08/17/2022 12:00:00 |
| 100259 | 148 | 5.10 | 35 | Completed | 08/25/2022 12:00:00 |
| 100264 | 148 | 4.80 | 40 | Completed | 08/26/2022 12:00:00 |
| 100305 | 300 | 10.00 | 15 | Completed | 09/05/2022 12:00:00 |
| 100400 | 178 | 9.90 | 15 | Completed | 09/09/2022 12:00:00 |
| 100565 | 265 | 25.60 | 5 | Completed | 12/19/2022 12:00:00 |

**users Example Input:**

| **user\_id** | **city** | **email** | **signup\_date** |
| --- | --- | --- | --- |
| 111 | San Francisco | [rrok10@gmail.com](mailto:rrok10@gmail.com) | 08/03/2021 12:00:00 |
| 148 | Boston | [sailor9820@gmail.com](mailto:sailor9820@gmail.com) | 08/20/2021 12:00:00 |
| 178 | San Francisco | [harrypotterfan182@gmail.com](mailto:harrypotterfan182@gmail.com) | 01/05/2022 12:00:00 |
| 265 | Denver | [shadower\_@hotmail.com](mailto:shadower_@hotmail.com) | 02/26/2022 12:00:00 |
| 300 | San Francisco | [houstoncowboy1122@hotmail.com](mailto:houstoncowboy1122@hotmail.com) | 06/30/2022 12:00:00 |

**Write a SQL query to join the trades and users table.**

The output should look something like this:

| **order\_id** | **user\_id** | **quantity** | **status** | **date** | **price** | **user\_id** | **city** | **email** | **signup\_date** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 100102 | 111 | 10 | Completed | 08/17/2022 12:00:00 | 10.00 | 111 | San Francisco | [rrok10@gmail.com](mailto:rrok10@gmail.com) | 08/03/2021 12:00:00 |
| 100101 | 111 | 10 | Cancelled | 08/17/2022 12:00:00 | 9.80 | 111 | San Francisco | [rrok10@gmail.com](mailto:rrok10@gmail.com) | 08/03/2021 12:00:00 |
| 100900 | 148 | 50 | Completed | 07/14/2022 12:00:00 | 9.78 | 148 | Boston | [sailor9820@gmail.com](mailto:sailor9820@gmail.com) | 08/20/2021 12:00:00 |
| 100259 | 148 | 35 | Completed | 08/25/2022 12:00:00 | 5.10 | 148 | Boston | [sailor9820@gmail.com](mailto:sailor9820@gmail.com) | 08/20/2021 12:00:00 |
| 100264 | 148 | 40 | Completed | 08/26/2022 12:00:00 | 4.80 | 148 | Boston | [sailor9820@gmail.com](mailto:sailor9820@gmail.com) | 08/20/2021 12:00:00 |
| 100777 | 178 | 60 | Completed | 07/25/2022 17:47:00 | 3.56 | 178 | San Francisco | [harrypotterfan182@gmail.com](mailto:harrypotterfan182@gmail.com) | 01/05/2022 12:00:00 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |

**[Practice Problem](https://datalemur.com/questions/sql-join-practice-exercise-robinhood)**

[Join the trades table to the users table, and display the output of the results.](https://datalemur.com/questions/sql-join-practice-exercise-robinhood)

[Try it out](https://datalemur.com/questions/sql-join-practice-exercise-robinhood)

Be sure to complete this [SQL JOIN practice exercise](https://datalemur.com/questions/sql-join-practice-exercise-robinhood), as it's meant to warm you up before you solve a real SQL interview question using the same data in the next section.

**Harder SQL Join Interview Question**

A Data Scientist interviewing at Robinhood was asked this [SQL interview question](https://datalemur.com/questions/completed-trades) where you need to write a SQL JOIN query to find the top three cities that have the highest number of completed trade orders.

Your output should look like this:

| **city** | **total\_orders** |
| --- | --- |
| San Francisco | 3 |
| Boston | 2 |
| Denver | 1 |

Besides needing a **JOIN**, you'll also need **COUNT** and **GROUP BY** commands to solve this interview question.

