**Intermediate SQL**

* 4 hours
* 14 Videos
* 50 Exercises
* 73,367 Participants
* 3,950 XP

**Course Description**



SQL is the most popular language for turning raw data stored in a database into actionable insights. Using a database of films made around the world, this course covers:  
  
✓ How to filter and compare data  
✓ How to use aggregate functions to summarize data  
✓ How to sort and group your data  
✓ How to present your data cleanly using tools such as rounding and aliasing  
  
Accompanied at every step with hands-on practice queries, this course teaches you everything you need to know to analyze data using your own SQL code today!

Read More

1. 1

**Selecting Data**

0%

In this first chapter, you’ll learn how to query a films database and select the data needed to answer questions about the movies and actors. You'll also understand how SQL code is executed and formatted.

**Querying a database**

50 xp

**Learning to COUNT()**

50 xp

**Practice with COUNT()**

100 xp

**SELECT DISTINCT**

100 xp

**Query execution**

50 xp

**Order of execution**

100 xp

**Debugging errors**

100 xp

**SQL style**

50 xp

**SQL best practices**

100 xp

**Formatting**

100 xp

**Non-standard fields**

50 xp

[Hide Chapter Details](https://www.datacamp.com/courses/intermediate-sql?embedded=true)

  2

**Filtering Records**

0%

Learn about how you can filter numerical and textual data with SQL. Filtering is an important use for this language. You’ll learn how to use new keywords and operators to help you narrow down your query to get results that meet your desired criteria and gain a better understanding of NULL values and how to handle them.

**Filtering numbers**

50 xp

**Filtering results**

50 xp

**Using WHERE with numbers**

100 xp

**Using WHERE with text**

100 xp

**Multiple criteria**

50 xp

**Using AND**

100 xp

**Using OR**

100 xp

**Using BETWEEN**

100 xp

**Filtering text**

50 xp

**LIKE and NOT LIKE**

100 xp

**WHERE IN**

100 xp

**Combining filtering and selecting**

100 xp

**NULL values**

50 xp

**What does NULL mean?**

50 xp

**Practice with NULLs**

100 xp

[Hide Chapter Details](https://www.datacamp.com/courses/intermediate-sql?embedded=true)

  3

**Aggregate Functions**

0%

SQL allows you to zoom in and out to better understand an entire dataset, its subsets, and its individual records. You'll learn to summarize data using aggregate functions and perform basic arithmetic calculations inside queries to gain insights into what makes a successful film.

**Summarizing data**

50 xp

**Aggregate functions and data types**

100 xp

**Practice with aggregate functions**

100 xp

**Summarizing subsets**

50 xp

**Combining aggregate functions with WHERE**

100 xp

**Using ROUND()**

100 xp

**ROUND() with a negative parameter**

100 xp

**Aliasing and arithmetic**

50 xp

**Using arithmetic**

50 xp

**Aliasing with functions**

100 xp

**Rounding results**

100 xp

[Hide Chapter Details](https://www.datacamp.com/courses/intermediate-sql?embedded=true)

  4

**Sorting and Grouping**

0%

This final chapter teaches you how to sort and group data. These skills will take your analyses to a new level by helping you uncover critical business insights and identify trends and performance. You'll get hands-on experience to determine which films performed the best and how movie durations and budgets changed over time.

**Sorting results**

50 xp

**Sorting text**

50 xp

**Sorting single fields**

100 xp

**Sorting multiple fields**

100 xp

**Grouping data**

50 xp

**GROUP BY single fields**

100 xp

**GROUP BY multiple fields**

100 xp

**Answering business questions**

50 xp

**Filtering grouped data**

50 xp

**Filter with HAVING**

100 xp

**HAVING and sorting**

100 xp

**All together now**

100 xp

**Daily XP750**

# Querying a database

**50 XP**

## 1. Querying a database

Hello, my name is Jasmin Ludolf, and I'll be your instructor for this course on using SQL to turn raw data into actionable insights. We'll build on our foundational knowledge of SQL, learn how to reveal insights, and how to present our results clearly.

## 2. Course roadmap

While SQL can be used to create and modify databases, the focus of this course will be querying databases. Recall that a query is a request for data from a database. In this course, we'll look at how to execute a query for a database using keywords that will enable us to count and view all or a specified amount of records. We'll go over common SQL errors, style guidelines, and the order in which our code will execute. We'll cover how to filter data using various techniques, how to use aggregate functions, and finally, how to sort and group the results. We'll be using PostgreSQL throughout.

## 3. Our films database

We will work with a films database containing four tables: films, reviews, people, and roles. Our database schema, pictured here, shows the table names, field names, and data types.

## 4. COUNT()

Here we go with our first new keyword. Let's say we wanted to count something from our people table. The COUNT function lets us do this by returning the number of records with a value in a field. For example, to count the number of birth dates present in the people table, we will use SELECT COUNT birthdate FROM people. The result is 6152 birthdates. We've used the alias "count birthdates" for the field name in this example to make the results more readable.

## 5. COUNT() multiple fields

If we want to count more than one field, we need to use COUNT multiple times. Here we are counting both the number of names and birth dates present in the people table.

## 6. Using \* with COUNT()

Using COUNT with a field name tells us how many values are in a field. However, if we want to count the number of records in a table, we can call COUNT with an asterisk. For example, this code gives the total number of records in the people table. The asterisk represents all fields. Passing the asterisk to COUNT is a shortcut for counting the total number of records.

## 7. DISTINCT

Often, our results will include duplicates. We can use the DISTINCT keyword to select all the unique values from a field. This might be useful if, for example, we're interested in knowing which languages are represented in the films table. Adding DISTINCT to our query will remove all duplicates, as we can see here.

## 8. COUNT() with DISTINCT

Combining COUNT with DISTINCT is also common to count the number of unique values in a field. This query counts the number of distinct birth dates in the people table. Let's take a moment to consider why this number is different from the birthdate count of 6152 we got before. Some people in our table likely share the same birthday; COUNT would include all the duplicates while DISTINCT counts all of the unique dates, no matter how many times they come up.