**[Practice Problem](https://datalemur.com/questions/completed-trades)**

[Find the top 3 cities with the most completed trades.](https://datalemur.com/questions/completed-trades)

[Try it out](https://datalemur.com/questions/completed-trades)

**4 Types of SQL Joins**

So far in this tutorial, we've simply use the **JOIN** keyword, which is short for an **INNER JOIN**. There are actually 4 different types of JOINS in SQL which we'll now cover in detail.

| **Type of Joins** | **Description** |
| --- | --- |
| INNER JOIN | Returns only the **rows with matching values** from both tables. |
| LEFT JOIN | Returns **all the rows from the left table** and the **matching rows from the right table**. |
| RIGHT JOIN | Returns **all the rows from the right table** and the **matching rows from the left table**. |
| FULL OUTER JOIN | Returns all rows when there is a **match in either the left or the right table**. If there is no match, **NULL** values are returned for columns from the table without a match. |

We'll be using the [Goodreads Books dataset](https://datalemur.com/questions/sql-tutorial-joins) to demonstrate these different types of JOINs.

**SQL INNER JOIN**

An **INNER JOIN** returns only the **rows with matching values** from both tables.

Picture this: you want to find out how many books priced at $20 and above have been ordered, and who the buyers are.

**SELECT**

**orders.order\_id,**

**orders.customer\_id,**

**goodreads.book\_title,**

**orders.quantity**

**FROM goodreads**

**INNER JOIN orders**

**ON goodreads.book\_id = orders.book\_id -- Columns with same data type (integer)**

**WHERE goodreads.price >= 20;**

The **INNER JOIN** returns only the rows where there is a matching book ID in both the **goodreads** and **orders** tables, focusing on books priced $20 and above.

A screenshot of a computer

AI-generated content may be incorrect.

**SQL LEFT JOIN**

Next, let's explore how the **LEFT JOIN** works using our **orders** and **deliveries** tables.

Suppose we want to retrieve all the orders along with their corresponding deliveries information. Here's the query:

**SELECT**

**orders.order\_id,**

**deliveries.delivery\_id,**

**deliveries.delivery\_date,**

**deliveries.delivery\_status**

**FROM orders**

**LEFT JOIN deliveries**

**ON orders.order\_id = deliveries.order\_id;**

With **LEFT JOIN**, **all rows from the left table** (**orders**) are fetched, along with **matching rows from the right table** (**deliveries**). If there is no matching data in the right table, the result will still include the left table's data with **NULL** values in the columns from the right table.

In our example, we're displaying a subset of 10 rows out of 20:A screenshot of a logistic

AI-generated content may be incorrect.

But what if we flip the tables? By swapping the positions of the tables in the **LEFT JOIN**, the same result can be achieved:

**SELECT**

**orders.order\_id,**

**deliveries.delivery\_id,**

**deliveries.delivery\_date,**

**deliveries.delivery\_status**

**FROM deliveries**

**LEFT JOIN orders**

**ON deliveries.order\_id = orders.order\_id;**

Notice how order IDs 2005, 2008, and 2010 are missing from the results? Since these order IDs are absent in the **deliveries** table, there are no corresponding matches in the **orders** table. This results in the **NULL** values you see.A screenshot of a logistic

AI-generated content may be incorrect.

**SQL RIGHT JOIN**

A **RIGHT JOIN** is the opposite of a **LEFT JOIN**. It **returns all rows from the right table** and pairs them with **matching rows from the left table**. If there is no match, the columns from the left table have **NULL** values.

Run the following query:

**SELECT**

**orders.order\_id,**

**deliveries.delivery\_id,**

**deliveries.delivery\_date,**

**deliveries.delivery\_status**

**FROM deliveries**

**RIGHT JOIN orders**

**ON deliveries.order\_id = orders.order\_id;**

You can also swap the positions of the tables, and you'll get the exact same results as a **LEFT JOIN**.

**💡A little tip:** In the real world, **RIGHT JOIN** is rarely used. Most people naturally think from left to right, making **LEFT JOIN** easier to understand and implement. So, if you find yourself in a situation where you might consider a **RIGHT JOIN**, simply switch the positions of the tables and use a **LEFT JOIN** instead!

**SQL FULL OUTER JOIN**

Now, let's explore the **FULL OUTER JOIN** which allows you to bring unmatched rows into the results.

Ever wondered what happens when both tables have something to offer, but they're not a perfect match?

A **FULL OUTER JOIN** returns all rows when there is a **match in either the left or the right table**. If there is no match, **NULL** values are returned for columns from the table without a match.