## 9. Let's practice!

Now, let's practice counting!

**Daily XP800**

**Exercise**

**Exercise**

**Learning to COUNT()**

You saw how to use COUNT() in the video. Do you remember what it returns?

Here is a query counting film\_id. Select the answer below that correctly describes what the query will return.

SELECT COUNT(film\_id) AS count\_film\_id

FROM reviews;

Run the query in the console to test your theory!

**Instructions**

**50 XP**

**Possible Answers**

* 

The number of unique films in the reviews table.

* 

**The number of records containing a film\_id.**

* 

The total number of records in the reviews table.

* 

The sum of the film\_id field.

Correct! COUNT(field\_name) returns the number of records containing a value in a field. In this example, that field is film\_id.

**Daily XP850**

**Exercise**

**Exercise**

**Practice with COUNT()**

As you've seen, COUNT(\*) tells you how many records are in a table. However, if you want to count the number of *non-missing* values in a particular field, you can call COUNT() on just that field.

Let's get some practice with COUNT()! You can look at the data in the tables throughout these exercises by clicking on the table name in the console.

**Instructions 1/3**

**35 XP**

* [1](javascript:void(0))
  + Count the number of records in the people table, aliasing the result as count\_records.

 [2](javascript:void(0))

* Count the number of records with a birthdate in the people table, aliasing the result as count\_birthdate.

 [3](javascript:void(0))

* Count the languages and countries in the films table; alias as count\_languages and count\_countries.
* -- Count the number of records in the people table
* \_\_\_
* -- Count the number of records in the people table
* SELECT COUNT(\*) AS count\_records
* FROM people;

| **count\_records** |
| --- |
| 8397 |

-- Count the number of birthdates in the people table

SELECT COUNT(birthdate) AS count\_birthdate

FROM people;

| **count\_birthdate** |
| --- |
| 6152 |

-- Count the languages and countries represented in the films table

SELECT COUNT(language) AS count\_languages, COUNT(country) AS count\_countries

FROM films;

| **count\_languages** | **count\_countries** |
| --- | --- |
| 4957 | 4966 |

Tres Bien! Looking at the differences between the count of unique values, total values, and all records can provide useful insights into your data.

**Daily XP950**

**Exercise**

**Exercise**

**SELECT DISTINCT**

Often query results will include many duplicate values. You can use the DISTINCT keyword to select the unique values from a field.

This might be useful if, for example, you're interested in knowing which languages are represented in the films table. See if you can find out what countries are represented in this table with the following exercises.

**Instructions 1/2**

**50 XP**

* [1](javascript:void(0))
  + Return the unique countries represented in the films table using DISTINCT.

 [2](javascript:void(0))

* Return the number of unique countries represented in the films table, aliased as count\_distinct\_countries.
* -- Return the unique countries from the films table
* SELECT DISTINCT country
* FROM films;

| **country** |
| --- |
| null |
| Soviet Union |
| Indonesia |
| Italy |
| Cameroon |
| Czech Republic |
| Sweden |
| USA |
| Dominican Republic |
| Cambodia |
| Ireland |
| Germany |
| Canada |
| Finland |
| South Korea |
| Colombia |
| New Line |
| Argentina |
| Slovenia |
| Egypt |
| Greece |

-- Count the distinct countries from the films table

SELECT COUNT(DISTINCT country) AS count\_distinct\_countries

FROM films;

| **count\_distinct\_countries** |
| --- |
| 64 |

Congratulations! Using DISTINCT is a great tool to see the unique values of a dataset. This table has 64 unique countries.

**Daily XP1050**

# Query execution

**50 XP**

## 1. Query execution

Fantastic work on using COUNT and DISTINCT! Now that we've flexed our SQL muscle a bit, we'll take a small step back and better understand how SQL code works.

## 2. Order of execution

Unlike many programming languages, SQL code is not processed in the order it is written. Consider we want to grab a coat from a closet: first, we need to know which closet contains the coats. This is similar to the FROM statement, which is the first line to be processed. Before any data can be selected, the table from which the data will be selected needs to be indicated. Next, our SELECTion is made. Finally, the results are refined. Here we use the LIMIT keyword that limits the results to a specified number of records. In this case, we only want to return the first ten names from the people table. Knowing processing order is especially useful when debugging and aliasing fields and tables. Suppose we need to refer to an alias later on in our code. In that case, that alias will only make sense to a processor when its declaration in the SELECT statement is processed before the alias reference is made elsewhere in the query.

## 3. Debugging SQL

Before we begin working with more advanced queries, it's useful to know more about debugging SQL code and how to read the error messages. Some messages are extremely helpful, pinpointing and even suggesting a solution for the error, as this message does when we misspell the "name" field we'd like to select. Other common errors may involve incorrect capitalization or punctuation.

## 4. Comma errors

Other error messages are less helpful and require us to review our code more closely. Forgetting a comma is a very common error. Let's say we've drafted this code to find all titles, country of origin, and duration of films. The error message will alert us to the general location of the error using a caret below the line of code, which in this case points to the "country" field name. We must examine the code a little further, though, to discover the missing comma is between "country" and "duration".

## 5. Keyword errors

SQL displays a similar error message when a keyword is misspelled, but this time, the caret indicator below the offending line is spot on.

## 6. Final note on errors

There are a few more SQL errors out there, but the three mentioned in this lesson will be the most common ones we will encounter. Debugging is a major skill, and the best way to master this skill is to make mistakes and learn from them.

## 7. Let's practice!

Let's practice!

**Daily XP1100**

**Exercise**

**Order of execution**

SQL code is processed differently than other programming languages in that you need to let the processor know where to pull the data from before making selections.

It's essential to know your code's order of execution to understand what results you'll get from your query and how to fix any errors that may come up.

**Instructions**

**100XP**

* Drag the SQL keywords into the order that they will be executed in from first (top) to last (bottom).

Congratulations! This is the correct order of execution. It makes sense that SQL needs to SELECT data FROM a table before it can LIMIT the results

**FROM SELECT LIMIT in that order of execution**

**Daily XP1200**

**Exercise**

**Exercise**

**Debugging errors**

Debugging is an essential skill for all coders, and it comes from making many mistakes and learning from them.

In this exercise, you'll be given some buggy code that you'll need to fix.

**Instructions 1/3**

**35 XP**

* [1](javascript:void(0))
  + Debug and fix the SQL query provided.

 [2](javascript:void(0))

* Find the two errors in this code; the same error has been repeated twice.

 [3](javascript:void(0))

* Find the two bugs in this final qu
* -- Debug this code
* SELECT certfication
* FROM films
* LIMIT 5;
* -- Debug this code
* SELECT certification
* FROM films
* LIMIT 5;

| **certification** |
| --- |
| Not Rated |
| null |
| Not Rated |
| Not Rated |
| Not Rated |

-- Debug this code

SELECT film\_id imdb\_score num\_votes

FROM reviews;

-- Debug this code

SELECT film\_id, imdb\_score, num\_votes

FROM reviews;

| **film\_id** | **imdb\_score** | **num\_votes** |
| --- | --- | --- |
| 3934 | 7.1 | 203461 |
| 3405 | 6.4 | 149998 |
| 478 | 3.2 | 8465 |
| 74 | 7.6 | 7071 |
| 1254 | 8 | 241030 |
| 740 | 6.4 | 64742 |
| 4841 | 8.1 | 479047 |
| 2869 | 6.8 | 18442 |
| 3252 | 7.2 | 49855 |

-- Debug this code

SELECT COUNNT(birthdate) AS count\_birthdays

FROM peeple;

-- Debug this code

SELECT COUNT(birthdate) AS count\_birthdays

FROM people;

| **count\_birthdays** |
| --- |
| 6152 |

Excellent extermination of those bugs! This is an important skill that will come in very handy.

**Daily XP1300**

# SQL style

**50 XP**

## 1. SQL style

Terrific job! Now that we understand how SQL works, we'll review how it looks.

## 2. SQL formatting

SQL is a generous language when it comes to formatting. New lines, capitalization, and indentation are not required in SQL as they sometimes are in other programming languages. For example, the code on this slide will run just fine, returning the first three titles, release years, and countries from the films table. However, writing queries like this won't make us any friends in the SQL world because the lack of formatting makes the code difficult to read, especially as queries become more complex.

## 3. Best practices

Over time, SQL users have developed style standards that are generally accepted across industries. This code returns the same results as the code on the previous slide, but it is much easier to read due to the addition of capitalized keywords and new lines between them.

## 4. Style guides

While keyword capitalization and new lines are standard practice, many of the finer details of SQL style are not. For instance, some SQL users prefer to create a new line and indent each selected field when a query selects multiple fields, as the query on this slide does.

## 5. Style guides

Because of the different formatting styles, it's helpful to follow a SQL style guide, such as Holywell's, which outlines a standard of best practices for indentation, capitalization, and naming conventions for tables, fields, and aliases. Remember, though, that there is no single required formatting in SQL: the guiding principle is writing clear and readable code.

## 6. Semicolon

Have you noticed the sample code we've been looking at throughout this lesson has a semicolon at the end? Like capitalization and new lines, this semicolon is unnecessary in PostgreSQL; we could leave it out of the query and still expect the same results with no errors. However, including a semicolon at the end of the query is considered best practice for several reasons. First, some SQL flavors require it, so it's a good habit to have. Including a semicolon in a PostgreSQL query means that the query is more easily translated to another flavor if necessary. Additionally, like a period at the end of a sentence, a semicolon at the end of a query indicates its end, which is helpful in a file containing several queries.

## 7. Dealing with non-standard field names

One last note on SQL style: while we can ensure our code is formatted beautifully, we don't have control over other people's SQL style. When creating a table, a SQL mistake is including spaces in a field name. To query that table, we'll need to enclose the field name in double-quotes to indicate that, despite being two words, the name refers to just one field. For example, if a sloppy SQL coder had named a field release-space-year as two words, we'd need to update the query we've seen throughout this chapter to the one shown here.

## 8. Why do we format?

Adhering to SQL style guides allows for easier collaboration between peers. Having clean and readable code is highly valued in the community and a professional setting and will make things easier for anyone wanting to understand or debug our queries.

## 9. Let's practice!

Let's try it out.

**Daily XP1350**

**Exercise**

**SQL best practices**

SQL style guides outline standard best practices for writing code.

This exercise will present several SQL style tips. Your job will be to decide whether they are considered best practices.

We'll be following [Holywell's style guide.](https://www.sqlstyle.guide/)

**Instructions**

**100XP**

* Drag and drop the items into the correct zone.

Well done! You'll soon become everyone's favorite SQL programmer.

**ercise**

**Exercise**

**Formatting**

Readable code is highly valued in the coding community and professional settings. Without proper formatting, code and results can be difficult to interpret. You'll often be working with other people that need to understand your code or be able to explain your results, so having a solid formatting habit is essential.

In this exercise, you'll correct poorly written code to better adhere to SQL style standards.

**Instructions**

**100 XP**

* Adjust the sample code so that it is in line with standard practices.
* -- Rewrite this query
* select person\_id, role from roles limit 10
* -- Rewrite this query
* SELECT person\_id, role
* FROM roles
* LIMIT 10;

| **person\_id** | **role** |
| --- | --- |
| 1630 | director |
| 4843 | actor |
| 5050 | actor |
| 8175 | actor |
| 3000 | director |
| 4019 | actor |
| 5274 | actor |
| 7449 | actor |

Great work formatting the code! Clean code allows for clean communication.

**Daily XP1550**

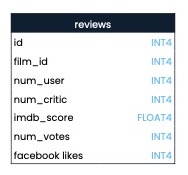
**Non-standard fields**

You may occasionally receive a dataset with poorly named fields. Ideally, you would fix these, but you can work around it with some added punctuation in this instance.

A sample query and schema have been provided; imagine you need to be able to run it with a non-standard field name. Select the multiple-choice answer that would correctly fill in the blank to return both a film's id and its number of Facebook likes for all reviews:

SELECT film\_id, **“facebook likes”**

FROM reviews;



**Answer the question**

**50XP**

**Possible Answers**

* 

facebook likes

press1

* 

"**facebook likes"**

press2

* 

facebook, likes

press3

Correct! Using double quotes around a non-standard name allows us to run the SQL query.

**Daily XP1600**

# Filtering numbers

**50 XP**

## 1. Filtering numbers

Welcome back! Now that we've mastered selecting and counting data from a database, we'll add to our repertoire by learning about filtering.

## 2. WHERE

To filter, we need to use a new clause called WHERE, which allows us to focus on only the data relevant to our business questions. Going back to our coat analogy, we may want to select a coat from the closet

## 3. WHERE

where the color is green. The WHERE clause can help us with that.

## 4. WHERE with comparison operators

We will focus on filtering numbers in this lesson. To do this, we will be using comparison operators such as greater than. Here is an example of a query where we filtered to see only films released after the year 1960 using the greater than operator.

## 5. Comparison operators

Let's explore some of the other operators. We would use the less-than operator to see films released before the year 1960.

## 6. Comparison operators

We would use the less than or equal to operator to see films released during or before the year 1960.

## 7. Comparison operators

If we want to see films released in a specific year, we can use equals.

## 8. Comparison operators

Here is a final example that isn't as intuitive as the others. If we wanted to filter films to see all releases EXCEPT those from the year 1960, we would combine the less than and greater than symbols as shown here. This is the SQL standard symbol that means "not equal to".

## 9. Comparison operators

Let's recap all the comparison operators we can use with WHERE to filter numbers. We have: greater than (that also means after), less than (that also means before), equal to, greater than or equal to, less than or equal to, and not equal to.

## 10. WHERE with strings

WHERE and the comparison operator, equals, can also be used with strings. In these cases, we will have to use single quotation marks around the strings we want to filter. For example, here, we want to filter titles where the country is Japan.

## 11. Order of execution

A final note on using WHERE. Similar to LIMIT, this clause comes after the FROM statement when writing a query. If we use both WHERE and LIMIT, the written order will be SELECT, FROM, WHERE, LIMIT; however, the order of execution will now be FROM, WHERE, SELECT, LIMIT. Thinking about the coats in our closet, we go to the closet we want to get the coat from, find where the green coats are, and select five of them.

## 12. Let's practice!

Ok, now it's time for practice!

**Daily XP1650**

**Filtering results**

The WHERE clause allows you to filter based on text and numeric values in a table using comparison operators.

What does the following query return?

SELECT title

FROM films

WHERE release\_year > 2000;

**Answer the question**

**50XP**

**Possible Answers**

* 

Films released before the year 2000

press1

* 

**Films released after the year 2000**

press2

* 

Films released after the year 2001

press3

* 

Films released in 2000

press4

Correct! Our query will return films that were released after the year 2000.

**Daily XP1700**

**Exercise**

**Exercise**

**Using WHERE with numbers**

Filtering with WHERE allows you to analyze your data better. You may have a dataset that includes a range of different movies, and you need to do a case study on the most notable films with the biggest budgets. In this case, you'll want to filter your data to a specific budget range.

Now it's your turn to use the WHERE clause to filter numeric values!

**Instructions 1/3**

**35 XP**

* [1](javascript:void(0))
  + Select the film\_id and imdb\_score from the reviews table and filter on scores higher than 7.0.

 [2](javascript:void(0))

* Select the film\_id and facebook\_likes of the first ten records with less than 1000 likes from the reviews table.

 [3](javascript:void(0))

* Count how many records have a num\_votes of at least 100,000; use the alias films\_
* -- Select film\_ids and imdb\_score with an imdb\_score over 7.0
* \_\_\_
* -- Select film\_ids and imdb\_score with an imdb\_score over 7.0
* SELECT film\_id, imdb\_score
* FROM reviews
* WHERE imdb\_score > 7.0;

| **film\_id** | **imdb\_score** |
| --- | --- |
| 3934 | 7.1 |
| 74 | 7.6 |
| 1254 | 8 |
| 4841 | 8.1 |
| 3252 | 7.2 |
| 1181 | 7.3 |
| 3929 | 7.1 |
| 3298 | 7.4 |

-- Select film\_ids and facebook\_likes for ten records with less than 1000 likes

\_\_\_

-- Select film\_ids and facebook\_likes for ten records with less than 1000 likes

SELECT film\_id, facebook\_likes

FROM reviews

WHERE facebook\_likes <1000

LIMIT 10;

| **film\_id** | **facebook\_likes** |
| --- | --- |
| 3405 | 0 |
| 478 | 491 |
| 74 | 930 |
| 740 | 0 |
| 2869 | 689 |
| 1181 | 0 |
| 2020 | 0 |
| 2312 | 912 |
| 1820 | 872 |

-- Count the records with at least 100,000 votes

\_\_\_

-- Count the records with at least 100,000 votes

SELECT COUNT(\*) AS films\_over\_100k\_votes

FROM reviews

WHERE num\_votes >= 100000;

| **films\_over\_100k\_votes** |
| --- |
| 1211 |

ell done! Applying a WHERE filter with SQL is much easier and faster than scrolling through a spreadsheet or using a highlighter!

**Using WHERE with text**

WHERE can also filter string values.

Imagine you are part of an organization that gives cinematography awards, and you have several international categories. Before you confirm an award for every language listed in your dataset, it may be worth seeing if there are enough films of a specific language to make it a fair competition. If there is only one movie or a significant skew, it may be worth considering a different way of giving international awards.

Let's try this out!

**Instructions**

**100 XP**

* Select and count the language field using the alias count\_spanish.
* Apply a filter to select only Spanish from the language field.
* -- Count the Spanish-language films
* \_\_\_
* -- Count the Spanish-language films
* SELECT COUNT(language) AS count\_spanish
* FROM films
* WHERE language = 'Spanish';

| **count\_spanish** |
| --- |
| 40 |

Bien hecho! Well done! There are 40 Spanish-language films in this table.

Ctrl+O

**Daily XP1900**

# Multiple criteria

**50 XP**

## 1. Multiple criteria

Great work on filtering numbers! Our SQL skills are growing fast. Next up, we will look at how to filter with multiple criteria.

## 2. Multiple criteria

There will often be the case that we have more than one criteria we'd like to meet. Looking again at our favorite coats, perhaps we want narrow down our choices to coats

## 3. Multiple criteria

that are yellow

## 4. Multiple criteria

and shorter in length.

## 5. Multiple criteria

We will be learning about three additional keywords that will allow us to enhance our filters when using WHERE by adding multiple criteria. These are OR, AND, and BETWEEN. In the context of our coats, we could look at coats where the color is yellow or the length is short, or we could filter for coats where both criteria are true. We can also look for coats that have between one and five buttons.

## 6. OR operator

The first keyword we will look at is the OR operator. OR is used when we want to filter multiple criteria and only need to satisfy at least one condition. Perhaps we want to select green or purple coat options as an example.

## 7. OR operator

In SQL, we combine OR with WHERE to achieve this type of filtering. Here is an example using the films database. The query on the left returns all films released in either 1994 or 2000. Note that we must specify the field for every OR condition, so the query on the right is invalid. That query hasn't specified what field or operator should be associated with the year 2000.

## 8. AND operator

If we want to satisfy all criteria in our filter, we need to use AND with WHERE. For example, this query gives us the titles of films released between 1994 and 2000. We need to specify the field name separately for every AND condition as with OR.

## 9. AND, OR

Let's kick it up a notch. We now want to filter films released in 1994 OR 1995, AND with a certification of either PG or R. Thankfully, we can combine AND and OR to answer this question. If a query has multiple filtering conditions, we will need to enclose the individual clauses in parentheses to ensure the correct execution order; otherwise, we may not get the expected results.

## 10. BETWEEN, AND

As we've learned, we can use this query to get titles of all films released in and between 1994 and 2000. Checking for ranges like this is very common, so in SQL the BETWEEN keyword provides a valuable shorthand for filtering values within a specified range. This second query is equivalent to the one on the left. It's important to remember that BETWEEN is inclusive, meaning the results contain the beginning and end values.

## 11. BETWEEN, AND, OR

Like the WHERE clause, the BETWEEN clause can be used with multiple AND and OR operators, so we can build up our queries and make them even more powerful! For example, we can get the titles of all films released between 1994 and 2000 from the United Kingdom.

## 12. Let's practice!

Have a go at using BETWEEN, AND, and OR on the films database.

**Daily XP1950**

**Exercise**

**Exercise**

**Using AND**

The following exercises combine AND and OR with the WHERE clause. Using these operators together strengthens your queries and analyses of data.

You will apply these new skills now on the films database.

**Instructions 1/3**

**35 XP**

* [1](javascript:void(0))
  + Select the title and release\_year for all German-language films released before 2000.

 [2](javascript:void(0))

* Update the query from the previous step to show German-language films released after 2000 rather than before.

 [3](javascript:void(0))

* Select all details for German-language films released after 2000 but before 2010 using **only** WHERE and AND.
* - Select the title and release\_year for all German-language films released before 2000
* \_\_\_
* -- Select the title and release\_year for all German-language films released before 2000
* SELECT title, release\_year
* FROM films
* WHERE language = 'German' AND release\_year < 2000;

| **title** | **release\_year** |
| --- | --- |
| Metropolis | 1927 |
| Pandora's Box | 1929 |
| The Torture Chamber of Dr. Sadism | 1967 |
| Das Boot | 1981 |
| Run Lola Run | 1998 |
| Aimee & Jaguar |  |

-- Update the query to see all German-language films released after 2000

SELECT title, release\_year

FROM films

WHERE release\_year > 2000

    AND language = 'German';

| **title** | **release\_year** |
| --- | --- |
| Good Bye Lenin! | 2003 |
| Downfall | 2004 |
| Summer Storm | 2004 |
| The Lives of Others | 2006 |
| The Baader Meinhof Complex | 2008 |
| The Wave | 2008 |
| Cargo | 2009 |
| Soul Kitchen |  |

-- Select all records for German-language films released after 2000 and before 2010

SELECT \*

FROM films

WHERE (release\_year > 2000 AND release\_year < 2010)

    AND language = 'German';

| **id** | **title** | **release\_year** | **country** | **duration** | **language** | **certification** | **gross** | **budget** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1952 | Good Bye Lenin! | 2003 | Germany | 121 | German | R | 4063859 | 4800000 |
| 2130 | Downfall | 2004 | Germany | 178 | German | R | 5501940 | 13500000 |
| 2224 | Summer Storm | 2004 | Germany | 98 | German | R | 95016 | 2700000 |
| 2709 | The Lives of Others | 2006 | Germany | 137 | German | R | 11284657 | 2000000 |
| 3100 | The Baader Meinhof Complex | 2008 | Germany | 184 | German | R | 476270 | 20000000 |
| 3143 | The Wave | 2008 | Germany | 107 | German | null | null | 5000000 |
| 3220 | Cargo | 2009 | Switzerland | 112 | German | null | null | 4500000 |
| 3346 | Soul Kitchen | 2009 | Germany | 99 | German | null | 274385 | 4000000 |

Great work! Combining conditions with AND will prove to be very useful when we want our query to return a specific subset of records

**Daily XP2050**

**Exercise**

**Exercise**

**Using OR**

This time you'll write a query to get the title and release\_year of films released in 1990 or 1999, which were in English or Spanish and took in more than $2,000,000 gross.

It looks like a lot, but you can build the query up one step at a time to get comfortable with the underlying concept in each step. Let's go!

**Instructions 1/3**

**35 XP**

* [1](javascript:void(0))
* [2](javascript:void(0))
* [3](javascript:void(0))
* Select the title and release\_year for films released in 1990 or 1999 using only WHERE and OR.
* -- Find the title and year of films from the 1990 or 1999
* \_\_\_
* -- Find the title and year of films from the 1990 or 1999
* SELECT title, release\_year
* FROM films
* WHERE release\_year = 1990 OR release\_year = 1999;

| **title** | **release\_year** |
| --- | --- |
| Arachnophobia | 1990 |
| Back to the Future Part III | 1990 |
| Child's Play 2 | 1990 |
| Dances with Wolves | 1990 |
| Days of Thunder | 1990 |
| Dick Tracy | 1990 |
| Die Hard 2 | 1990 |
| Edward Scissorhands | 1990 |
| Flatliners | 1990 |

SELECT title, release\_year

FROM films

WHERE (release\_year = 1990 OR release\_year = 1999)

-- Add a filter to see only English or Spanish-language films

    AND (language = 'English' OR language = 'Spanish');

| **title** | **release\_year** |
| --- | --- |
| Arachnophobia | 1990 |
| Back to the Future Part III | 1990 |
| Child's Play 2 | 1990 |
| Dances with Wolves | 1990 |
| Days of Thunder | 1990 |
| Dick Tracy | 1990 |
| Die Hard 2 | 1990 |
| Edward Scissorhands |  |

* Finally, restrict the query to only return films worth more than $2,000,000 gross.

SELECT title, release\_year

FROM films

WHERE (release\_year = 1990 OR release\_year = 1999)

    AND (language = 'English' OR language = 'Spanish')

-- Filter films with more than $2,000,000 gross

    \_\_\_;

SELECT title, release\_year

FROM films

WHERE (release\_year = 1990 OR release\_year = 1999)

    AND (language = 'English' OR language = 'Spanish')

-- Filter films with more than $2,000,000 gross

    AND gross > 2000000;

| **title** | **release\_year** |
| --- | --- |
| Arachnophobia | 1990 |
| Back to the Future Part III | 1990 |
| Child's Play 2 | 1990 |
| Dances with Wolves | 1990 |
| Days of Thunder | 1990 |
| Dick Tracy | 1990 |
| Die Hard 2 | 1990 |
| Edward Scissorhands | 1990 |
| Flatliners | 1990 |

**Daily XP2150**

**Exercise**

**Exercise**

**Using BETWEEN**

Let's use BETWEEN with AND on the films database to get the title and release\_year of all Spanish-language films released between 1990 and 2000 (inclusive) with budgets over $100 million.

We have broken the problem into smaller steps so that you can build the query as you go along!

**Instructions 1/4**

**25 XP**

* [1](javascript:void(0))
* [2](javascript:void(0))
* [3](javascript:void(0))
* [4](javascript:void(0))
* Select the title and release\_year of all films released between 1990 and 2000 (inclusive) using BETWEEN.
* -- Select the title and release\_year for films released between 1990 and 2000
* \_\_\_
* -- Select the title and release\_year for films released between 1990 and 2000
* SELECT title, release\_year
* FROM films
* WHERE release\_year
* BETWEEN 1990 AND 2000;

| **title** | **release\_year** |
| --- | --- |
| Arachnophobia | 1990 |
| Back to the Future Part III | 1990 |
| Child's Play 2 | 1990 |
| Dances with Wolves | 1990 |
| Days of Thunder | 1990 |
| Dick Tracy | 1990 |
| Die Hard 2 | 1990 |
| Edward Scissorhands | 1990 |
| Flatliners | 1990 |

Build on your previous query to select only films with a budget over $100 million

SELECT title, release\_year

FROM films

WHERE release\_year BETWEEN 1990 AND 2000

-- Narrow down your query to films with budgets > $100 million

    AND budget > 100000000;

| **title** | **release\_year** |
| --- | --- |
| Terminator 2: Judgment Day | 1991 |
| True Lies | 1994 |
| Waterworld | 1995 |
| Batman & Robin | 1997 |
| Dante's Peak | 1997 |
| Princess Mononoke | 1997 |
| Speed 2: Cruise Control | 1997 |
| Starship Troopers | 1997 |
| Titanic | 1997 |

Now, restrict the query to only return Spanish-language films.

SELECT title, release\_year

FROM films

WHERE release\_year BETWEEN 1990 AND 2000

    AND budget > 100000000

-- Restrict the query to only Spanish-language films

    \_\_\_;

SELECT title, release\_year

FROM films

WHERE release\_year BETWEEN 1990 AND 2000

    AND budget > 100000000

-- Restrict the query to only Spanish-language films

    AND language = 'Spanish';

| **title** | **release\_year** |
| --- | --- |
| Tango | 1998 |

Finally, amend the query to include all Spanish-language or French-language films with the same criteria.

SELECT title, release\_year

FROM films

WHERE release\_year BETWEEN 1990 AND 2000

    AND budget > 100000000

-- Amend the query to include Spanish or French-language films

    AND (language = 'Spanish' OR language = 'French');

| **title** | **release\_year** |
| --- | --- |
| Les couloirs du temps: Les visiteurs II | 1998 |
| Tango | 1998 |

Superb! Using WHERE with a combination of AND, OR, and BETWEEN is an efficient way to query a desired range of values.

**Daily XP100**

# Filtering text

**50 XP**

## 1. Filtering text

We're making excellent progress! We will now switch our focus away from filtering numbers to filtering textual data.

## 2. Filtering text

As we've briefly seen, we can use the WHERE clause to filter text data. However, so far, we've only been able to filter by specifying the exact text we're interested in.

## 3. Filtering text

We'll often want to search for a pattern rather than a specific text string in the real world. We'll be introducing three more SQL keywords into our vocabulary to help us achieve this: LIKE, NOT LIKE, and IN.

## 4. LIKE

In SQL, we can use the LIKE operator with a WHERE clause to search for a pattern in a field. We use a wildcard as a placeholder for some other values to accomplish this. There are two wildcards with LIKE, the percent, and the underscore. The percent wildcard will match zero, one, or many characters in the text. For example, the query on the left matches people like Adel, Adelaide, and Aden. The underscore wildcard will match a single character. For example, the query on the right matches only three-letter names like Eve. We'd also see names like Eva if it were in our dataset. Eva Mendes, however, would not be visible unless the search criteria looked like this.

## 5. NOT LIKE

We can also use the NOT LIKE operator to find records that don't match the specified pattern. In this query, we are finding records for people who do not have A-dot as part of their first name. It's important to note that this operation is case-sensitive, so we must be mindful of what we are querying.

## 6. Wildcard position

We've reviewed one example of where to position each wildcard, but we can actually put them anywhere and combine them! We can find values that start, end, or contain characters in any position, as well as find records of a certain length. For example, this code on the left will find all people whose name ends in r. The code on the right will find records where the third character is t.

## 7. WHERE, OR

What if we want to filter based on many conditions or a range of numbers? We could chain several ORs to the WHERE clause based on what we know, but that can get messy. We can see an example here where we select the film titles released in 1920, 1930, or 1940.

## 8. WHERE, IN

A helpful operator here is IN. The IN operator allows us to specify multiple values in a WHERE clause, making it easier and quicker to set numerous OR conditions. Neat, right? So, the example shown on the previous slide would simply become WHERE release\_year IN 1920, 1930, 1940, where the years are enclosed in parentheses.

## 9. WHERE, IN

Here is another example using a text field where we want to find the title WHERE the associated country is either Germany or France.

## 10. Let's practice!

Let's practice!

**Daily XP150**

**Exercise**

**Exercise**

**LIKE and NOT LIKE**

The LIKE and NOT LIKE operators can be used to find records that either match or do not match a specified pattern, respectively. They can be coupled with the wildcards % and \_. The % will match zero or many characters, and \_ will match a single character.

This is useful when you want to filter text, but not to an exact word.

Do the following exercises to gain some practice with these keywords.

**Instructions 1/3**

**35 XP**

* [1](javascript:void(0))
  + Select the names of all people whose names begin with 'B'.

 [2](javascript:void(0))

* Select the names of people whose names have 'r' as the second letter.

 [3](javascript:void(0))

* Select the names of people whose names don't start with 'A'.
* -- Select the names that start with B
* \_\_\_
* -- Select the names that start with B
* SELECT name
* FROM people
* WHERE name LIKE 'B%';

| **name** |
| --- |
| B.J. Novak |
| Babak Najafi |
| Babar Ahmed |
| Bahare Seddiqi |
| Bai Ling |
| Bailee Madison |
| Balinese Tari Legong Dancers |
| Bálint Péntek |
| Baltasar Kormákur |

SELECT name

FROM people

-- Select the names that have r as the second letter

WHERE name LIKE '\_r%';

| **name** |
| --- |
| Ara Celi |
| Aramis Knight |
| Arben Bajraktaraj |
| Arcelia Ramírez |
| Archie Kao |
| Archie Panjabi |
| Aretha Franklin |
| Ari Folman |
| Ari Gold |

SELECT name

FROM people

-- Select names that don't start with A

WHERE name NOT LIKE 'A%';

| **name** |
| --- |
| 50 Cent |
| Álex Angulo |
| Álex de la Iglesia |
| Ángela Molina |
| B.J. Novak |
| Babak Najafi |
| Babar Ahmed |
| Bahare Seddiqi |

I LIKE to see the progress we're making! Filtering your data to find specified patterns is vital to your skillset. Our results still had names that started with Á with an accent, showing that we need to be specific with our filtering criteria.

**Daily XP250**

**Exercise**

**Exercise**

**WHERE IN**

You now know you can query multiple conditions using the IN operator and a set of parentheses. It is a valuable piece of code that helps us keep our queries clean and concise.

Try using the IN operator yourself!

**Instructions 1/3**

**35 XP**

* [1](javascript:void(0))
  + Select the title and release\_year of all films released in 1990 or 2000 that were longer than two hours.

 [2](javascript:void(0))

* Select the title and language of all films in English, Spanish, or French using IN.

 [3](javascript:void(0))

* Select the title, certification and language of all films certified NC-17 or R that are in English, Italian, or Greek.
* -- Find the title and release\_year for all films over two hours in length released in 1990 and 2000
* SELECT title, release\_year
* FROM films
* WHERE release\_year IN (1990, 2000)
* AND duration > 120;

| **title** | **release\_year** |
| --- | --- |
| Dances with Wolves | 1990 |
| Die Hard 2 | 1990 |
| Ghost | 1990 |
| Goodfellas | 1990 |
| Mo' Better Blues | 1990 |
| Pretty Woman | 1990 |
| The Godfather: Part III | 1990 |
| The Hunt for Red October | 1990 |
| All the Pretty Horses |  |

-- Find the title and language of all films in English, Spanish, and French

SELECT title, language

FROM films

WHERE language IN ('English', 'Spanish', 'French');

| **title** | **language** |
| --- | --- |
| The Broadway Melody | English |
| Hell's Angels | English |
| A Farewell to Arms | English |
| 42nd Street | English |
| She Done Him Wrong | English |
| It Happened One Night | English |
| Top Hat | English |
| Modern Times | English |
| The Charge of the Light Brigad |  |

-- Find the title, certification, and language all films certified NC-17 or R that are in English, Italian, or Greek

SELECT title, certification, language

FROM films

WHERE certification IN ('NC-17', 'R')

    AND language IN ('English', 'Italian', 'Greek');

| **title** | **certification** | **language** |
| --- | --- | --- |
| Psycho | R | English |
| A Fistful of Dollars | R | Italian |
| Rosemary's Baby | R | English |
| The Wild Bunch | R | English |
| Catch-22 | R | English |
| Cotton Comes to Harlem | R | English |
| The Ballad of Cable Hogue | R | English |
| The Conformist |  |  |

Your SQL vocabulary is growing by the minute! Interestingly, A Fistful of Dollars starring Clint Eastwood is listed as Italian.

**Daily XP350**

**Exercise**

**Exercise**

**Combining filtering and selecting**

Time for a little challenge. So far, your SQL vocabulary from this course includes COUNT(), DISTINCT, LIMIT, WHERE, OR, AND, BETWEEN, LIKE, NOT LIKE, and IN. In this exercise, you will try to use some of these together. Writing more complex queries will be standard for you as you become a qualified SQL programmer.

As this query will be a little more complicated than what you've seen so far, we've included a bit of code to get you started. You will be using DISTINCT here too because, surprise, there are two movies named 'Hamlet' in this dataset!

Follow the instructions to find out what 90's films we have in our dataset that would be suitable for English-speaking teens.

**Instructions**

**100 XP**

* Count the unique titles from the films database and use the alias provided.
* Filter to include only movies with a release\_year from 1990 to 1999, inclusive.
* Add another filter narrowing your query down to English-language films.
* Add a final filter to select only films with 'G', 'PG', 'PG-13' certifications.
* -- Count the unique titles
* SELECT \_\_\_ AS nineties\_english\_films\_for\_teens
* FROM films
* -- Filter to release\_years to between 1990 and 1999
* WHERE \_\_\_
* -- Filter to English-language films
* \_\_\_
* -- Narrow it down to G, PG, and PG-13 certifications
* \_\_\_;
* -- Count the unique titles
* SELECT COUNT(DISTINCT title) AS nineties\_english\_films\_for\_teens
* FROM films
* -- Filter to release\_years to between 1990 and 1999
* WHERE release\_year BETWEEN 1990 AND 1999
* -- Filter to English-language films
* AND language = 'English'
* -- Narrow it down to G, PG, and PG-13 certifications
* AND certification IN ('G', 'PG', 'PG-13');

| **nineties\_english\_films\_for\_teens** |
| --- |
| 310 |

You've got a natural flair for filtering! Nice work, this filter tells us we have 310 films that the 90's obsessed teenagers can enjoy.

**Daily XP450**

# NULL values

**50 XP**

## 1. NULL values

Our next lesson will review how to filter data that includes NULL values.

## 2. Missing values

When we were learning how to use the COUNT keyword, we learned that we could include or exclude non-missing values depending on whether or not we use the asterisk in our query. But what is a missing or non-missing value? In SQL, NULL represents a missing or unknown value. Why is this useful? In the real world, our databases will likely have empty fields either because of human error or because the information is not available or is unknown. Knowing how to handle these fields is essential as they can affect any analyses we do.

## 3. null

For example, we used COUNT all with an asterisk on the left. Suppose our goal is to analyze posthumous success using data from the people table. We might make the wrong assumption that because we have a field name called deathdate, this information is available for everyone. Half of them are, in fact, NULL, as we can see on the right, so we would make an inaccurate judgment on what the data means.

## 4. IS NULL

One quick way to see how much of our data is missing is by using IS NULL with the WHERE clause. Here is an example where we have checked to see which names do not have a recorded birthdate in our table.

## 5. IS NOT NULL

On the left is an example of counting the missing birthdates in the people table. The count is 2245. Sometimes, we'll want to filter out missing values, so we only get results that are not NULL. To do this, we can use the IS NOT NULL operator. For example, this query on the right gives the count of all people whose birth dates are not missing in the people table, giving us a new count of 6152.

## 6. COUNT() vs IS NOT NULL

There may be a question about the difference between using COUNT with a field name and using the same COUNT with the added WHERE clause with IS NOT NULL. The answer is there is no difference, as both will be counting non-missing values.

## 7. NULL put simply

Before we wrap up this lesson, let's review what we've learned. NULL values are missing values, and they are very common in the real world. It is good practice to know how many NULL values are in our data by using the IS NULL or IS NOT NULL operator for filtering. These keywords will help to identify, select, or exclude missing values. Don't worry; this will soon become second nature because it is that common!

## 8. Let's practice!

That's a wrap! It's time for more practice.

**Daily XP500**

**What does NULL mean?**

I hope you were paying attention in the video. Pop quiz: What does NULL represent?

**Answer the question**

**50XP**

**Possible Answers**

* 

A corrupt entry

press1

* 

**A missing value**

press2

* 

An empty string

press3

* 

An invalid value

press4

+50 XP

Correct! NULL is used to represent unknown values.

**Daily XP550**

**Exercise**

**Exercise**

**Practice with NULLs**

Well done. Now that you know what NULL means and what it's used for, it's time for some more practice!

Let's explore the films table again to better understand what data you have.

**Instructions 1/2**

**50 XP**

* [1](javascript:void(0))
  + Select the title of every film that doesn't have a budget associated with it and use the alias no\_budget\_info.

 [2](javascript:void(0))

* Count the number of films with a language associated with them and use the alias count\_language\_known.
* -- List all film titles with missing budgets
* SELECT title AS no\_budget\_info
* FROM films
* WHERE budget IS NULL;

| **no\_budget\_info** |
| --- |
| Pandora's Box |
| The Prisoner of Zenda |
| The Blue Bird |
| Bambi |
| State Fair |
| Open Secret |
| Deadline - U.S.A. |
| Ordet |
| The Party's Over |

-- Count the number of films we have language data for

SELECT COUNT(language) AS count\_language\_known

FROM films

WHERE language IS NOT NULL;

| **count\_language\_known** |
| --- |
| 4957 |

Alright! That's 4957 films with language data. We've mastered selecting and filtering data which means you're halfway through the course!