**SELECT**

**orders.order\_id,**

**deliveries.delivery\_id,**

**deliveries.delivery\_date,**

**deliveries.delivery\_status**

**FROM orders**

**FULL OUTER JOIN deliveries**

**ON orders.order\_id = deliveries.order\_id;**

Running this query is like uniting two tables: **orders** and **deliveries**. Each row represents an attempt at a match. If there's no match, you'll see **NULL** values in the columns from the table without a match.

The first 20 rows are expected (we're just showing you the first 5 rows):A screenshot of a logistic

AI-generated content may be incorrect.

Scroll down, and you'll encounter rows with **NULL** order IDs (marked in the red box). The **FULL OUTER JOIN** ensures that even if there isn't an **order\_id** to match these rows, they're still include in the results:A screenshot of a computer

AI-generated content may be incorrect.

**💡 A little tip:** Be careful with **FULL OUTER JOIN** as it can introduce duplicate rows to your results without you realizing!

**Conditional SQL JOINS**

Conditional joins involve using single or multiple conditions using logical operators like **AND** and **OR**, or incorporating complex logical expressions such as ranges, pattern matching, or subqueries.

These conditions allow you to filter and retrieve data based on specific criteria during the join operation using the **ON** clause to specify conditions.

**Example 1: Joining goodreads and orders table with a condition on quantity**

Retrieve the list of book titles and their quantities ordered where the quantity ordered is more than 2.

**SELECT**

**g.book\_title,**

**o.quantity**

**FROM goodreads AS g**

**INNER JOIN orders AS o**

**ON g.book\_id = o.book\_id**

**AND o.quantity > 2;**

What happens is:

* This query applies the **o.quantity > 2** condition as part of the join condition itself. The join only includes rows where both **g.book\_id = o.book\_id** and **o.quantity > 2** are **true**.
* Since the filter condition is part of the join, no additional filtering is required after the join is performed.

**Example 2: Joining orders and deliveries tables with a condition on delivery\_status**

Retrieve order IDs and their corresponding delivery status where the delivery status is either 'Delivered' or 'Shipped'.

**SELECT**

**o.order\_id,**

**d.delivery\_status**

**FROM orders AS o**

**INNER JOIN deliveries AS d**

**ON o.order\_id = d.order\_id**

**AND d.delivery\_status IN ('Delivered', 'Shipped');**

**Example 3: Joining goodreads and orders with multiple conditions**

Retrieve book titles, their authors, and the order dates for books released after 2015 and ordered in quantities greater than 1.

**SELECT**

**g.book\_title,**

**g.author,**

**o.order\_date**

**FROM goodreads AS g**

**INNER JOIN orders AS o**

**ON g.book\_id = o.book\_id**

**AND g.year\_released > 2015**

**AND o.quantity > 1;**

**Example 4: Joining all three tables with a condition on book\_rating and delivery\_status**

Retrieve the book titles, their average ratings, order dates, and delivery statuses for books with a rating higher than 4.0 that have been delivered.

**SELECT**

**g.book\_title,**

**g.book\_rating,**

**o.order\_date,**

**d.delivery\_status**

**FROM goodreads g**

**JOIN orders o**

**ON g.book\_id = o.book\_id**

**AND g.book\_rating > 4.0**

**INNER JOIN deliveries d**

**ON o.order\_id = d.order\_id**

**AND d.delivery\_status = 'Delivered';**

These examples illustrate how to use conditional joins to filter and retrieve specific data based on various conditions across multiple tables.

**Handling Nulls + JOIN SQL Interview Question**

Let's tackle a real [Facebook/Meta SQL Interview](https://datalemur.com/questions/sql-page-with-no-likes) which requires a JOIN along with some [null handling](https://datalemur.com/sql-tutorial/sql-null).

**[Practice Problem](https://datalemur.com/questions/sql-page-with-no-likes)**

[Given tables containing Facebook pages and their respective likes, write a query to return the IDs of the Facebook pages that have zero likes.](https://datalemur.com/questions/sql-page-with-no-likes)

[Try it out](https://datalemur.com/questions/sql-page-with-no-likes)

**SQL JOIN with CASE WHEN**

Remember the last lesson on how we used [CASE WHEN](https://datalemur.com/sql-tutorial/sql-case-statement) to segment Marvel Actors based on how active they were on social media:



Let's combine the **CASE WHEN** statement, along with a **JOIN** to solve this real [Meta SQL interview question](https://datalemur.com/questions/updated-status) about segmenting Facebook advertisers based on their payment status: