# CodeAcademy - SQL

* Data extraction with SQL
* Programming basics with Python
* Data analysis using pandas, a Python library
* Data visualization using Matplotlib, a Python library
* Machine Learning using scikit-learn, a Python library

# Exploring Data with SQL

A **database** is a set of data stored in a computer. This data is usually structured into *tables*. Tables can grow large and have a multitude of columns and records.

Spreadsheets, like Microsoft Excel and Google Sheets, allow you to view and manipulate data directly: with selecting, filtering, sorting, etc. By applying a number of these operations you can obtain the subset of data you are seeking.

SQL (pronounced “S-Q-L” or “sequel”) allows you to write **queries** which define the subset of data you are seeking. Unlike Excel and Sheets, your computer and SQL will handle how to get the data; you can focus on what data you would like. You can save these queries, refine them, share them, and run them on different databases.

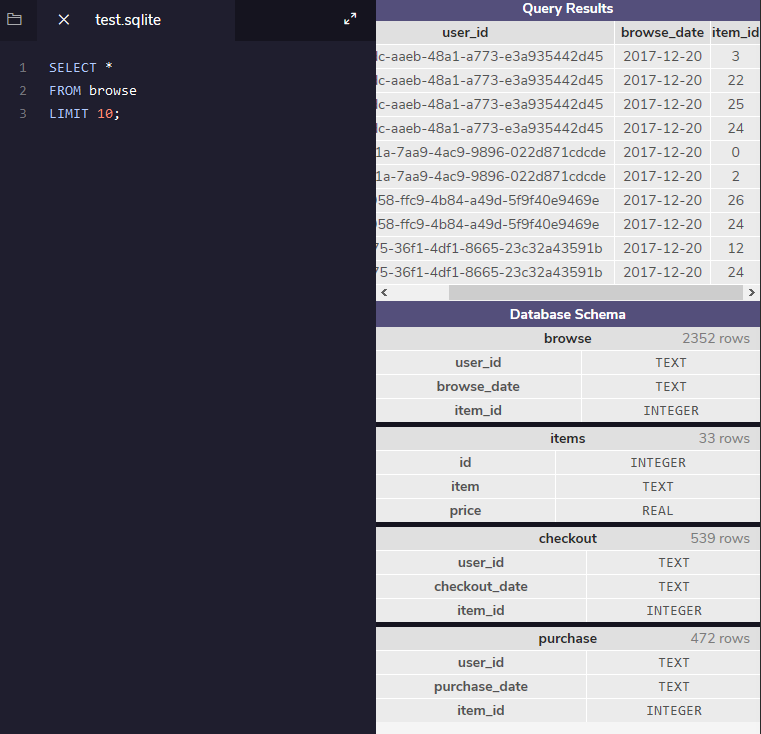
It is a great way to access data and a great entry point to programming because its syntax (the specific vocabulary that gives instructions to the computer) is very human-readable. Without knowing any SQL, you might still be able to guess what each command will do.

This code will select all (\*) columns from browse table for the first 10 records.

SELECT \*

FROM browse

LIMIT 10;



What is a Database?

A **database** is a set of data stored in a computer. This data is usually structured in a way that makes the data easily accessible.

What is a Relational Database?

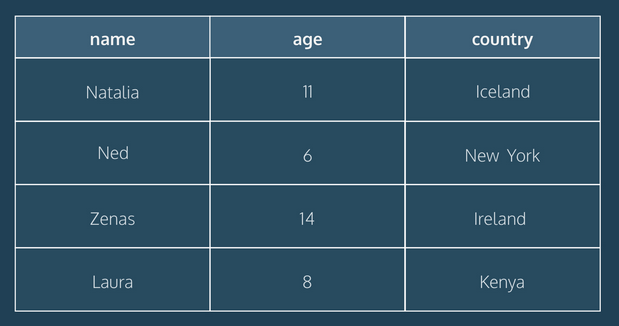
A **relational database** is a type of database. It uses a structure that allows us to identify and access data **in relation** to another piece of data in the database. Often, data in a relational database is organized into tables.

Tables: Rows and Columns

Tables can have hundreds, thousands, sometimes even millions of rows of data. These rows are often called **records**.

Tables can also have many **columns** of data. Columns are labelled with a descriptive name (say, age for example) and have a specific **data type**.

For example, a column called age may have a type of INTEGER (denoting the type of data it is meant to hold).



In the table above, there are three columns (name, age, and country).

The name and country columns store string data types, whereas age stores integer data types. The set of columns and data types make up the schema of this table.

The table also has four rows, or records, in it (one each for Natalia, Ned, Zenas, and Laura).

# What is a Relational Database Management System (RDBMS)?

A relational database management system (RDBMS) is a program that allows you to create, update, and administer a relational database. Most relational database management systems use the SQL language to access the database.

What is SQL?

SQL (**S**tructured **Q**uery **L**anguage) is a programming language used to communicate with data stored in a relational database management system. SQL syntax is similar to the English language, which makes it relatively easy to write, read, and interpret.

Many RDBMSs use SQL (and variations of SQL) to access the data in tables. For example, SQLite is a relational database management system. SQLite contains a minimal set of SQL commands (which are the same across all RDBMSs). Other RDBMSs may use other variants.

# Popular Relational Database Management Systems

SQL syntax may differ slightly depending on which RDBMS you are using. Here is a brief description of popular RDBMSs:

[MySQL](https://www.mysql.com/)

MySQL is the most popular open source SQL database. It is typically used for web application development, and often accessed using PHP.

The main advantages of MySQL are that it is easy to use, inexpensive, reliable (has been around since 1995), and has a large community of developers who can help answer questions.

Some of the disadvantages are that it has been known to suffer from poor performance when scaling, open source development has lagged since Oracle has taken control of MySQL, and it does not include some advanced features that developers may be used to.

[PostgreSQL](https://www.postgresql.org/)

PostgreSQL is an open source SQL database that is not controlled by any corporation. It is typically used for web application development.

PostgreSQL shares many of the same advantages of MySQL. It is easy to use, inexpensive, reliable and has a large community of developers. It also provides some additional features such as foreign key support without requiring complex configuration.

The main disadvantage of PostgreSQL is that it is slower in performance than other databases such as MySQL. It is also less popular than MySQL which makes it harder to come by hosts or service providers that offer managed PostgreSQL instances.

[Oracle DB](https://www.oracle.com/database/)

Oracle Corporation owns Oracle Database, and the code is not open sourced.

Oracle DB is for large applications, particularly in the banking industry. Most of the world’s top banks run Oracle applications because Oracle offers a powerful combination of technology and comprehensive, pre-integrated business applications, including essential functionality built specifically for banks.

The main disadvantage of using Oracle is that it is not free to use like its open source competitors and can be quite expensive.

[SQL Server](https://www.microsoft.com/en-us/sql-server/sql-server-2017)

Microsoft owns SQL Server. Like Oracle DB, the code is close sourced.

Large enterprise applications mostly use SQL Server.

Microsoft offers a free entry-level version called **Express** but can become very expensive as you scale your application.

[SQLite](https://www.sqlite.org/)

SQLite is a popular open source SQL database. It can store an entire database in a single file. One of the most significant advantages this provides is that all of the data can be stored locally without having to connect your database to a server.

SQLite is a popular choice for databases in cellphones, PDAs, MP3 players, set-top boxes, and other electronic gadgets. The SQL courses on Codecademy use SQLite.

For more info on SQLite, including installation instructions, read [this](https://www.codecademy.com/courses/learn-sql/articles/what-is-sqlite) article (<https://www.codecademy.com/courses/learn-sql/articles/what-is-sqlite>)

Conclusion:

Relational databases store data in tables. Tables can grow large and have a multitude of columns and records. Relational database management systems (RDBMSs) use SQL (and variants of SQL) to manage the data in these large tables. The RDBMS you use is your choice and depends on the complexity of your application.

# Manipulation on SQL

SQL, **S**tructured **Q**uery **L**anguage, is a programming language designed to manage data stored in relational databases. SQL operates through simple, declarative statements. This keeps data accurate and secure, and helps maintain the integrity of databases, regardless of size.

The SQL language is widely used today across web frameworks and database applications. Knowing SQL gives you the freedom to explore your data, and the power to make better decisions. By learning SQL, you will also learn concepts that apply to nearly every data storage system.

SELECT \* FROM celebs;

We’ll take a look at what this code means soon, for now, let’s focus on what relational databases are and how they are organized.

A relational database is a database that organizes information into one or more tables. Here, the relational database contains one table.

A table is a collection of data organized into rows and columns. Tables are sometimes referred to as relations. Here the table is celebs.

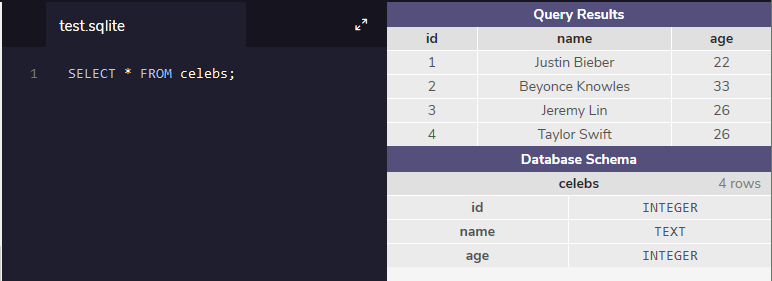
A column is a set of data values of a particular type. Here, id, name, and age are the columns.

A row is a single record in a table. The first row in the celebs table has:

* An id of 1
* A name of Justin Bieber
* An age of 22

All data stored in a relational database is of a certain data type. Some of the most common data types are:

* INTEGER, a positive or negative whole number
* TEXT, a text string
* DATE, the date formatted as YYYY-MM-DD
* REAL, a decimal value



# Statements

The code below is a SQL statement. A statement is text that the database recognizes as a valid command. Statements always end in a semicolon;

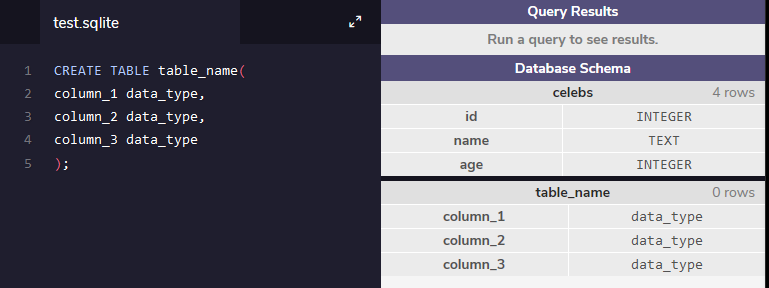
CREATE TABLE table\_name(

Column\_1 data\_type,

Column\_2 data\_type,

Column\_3 data\_type

);



Let’s break down the components of a statement:

CREATE TABLE is a clause. Clauses perform specific tasks in SQL. By convention, clauses are written in capital letters. Clauses can also be referred to as commands.

table\_name refers to the name of the table that the command is applied to.

(column\_1 data\_type, column\_2 data\_type, column\_3 data\_type) is a parameter. A parameter is a list of columns, data types, or values that are passed to a clause as an argument. Here, the parameter is a list of column names and the associated data type.

The structure of SQL statements vary. The number of lines used does not matter. A statement can be written all on one line, or split up across multiple lines if it makes it easier to read. In this course, you will become familiar with the structure of common statements.

# Create

CREATE statements allow us to create a new table in the database. You can use the CREATE statement anytime you want to create a new table from scratch. The statement below creates a new table named celebs.

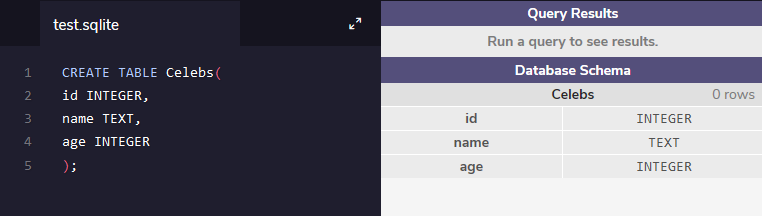
CREATE TABLE celebs(

Id INTEGER,

Name TEXT,

Age INTEGER

);



1. CREATE TABLE is a clause that tells SQL you want to create a new table.   
2. celebs is the name of the table.   
3. (id INTEGER, name TEXT, age INTEGER) is a list of parameters defining each column, or attribute in the table and its data type:

* id is the first column in the table. It stores values of data type INTEGER
* name is the second column in the table. It stores values of data type TEXT
* age is the third column in the table. It stores values of data type INTEGER

# Insert

The INSERT statement inserts a new row into a table. You can use the INSERT statement when you want to add new records. The statement below enters a record for Justin Bieber into the celebs table.

INSERT INTO Celebs(id, name, age)

VALUES (1, ‘Justin Beiber’, 29);

1. INSERT INTO is a clause that adds the specified row or rows.   
2. celebs is the name of the table the row is added to.   
3. (id, name, age) is a parameter identifying the columns that data will be inserted into.   
4. VALUES is a clause that indicates the data being inserted.   
(1, 'Justin Bieber', 22) is a parameter identifying the values being inserted.

* 1 is an integer that will be inserted into the id column
* 'Justin Bieber' is text that will be inserted into the name column
* 29 is an integer that will be inserted into the age column

INSERT INTO celebs(id, name, age)

VALUES (1, 'Justin Beiber',22);

INSERT INTO celebs (id, name, age)

VALUES (2, 'Beyonce Knowles', 33);

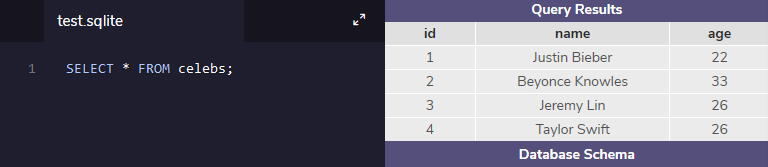
INSERT INTO celebs (id, name, age)

VALUES (3, 'Jeremy Lin', 26);

INSERT INTO celebs (id, name, age)

VALUES (4, 'Taylor Swift', 26);

SELECT \* FROM celebs;



# Select

SELECT statements are used to fetch data from a database. In the statement below, SELECT returns all data in the name column of the celebs table.

SELECT name FROM celebs;

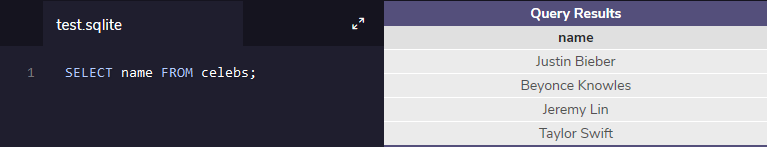
1. SELECT is a clause that indicates that the statement is a query. You will use SELECT every time you query data from a database.   
2. name specifies the column to query data from.   
3. FROM celebs specifies the name of the table to query data from. In this statement, data is queried from the celebs table.

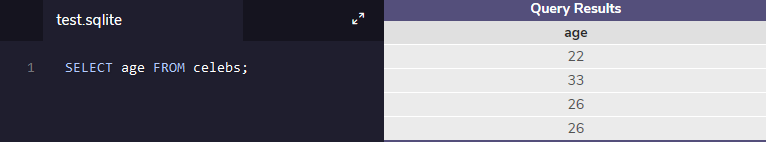
You can also query data from all columns in a table with SELECT.

SELECT \* FROM celebs;

\* is a special wildcard character that we have been using. It allows you to select every column in a table without having to name each one individually. Here, the result set contains every column in the celebs table.

SELECT statements always return a new table called the **result set**.





# Alter

The ALTER TABLE statement adds a new column to a table. You can use this command when you want to add columns to a table. The statement below adds a new column twitter\_handle to the celebs table.

ALTER TABLE celebs

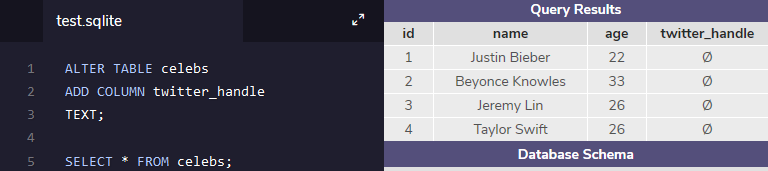
ADD COLUMN twitter\_handle TEXT;

1. ALTER TABLE is a clause that lets you make the specified changes.   
2. celebs is the name of the table that is being changed.   
3. ADD COLUMN is a clause that lets you add a new column to a table:

twitter\_handle is the name of the new column being added

TEXT is the data type for the new column

4. NULL is a special value in SQL that represents missing or unknown data. Here, the rows that existed before the column was added have NULL values for twitter\_handle.



# Update

The UPDATE statement edits a row in a table. You can use the UPDATE statement when you want to change existing records. The statement below updates the record with an id value of 4 to have the twitter\_handle @taylorswift13.

UPDATE celebs

SET twitter\_handle = '@taylorswift13'

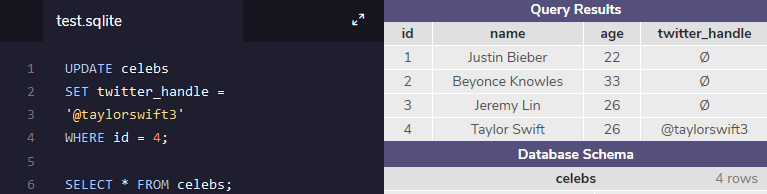
WHERE id = 4;

1. UPDATE is a clause that edits a row in the table.   
2. celebs is the name of the table.   
3. SET is a clause that indicates the column to edit.

twitter\_handle is the name of the column that is going to be updated

@taylorswift13 is the new value that is going to be inserted into the twitter\_handle column.

4. WHERE is a clause that indicates which row(s) to update with the new column value. Here the row with a 4 in the id column is the row that will have the twitter\_handle updated to @taylorswift13.



# Delete

The DELETE FROM statement deletes one or more rows from a table. You can use the statement when you want to delete existing records. The statement below deletes all records in the celeb table with no twitter\_handle:

DELETE FROM celebs

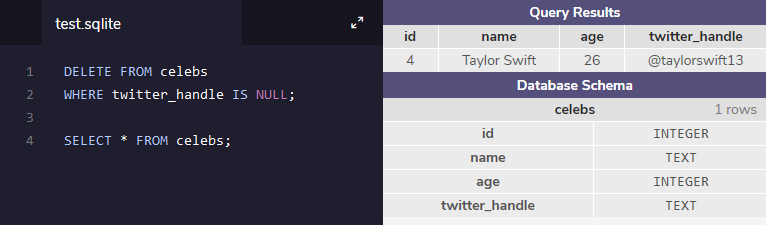
WHERE twitter\_handle IS NULL;

DELETE FROM is a clause that lets you delete rows from a table.

celebs is the name of the table we want to delete rows from.

WHERE is a clause that lets you select which rows you want to delete. Here we want to delete all of the rows where the twitter\_handle column IS NULL.

IS NULL is a condition in SQL that returns true when the value is NULL and false otherwise.



# Constraints

**Constraints** that add information about how a column can be used are invoked after specifying the data type for a column. They can be used to tell the database to reject inserted data that does not adhere to a certain restriction. The statement below sets **constraints** on the celebs table.

CREATE TABLE celebs (

id INTEGER PRIMARY KEY,

name TEXT UNIQUE,

date\_of\_birth TEXT NOT NULL,

date\_of\_death TEXT DEFAULT 'Not Applicable'

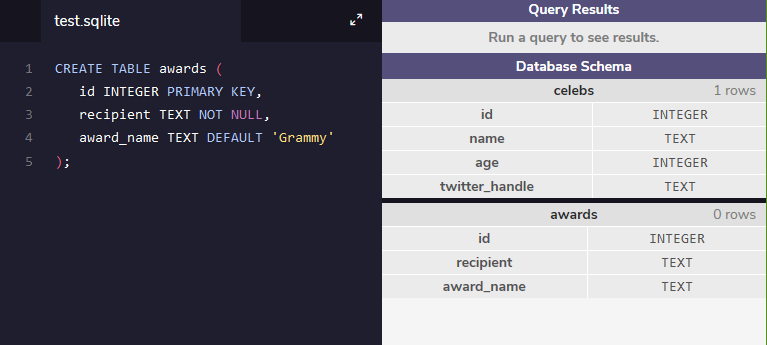
);

1. PRIMARY KEY columns can be used to uniquely identify the row. Attempts to insert a row with an identical value to a row already in the table will result in a **constraint violation** which will not allow you to insert the new row.

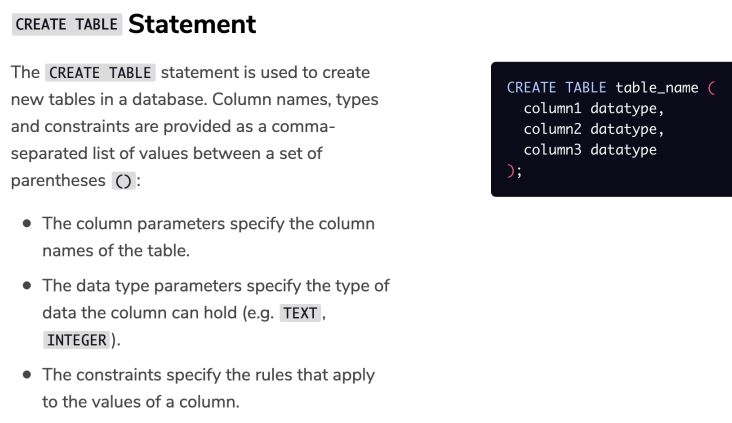
2. UNIQUE columns have a different value for every row. This is similar to PRIMARY KEY except a table can have many different UNIQUE columns.

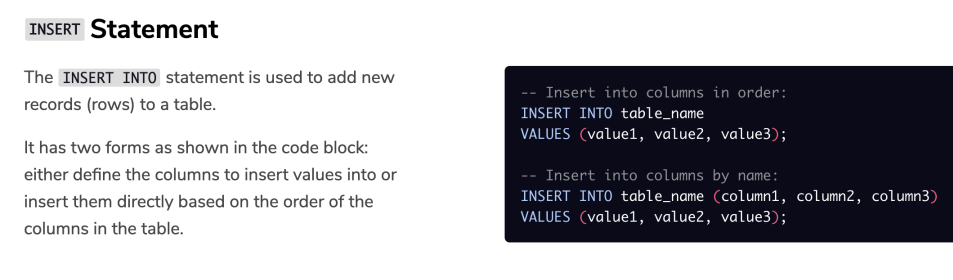
3. NOT NULL columns must have a value. Attempts to insert a row without a value for a NOT NULL column will result in a constraint violation and the new row will not be inserted.

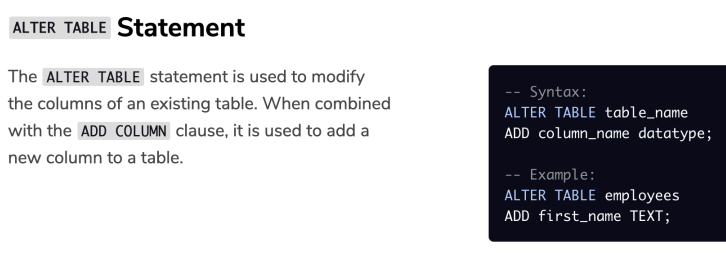
4. DEFAULT columns take an additional argument that will be the assumed value for an inserted row if the new row does not specify a value for that column.

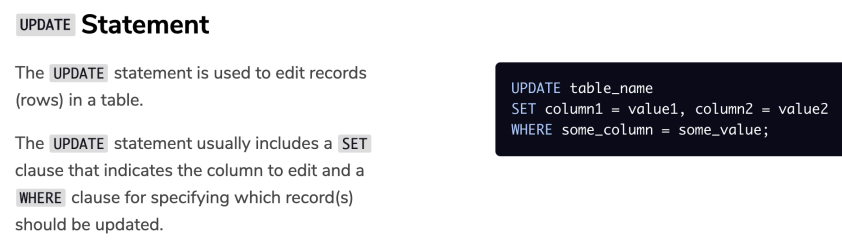


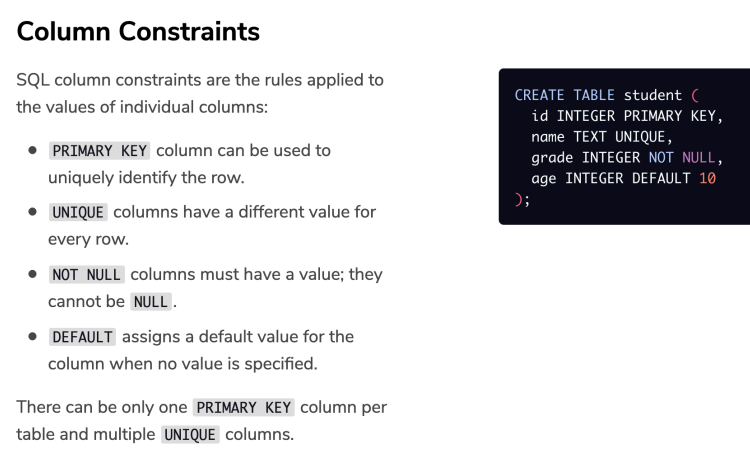
# Manipulation Review











# Create a Table

# Sample exercise:

1.Create a table named friends with three columns:

* id that stores INTEGER
* name that stores TEXT
* birthday that stores DATE

2. Beneath your current code, add Jane Doe to friends.

Her birthday is May 30th, 1990

3. Let’s make sure that Jane has been added to the database:

SELECT \* FROM friends;

Check for two things: Is friends table created? And Is Jane Doe added to it?

4. Add two of your friends to the table.

Insert an id, name, and birthday for each of them.

5. Jane Doe just got married! Her new last name is “Smith”.

Update her record in friends.

6. Add a new column named email.

7. Update the email address for everyone in your table.

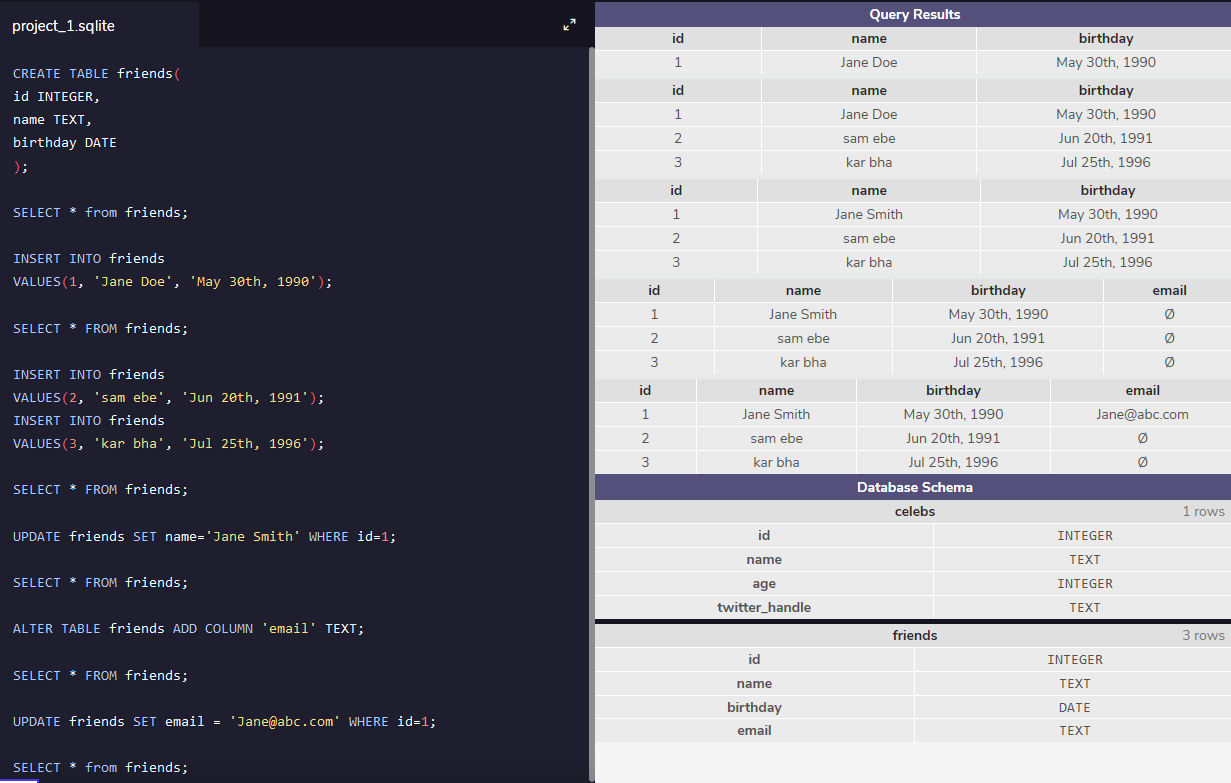
Jane Smith’s email is jane@codecademy.com.

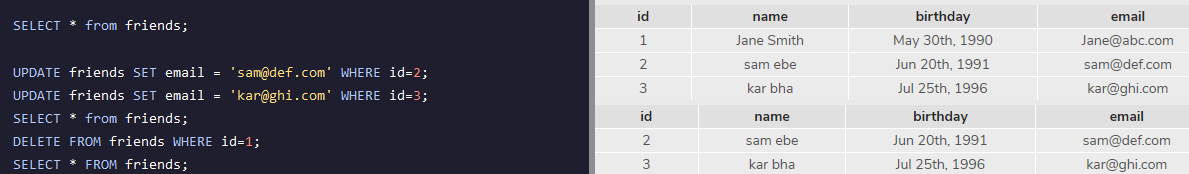
8. Wait, Jane Smith is not a real person.

Remove her from friends.

9. Let’s take a look at the result one last time:

Solution:





Queries

# Introduction

In this lesson, we will be learning different SQL commands to **query** a single table in a database.

One of the core purposes of the SQL language is to retrieve information stored in a database. This is commonly referred to as querying. Queries allow us to communicate with the database by asking questions and having the result set return data relevant to the question.

We will be querying a database with one table named movies.

# Select

Previously, we learned that SELECT is used every time you want to query data from a database and \* means **all** columns.

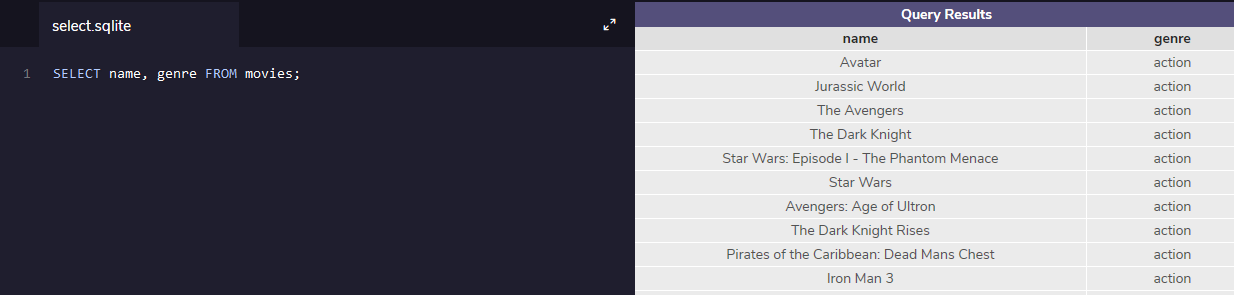
Suppose we are only interested in two of the columns. We can select individual columns by their names (separated by a comma):

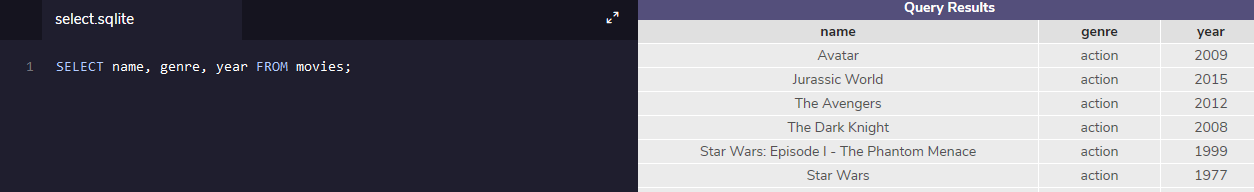
SELECT column1, column2

FROM table\_name;

To make it easier to read, we moved FROM to another line.

Line breaks don’t mean anything specific in SQL. We could write this entire query in one line, and it would run just fine.





# As

Knowing how SELECT works, suppose we have the code below:

SELECT name AS 'Titles'

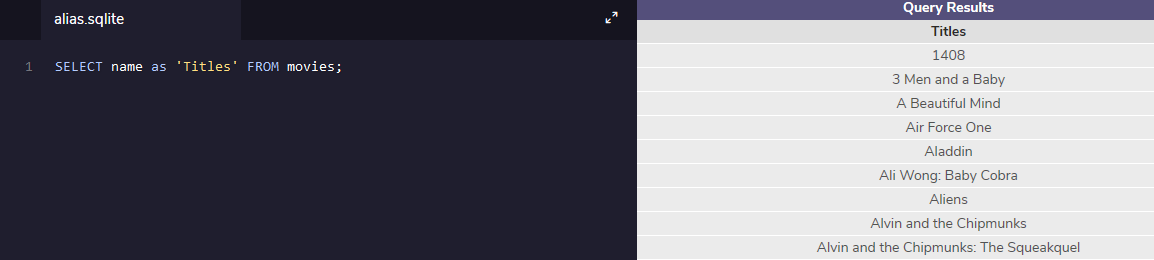
FROM movies;

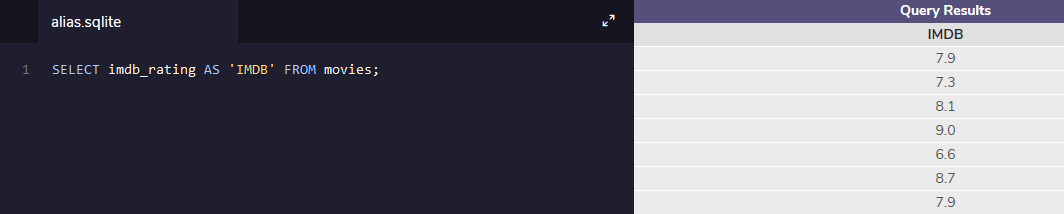
AS is a keyword in SQL that allows you to **rename** a column or table using an alias. The new name can be anything you want as long as you put it inside of single quotes. Here we renamed the name column as Titles.

Some important things to note:

Although it’s not always necessary, it’s best practice to surround your aliases with single quotes.

When using AS, the columns are not being renamed in the table. The aliases only appear in the result.





# Distinct

When we are examining data in a table, it can be helpful to know what **distinct** values exist in a particular column.

DISTINCT is used to return unique values in the output. It filters out all duplicate values in the specified column(s).

For instance,

SELECT tools

FROM inventory;

might produce:

| tools |
| --- |
| Hammer |
| Nails |
| Nails |
| Nails |

By adding DISTINCT before the column name,

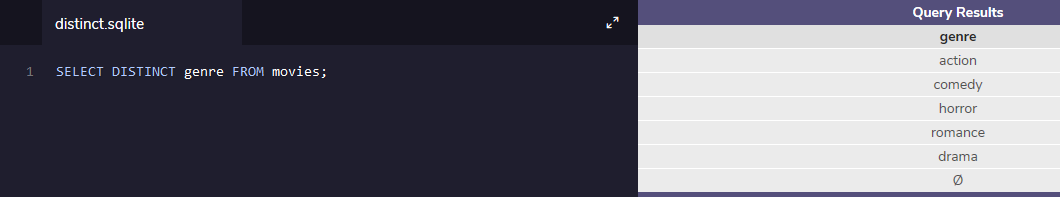
SELECT DISTINCT tools

FROM inventory;

the result would now be:

| tools |
| --- |
| Hammer |
| Nails |

Filtering the results of a query is an important skill in SQL. It is easier to see the different possible genres in the movie table after the data has been filtered than to scan every row in the table.



# Where

We can restrict our query results using the WHERE clause in order to obtain only the information we want.

Following this format, the statement below filters the result set to only include top rated movies (IMDb ratings greater than 8):

SELECT \*

FROM movies

WHERE imdb\_rating > 8;

How does it work?

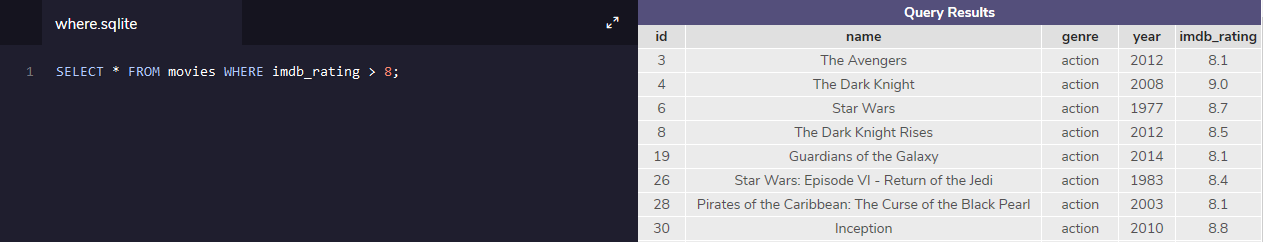
WHERE clause filters the result set to only include rows where the following **condition** is true.

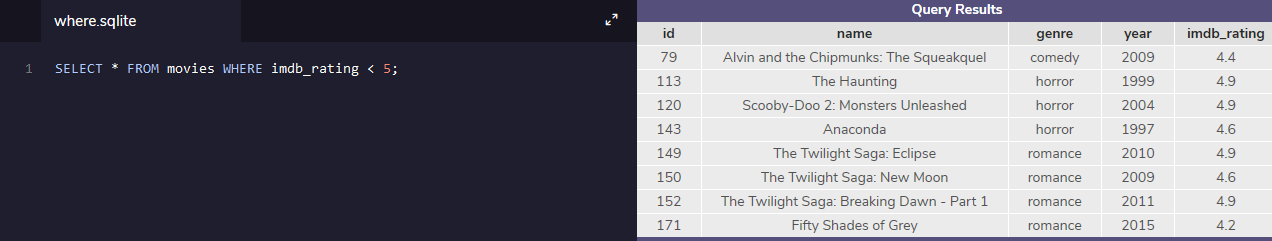
imdb\_rating > 8 is the condition. Here, only rows with a value greater than 8 in the imdb\_rating column will be returned.

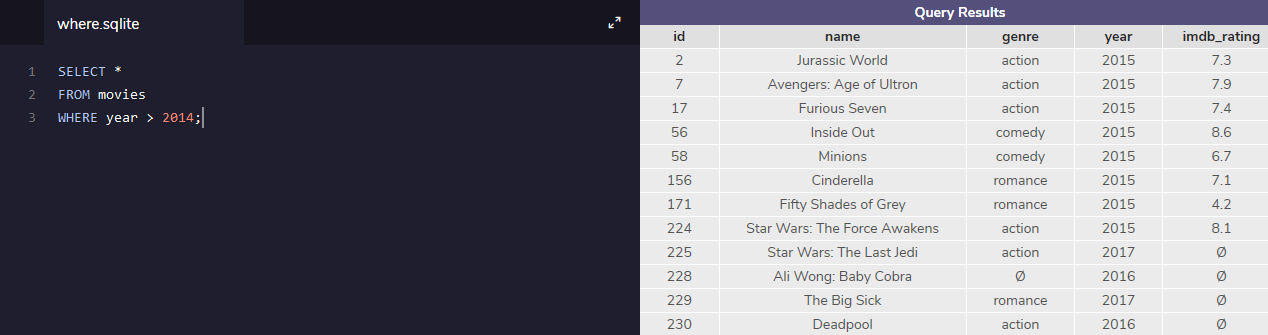
The > is an **operator**. Operators create a condition that can be evaluated as either **true** or **false**.

Comparison operators used with the WHERE clause are:

* = equal to
* != not equal to
* > greater than
* < less than
* >= greater than or equal to
* <= less than or equal to







# Like I

LIKE can be a useful operator when you want to compare similar values.

The movies table contains two films with similar titles, ‘Se7en’ and ‘Seven’.

How could we select all movies that start with ‘Se’ and end with ‘en’ and have exactly one character in the middle?

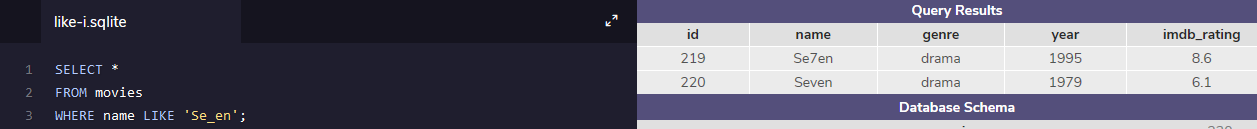
SELECT \*

FROM movies

WHERE name LIKE 'Se\_en';

* LIKE is a special operator used with the WHERE clause to search for a specific pattern in a column.
* name LIKE 'Se\_en' is a condition evaluating the name column for a specific pattern.
* Se\_en represents a pattern with a **wildcard** character.

The \_ means you can substitute any individual character here without breaking the pattern. The names Seven and Se7en both match this pattern.



# Like II

The percentage sign % is another wildcard character that can be used with LIKE.

This statement below filters the result set to only include movies with names that begin with the letter ‘A’:

SELECT \*

FROM movies

WHERE name LIKE 'A%';

% is a wildcard character that matches zero or more missing letters in the pattern. For example:

* A% matches all movies with names that begin with letter ‘A’
* %a matches all movies that end with ‘a’

We can also use % both before and after a pattern:

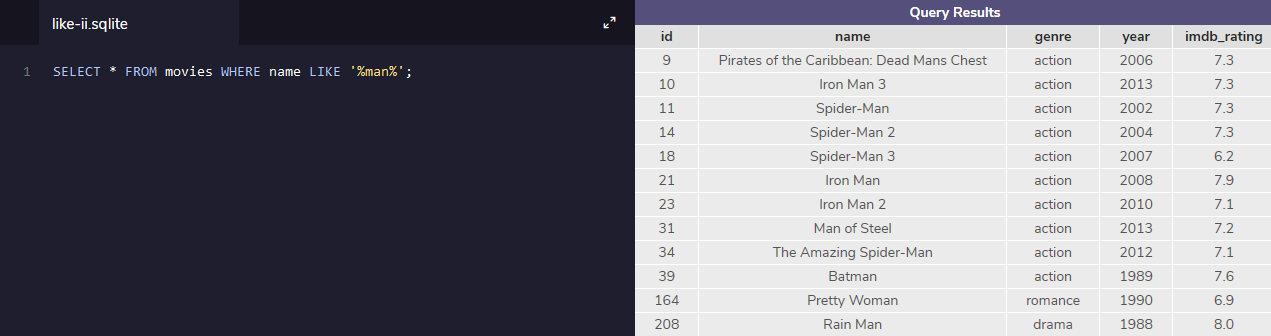
SELECT \*

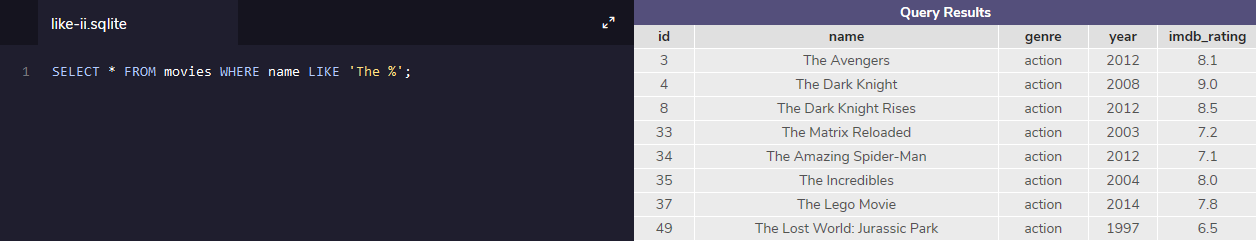
FROM movies

WHERE name LIKE '%man%';

Here, any movie that **contains** the word ‘man’ in its name will be returned in the result.

LIKE is not case sensitive. ‘Batman’ and ‘Man of Steel’ will both appear in the result of the query above.





# Is Null

Unknown values are indicated by NULL.

It is not possible to test for NULL values with comparison operators, such as = and !=.

Instead, we will have to use these operators:

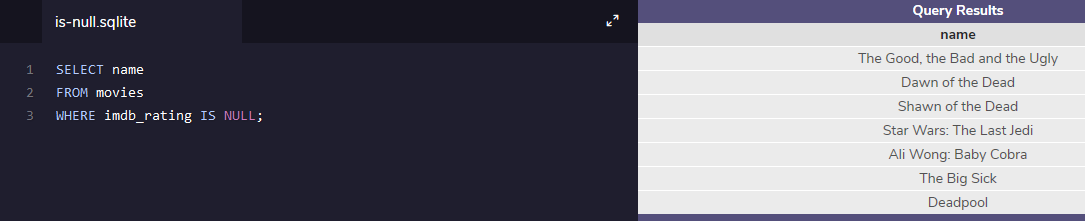
* IS NULL
* IS NOT NULL

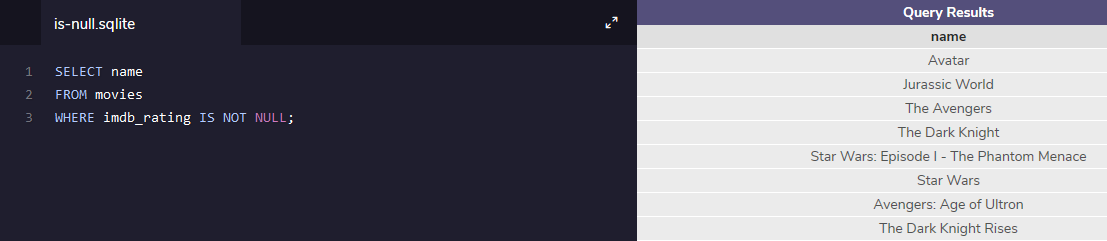
To filter for all movies **with** an IMDb rating:

SELECT name

FROM movies

WHERE imdb\_rating IS NOT NULL;





# Between

The BETWEEN operator is used in a WHERE clause to filter the result set within a certain **range**. It accepts two values that are either numbers, text or dates.

For example, this statement filters the result set to only include movies with years from 1990 up to, **and including** 1999.

SELECT \*

FROM movies

WHERE year BETWEEN 1990 AND 1999;

When the values are text, BETWEEN filters the result set for within the alphabetical range.

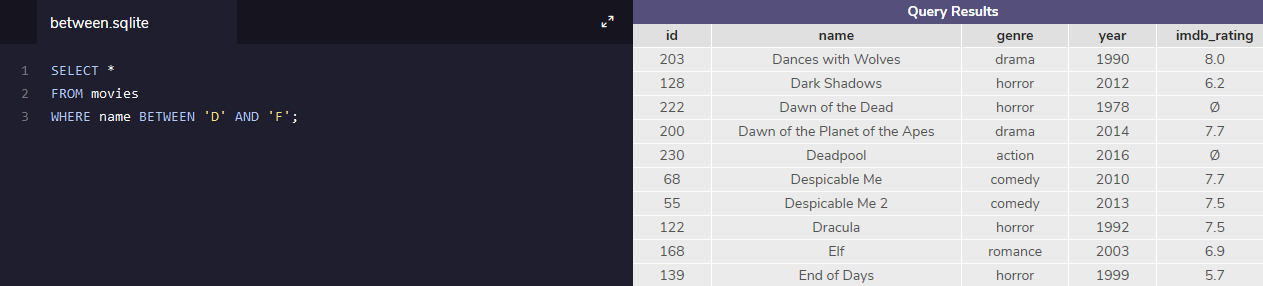
In this statement, BETWEEN filters the result set to only include movies with names that begin with the letter ‘A’ up to, **but not including** ones that begin with ‘J’.

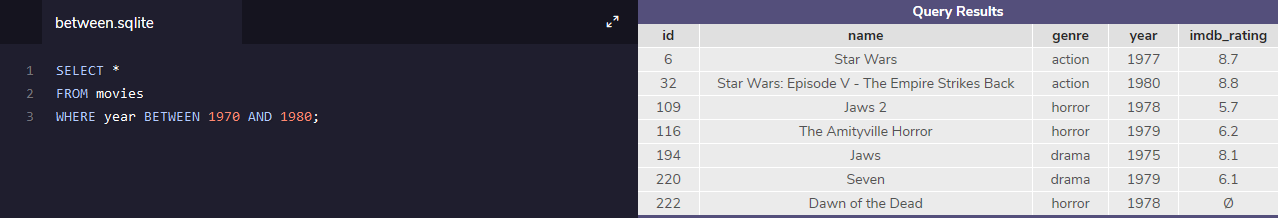
SELECT \*

FROM movies

WHERE name BETWEEN 'A' AND 'J';

However, if a movie has a name of simply ‘J’, it would actually match. This is because BETWEEN goes **up to** the second value — up to ‘J’. So the movie named ‘J’ would be included in the result set but not ‘Jaws’.





# And

Sometimes we want to **combine multiple conditions** in a WHERE clause to make the result set more specific and useful.

One way of doing this is to use the AND operator. Here, we use the AND operator to only return 90’s romance movies.

SELECT \*

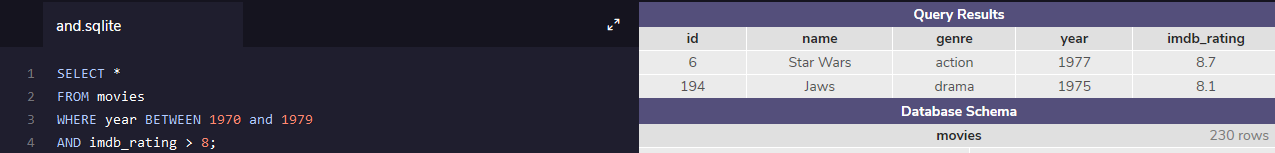
FROM movies

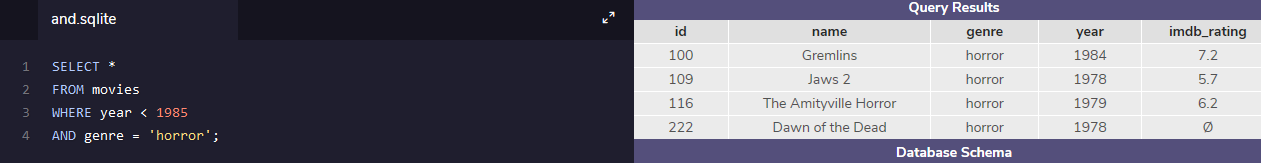
WHERE year BETWEEN 1990 AND 1999

AND genre = 'romance';

* year BETWEEN 1990 AND 1999 is the 1st condition.
* genre = 'romance' is the 2nd condition.
* AND combines the two conditions.

With AND, both conditions must be true for the row to be included in the result.





# Or

Similar to AND, the OR operator can also be used to combine multiple conditions in WHERE, but there is a fundamental difference:

* AND operator displays a row if **all** the conditions are true.
* OR operator displays a row if **any** condition is true.

Suppose we want to check out a new movie or something action-packed:

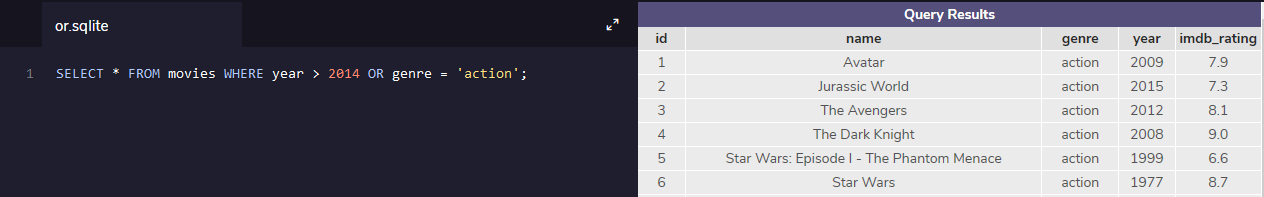
SELECT \*

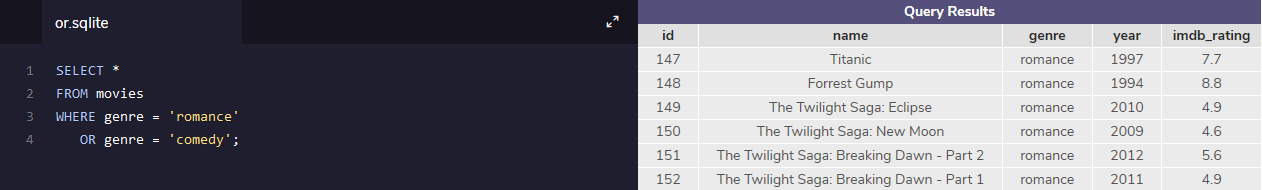
FROM movies

WHERE year > 2014

OR genre = 'action';

* year > 2014 is the 1st condition.
* genre = 'action' is the 2nd condition. OR combines the two conditions.





# Order By

It is often useful to list the data in our result set in a particular order.

We can **sort** the results using ORDER BY, either alphabetically or numerically. Sorting the results often makes the data more useful and easier to analyze.

For example, if we want to sort everything by the movie’s title from A through Z:

SELECT \*

FROM movies

ORDER BY name;

* ORDER BY is a clause that indicates you want to sort the result set by a particular column.
* name is the specified column.

Sometimes we want to sort things in a decreasing order. For example, if we want to select all of the well-received movies, sorted from highest to lowest by their year:

SELECT \*

FROM movies

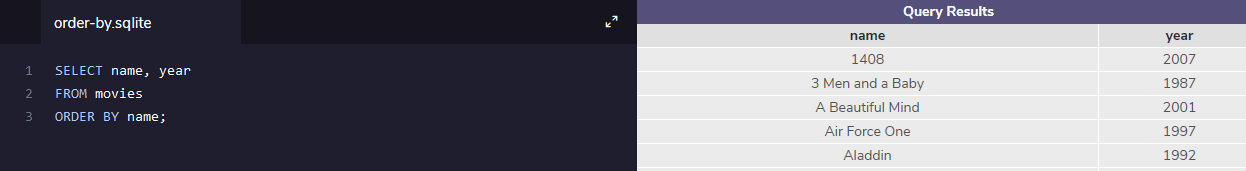
WHERE imdb\_rating > 8

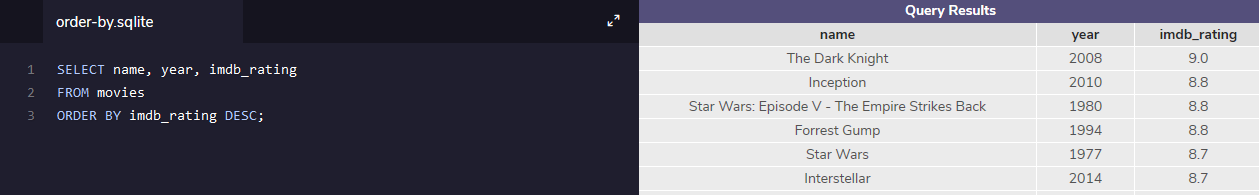
ORDER BY year DESC;

* DESC is a keyword used in ORDER BY to sort the results in **descending order** (high to low or Z-A).
* ASC is a keyword used in ORDER BY to sort the results in **ascending** order (low to high or A-Z).

The column that we ORDER BY doesn’t even have to be one of the columns that we’re displaying.

Note: ORDER BY always goes after WHERE (if WHERE is present).





# Limit

We’ve been working with a fairly small table (fewer than 250 rows), but most SQL tables contain hundreds of thousands of records. In those situations, it becomes important to cap the number of rows in the result.

For instance, imagine that we just want to see a few examples of records.

SELECT \*

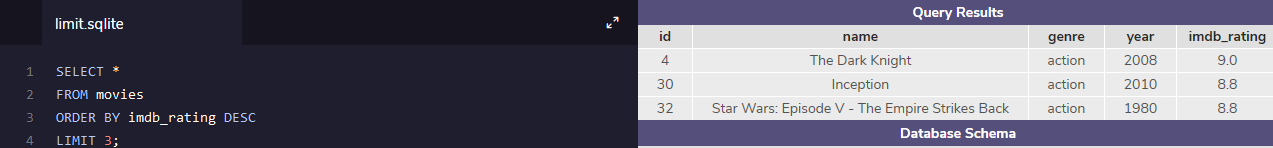
FROM movies

LIMIT 10;

LIMIT is a clause that lets you specify the maximum number of rows the result set will have. This saves space on our screen and makes our queries run faster.

Here, we specify that the result set can’t have more than 10 rows.

LIMIT always goes at the very end of the query. Also, it is not supported in all SQL databases.



# Case

A CASE statement allows us to create different outputs (usually in the SELECT statement). It is SQL’s way of handling [if-then](https://en.wikipedia.org/wiki/Conditional_(computer_programming)) logic.

Suppose we want to condense the ratings in movies to three levels:

* If the rating is above 8, then it is Fantastic.
* If the rating is above 6, then it is Poorly Received.
* Else, Avoid at All Costs.

SELECT name,

CASE

WHEN imdb\_rating > 8 THEN 'Fantastic'

WHEN imdb\_rating > 6 THEN 'Poorly Received'

ELSE 'Avoid at All Costs'

END

FROM movies;

* Each WHEN tests a condition and the following THEN gives us the string if the condition is true.
* The ELSE gives us the string if **all** the above conditions are false.
* The CASE statement must end with END.

In the result, you have to scroll right because the column name is very long. To shorten it, we can rename the column to ‘Review’ using AS:

SELECT name,

CASE

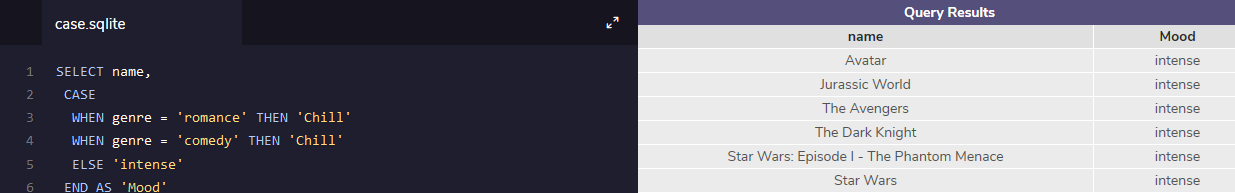
WHEN imdb\_rating > 8 THEN 'Fantastic'

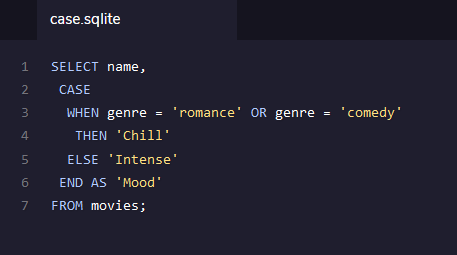
WHEN imdb\_rating > 6 THEN 'Poorly Received'

ELSE 'Avoid at All Costs'

END AS 'Review'

FROM movies;





# Review

Let’s summarize:

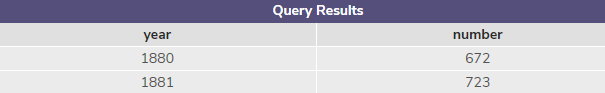
* SELECT is the clause we use every time we want to query information from a database.
* AS renames a column or table.
* DISTINCT return unique values.
* WHERE is a popular command that lets you filter the results of the query based on conditions that you specify?
* LIKE and BETWEEN are special operators.
* AND and OR combines multiple conditions.
* ORDER BY sorts the result.
* LIMIT specifies the maximum number of rows that the query will return.
* CASE creates different outputs.

Code Challenge:

* Find the number of girls who were named Lillian for the full span of time of the database. Select only the year and number columns.

SELECT year,number FROM babies

WHERE name = 'Lillian';



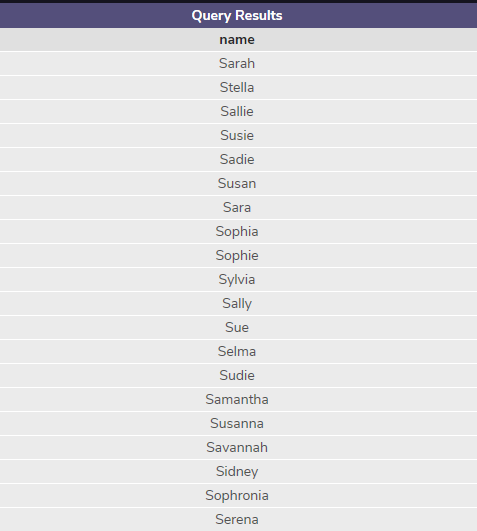
* Find 20 distinct names that start with ‘S’. Select only the name column.

SELECT DISTINCT name

FROM babies

WHERE name LIKE 'S%'

LIMIT 20;



* What are the top 10 names in 1880? Select the name, gender, and number columns.

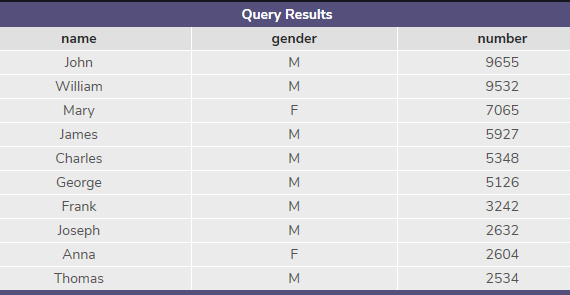
SELECT name, gender, number

FROM babies

WHERE year = 1880

ORDER BY number DESC

LIMIT 10;



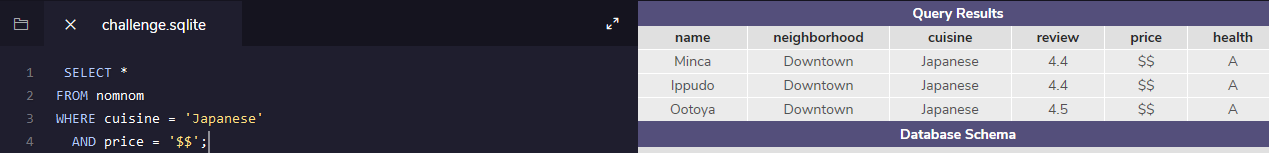
* Suppose your friend Jaime wants to go out to Japanese, but you’re on a budget. Return all the restaurants that are Japanese and $$. Select all the columns.

SELECT \*

FROM nomnom

WHERE cuisine = ‘Japanese’

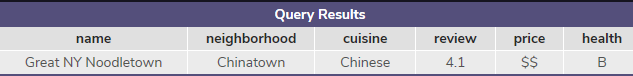
AND price = ‘$$’;



* Your roommate Bevers can’t remember the exact name of a restaurant he went to but he knows it contains the word ‘noodle’ in it. Can you find it for him using a query? Select all the columns.

SELECT \* FROM nomnom

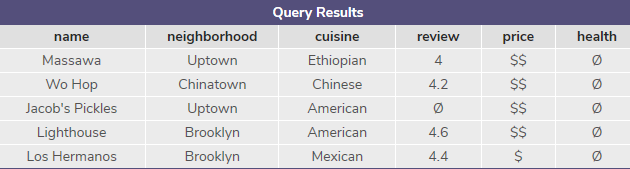
WHERE name LIKE '%noodle%';



* Some of the restaurants have not been inspected yet or are currently appealing their health grade score. Find the restaurants that have empty health values. Select all the columns.

SELECT \* FROM nomnom

WHERE health ISNULL;

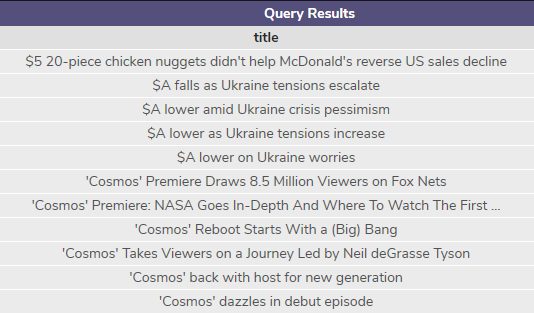


* Order the table by title (from A-Z). Select only the title and publisher columns.

SELECT title, publisher

FROM news

ORDER BY title ASC;



* Which article names have the word 'bitcoin' in it? Select all the columns.

SELECT \* FROM news

WHERE title LIKE '%bitcoin%';



* The category column contains the article category:
* 'b' stands for Business and 't' stands for Technology
* What are the 20 business articles published most recently? Select all the columns.

SELECT \* FROM news

WHERE category = 'b' OR 't'

ORDER BY timestamp DESC

LIMIT 20;



# Aggregate Functions

Calculations performed on multiple rows of a table are called **aggregates**.

Here is a quick preview of some important:

* COUNT(): count the number of rows
* SUM(): the sum of the values in a column
* MAX()/MIN(): the largest/smallest value
* AVG(): the average of the values in a column
* ROUND(): round the values in the column

# Count

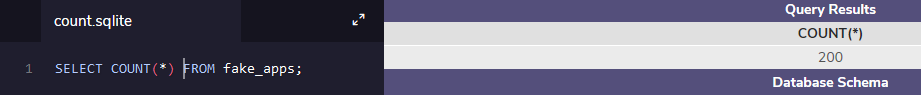
The fastest way to calculate how many rows are in a table is to use the COUNT() function.

COUNT() is a function that takes the name of a column as an argument and counts the number of non-empty values in that column.

SELECT COUNT(\*)

FROM table\_name;

Here, we want to count every row, so we pass \* as an argument inside the parenthesis.



# Sum

SQL makes it easy to add all values in a particular column using SUM().

SUM() is a function that takes the name of a column as an argument and returns the sum of all the values in that column.

SELECT SUM(downloads)

FROM fake\_apps;

This adds all values in downloads column.



# Max / Min

The MAX() and MIN() functions return the highest and lowest values in a column, resp.

How many downloads does the most popular app have?

SELECT MAX(downloads)

FROM fake\_apps;

The most popular app has 31,090 downloads!

MAX() takes the name of a column as an argument and returns the largest value in that column. Here, we returned the largest value in the downloads column.

MIN() works the same way but it does the exact opposite; it returns the smallest value.



# Average

SQL uses the AVG() function to quickly calculate the average value of a particular column.

The statement below returns the average number of downloads for an app in our database:

SELECT AVG(downloads)

FROM fake\_apps;

The AVG() function works by taking a column name as an argument and returns the average value for that column.



# Round

By default, SQL tries to be as precise as possible without rounding. We can make the result table easier to read using the ROUND() function.

ROUND() function takes two arguments inside the parenthesis:

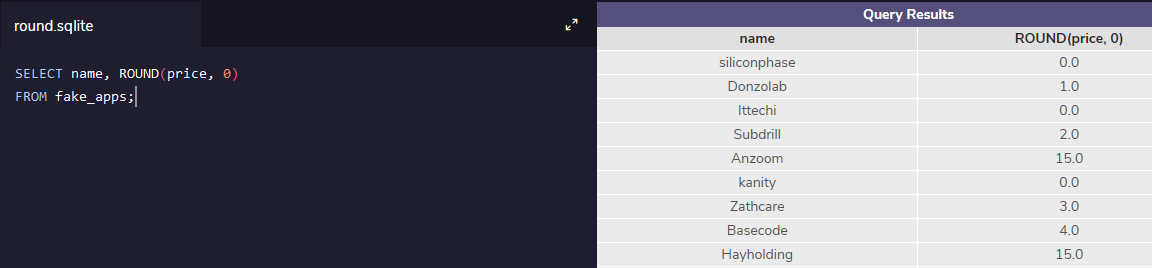
* a column name
* an integer

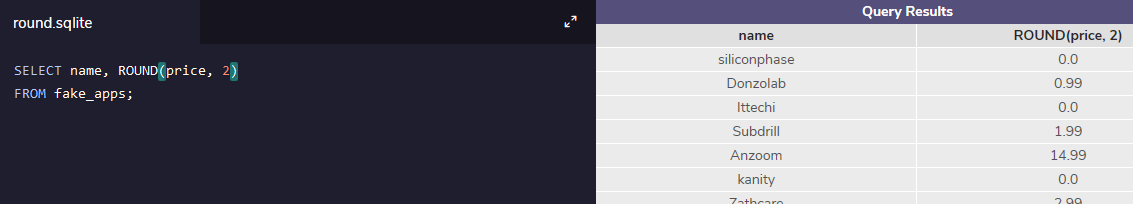
It rounds the values in the column to the number of decimal places specified by the integer.

SELECT ROUND(price, 0)

FROM fake\_apps;

Here, we pass the column price and integer 0 as arguments. SQL rounds the values in the column to 0 decimal places in the output.







# Group By I

For instance, we might want to know the mean IMDb ratings for all movies each year. We could calculate each number by a series of queries with different WHERE statements, like so:

SELECT AVG(imdb\_rating)

FROM movies

WHERE year = 1999;

SELECT AVG(imdb\_rating)

FROM movies

WHERE year = 2000;

SELECT AVG(imdb\_rating)

FROM movies

WHERE year = 2001;

and so on.

We can use GROUP BY to do this in a single step:

SELECT year,

AVG(imdb\_rating)

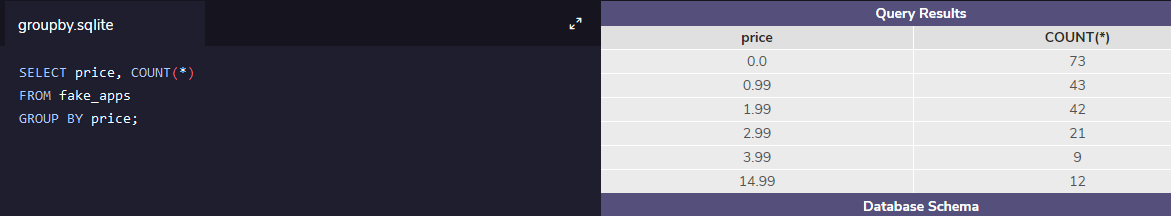
FROM movies

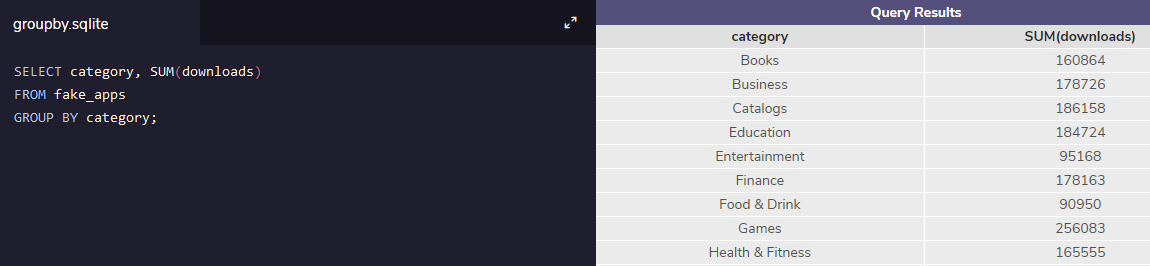
GROUP BY year

ORDER BY year;

GROUP BY is a clause in SQL that is used with aggregate functions. It is used in collaboration with the SELECT statement to arrange identical data into **groups**.

The GROUP BY statement comes after any WHERE statements, but before ORDER BY or LIMIT.





# Group By II

Sometimes, we want to GROUP BY a calculation done on a column.

For instance, we might want to know how many movies have IMDb ratings that round to 1, 2, 3, 4, 5. We could do this using the following syntax:

SELECT ROUND(imdb\_rating),

COUNT(name)

FROM movies

GROUP BY ROUND(imdb\_rating)

ORDER BY ROUND(imdb\_rating);

However, this query may be time-consuming to write and more prone to error.

SQL lets us use column reference(s) in our GROUP BY that will make our lives easier.

* 1 is the first column selected
* 2 is the second column selected
* 3 is the third column selected and so on.

The following query is equivalent to the one above:

SELECT ROUND(imdb\_rating),

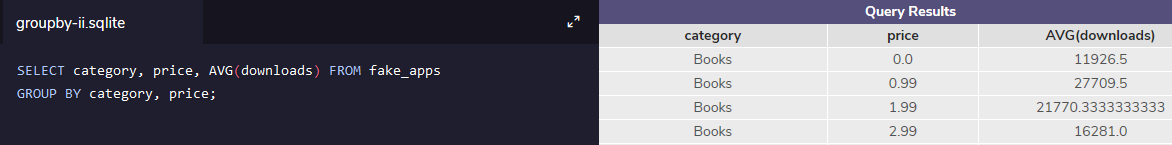
COUNT(name)

FROM movies

GROUP BY 1

ORDER BY 1;

Here, the 1 refers to the first column in our SELECT statement, ROUND(imdb\_rating).



# Having

In addition to being able to group data using GROUP BY, SQL also allows you to filter which groups to include and which to exclude.

For instance, imagine that we want to see how many movies of different genres were produced each year, but we only care about years and genres with at least 10 movies.

We can’t use WHERE here because we don’t want to filter the rows; we want to **filter groups**.

This is where HAVING comes in.

HAVING is very similar to WHERE. In fact, all types of WHERE clauses you learned about thus far can be used with HAVING.

We can use the following for the problem:

SELECT year,

genre,

COUNT(name)

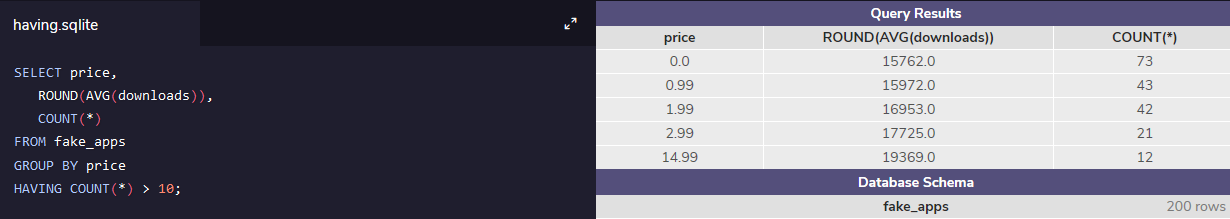
FROM movies

GROUP BY 1, 2

HAVING COUNT(name) > 10;

* When we want to limit the results of a query based on values of the individual rows, use WHERE.
* When we want to limit the results of a query based on an aggregate property, use HAVING.

HAVING statement always comes after GROUP BY, but before ORDER BY and LIMIT.



# Multiple Tables

# Combining Tables Manually with JOIN

Let’s return to our magazine company. Suppose we have the three tables:

* orders
* subscriptions
* customers

If we just look at the orders table, we can’t really tell what’s happened in each order. However, if we refer to the other tables, we can get a complete picture.

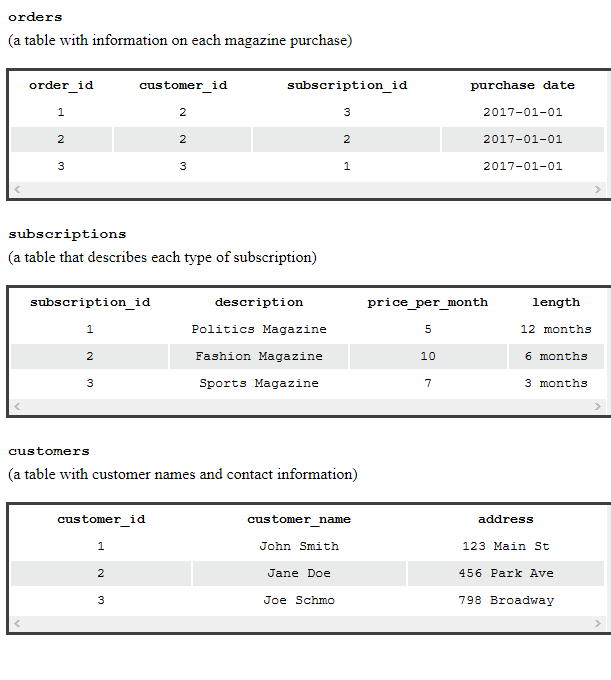
Let’s examine the order with an order\_id of 2. It was purchased by the customer with a customer\_id of 2.

To find out the customer’s name, we look at the customers table and look for the item with a customer\_id value of 2. We can see that Customer 2’s name is ‘Jane Doe’ and that she lives at ‘456 Park Ave’.

Doing this kind of matching is called joining two tables.

Combining tables manually is time-consuming. Luckily, SQL gives us an easy sequence for this: it’s called a JOIN.

If we want to combine orders and customers, we would type:



SELECT \*

FROM orders

JOIN customers

ON orders.customer\_id = customers.customer\_id;



Let’s break down this command:

* The first line selects all columns from our combined table. If we only want to select certain columns, we can specify which ones we want.
* The second line specifies the first table that we want to look in, orders
* The third line uses JOIN to say that we want to combine information from orders with customers.
* The fourth line tells us how to combine the two tables. We want to match orders table’s customer\_id column with customers table’s customer\_id column.

Because column names are often repeated across multiple tables, we use the syntax table\_name.column\_name to be sure that our requests for columns are unambiguous. In our example, we use this syntax in the ON statement, but we will also use it in the SELECT or any other statement where we refer to column names.

For example: Instead of selecting all the columns using \*, if we only wanted to select orders table’s order\_id column and customers table’s customer\_name column, we could use the following query:

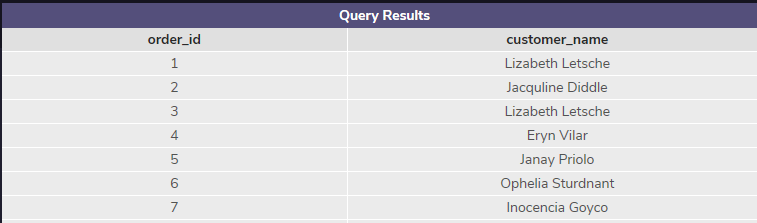
SELECT orders.order\_id,

customers.customer\_name

FROM orders

JOIN customers

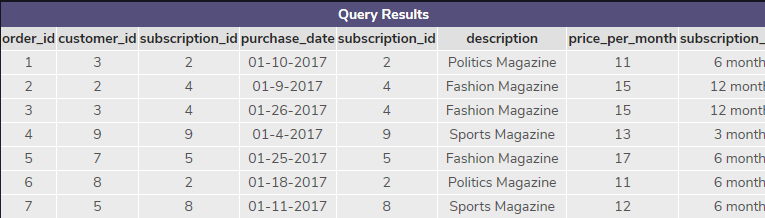
ON orders.customer\_id = customers.customer\_id;



SELECT \* FROM orders

JOIN subscriptions

ON orders.subscription\_id = subscriptions.subscription\_id;



# Inner Joins

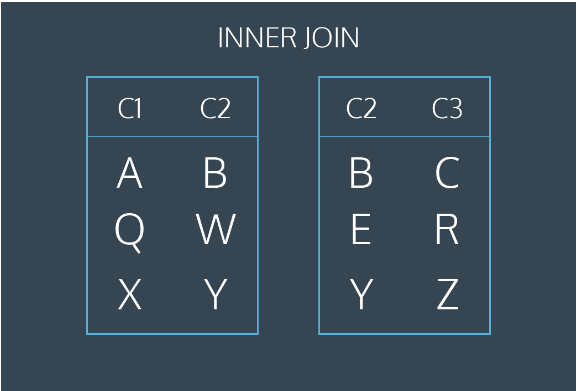
Let’s revisit how we joined orders and customers. For every possible value of customer\_id in orders, there was a corresponding row of customers with the same customer\_id.

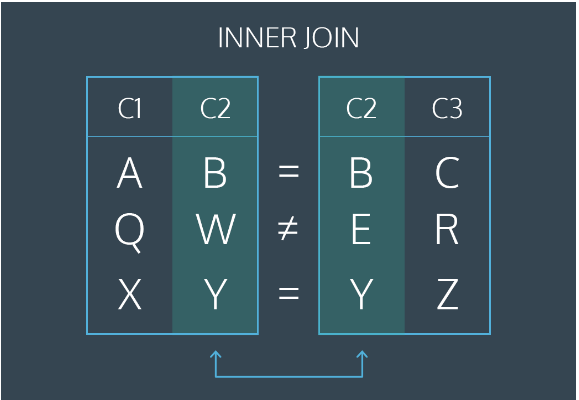
What if that wasn’t true?

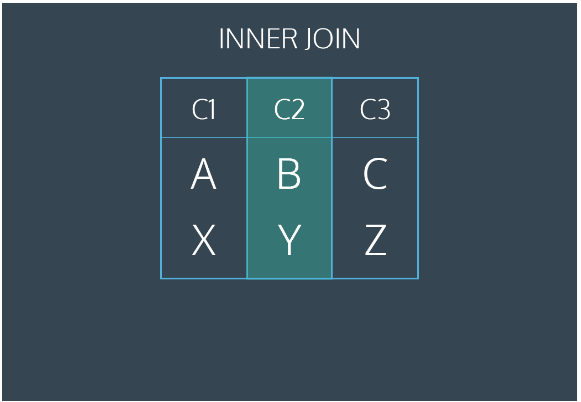
For instance, imagine that our customers table was out of date, and was missing any information on customer 11. If that customer had an order in orders, what would happen when we joined the tables?

When we perform a simple JOIN (often called an **inner join**) our result only includes rows that match our ON condition.

Consider the following animation, which illustrates an inner join of two tables on table1.c2 = table2.c2:







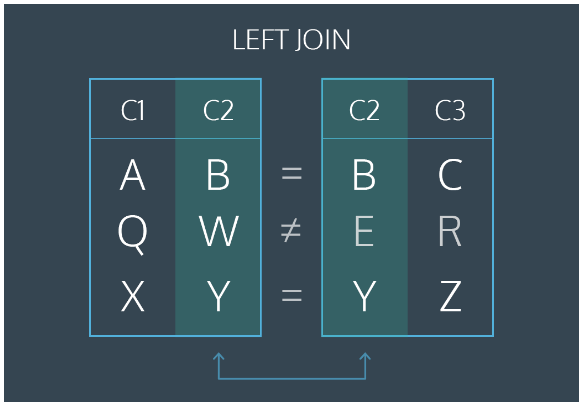
The first and last rows have matching values of c2. The middle rows do not match. The final result has all values from the first and last rows but does not include the non-matching middle row.

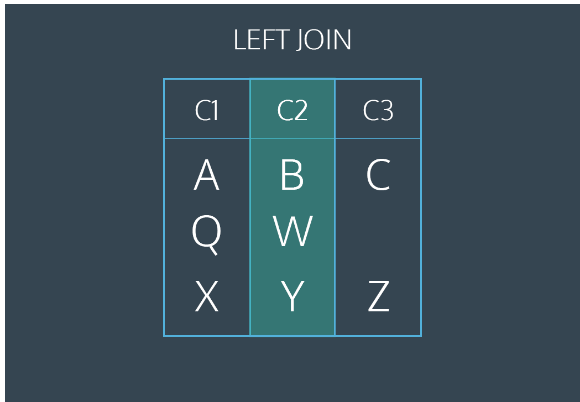
# Left Joins

What if we want to combine two tables and keep some of the un-matched rows?

SQL lets us do this through a command called LEFT JOIN. A **left join** will keep all rows from the first table, regardless of whether there is a matching row in the second table.

Consider the following animation:





The first and last rows have matching values of c2. The middle rows do not match. The final result will keep all rows of the first table but will omit the un-matched row from the second table.

This animation represents a table operation produced by the following command:

SELECT \*

FROM table1

LEFT JOIN table2

ON table1.c2 = table2.c2;

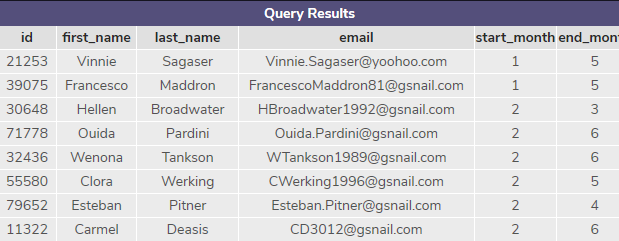
* The first line selects all columns from both tables.
* The second line selects table1 (the “left” table).
* The third line performs a LEFT JOIN on table2 (the “right” table).
* The fourth line tells SQL how to perform the join (by looking for matching values in column c2).

SELECT \*

FROM newspaper

LEFT JOIN online

ON newspaper.id = online.id;



# Primary Key vs Foreign Key

Let’s return to our example of the magazine subscriptions. Recall that we had three tables: orders, subscriptions, and customers.

Each of these tables has a column that uniquely identifies each row of that table:

* order\_id for orders
* subscription\_id for subscriptions
* customer\_id for customers

These special columns are called **primary keys**.

Primary keys have a few requirements:

* None of the values can be NULL.
* Each value must be unique (i.e., you can’t have two customers with the same customer\_id in the customers table).
* A table cannot have more than one primary key column.

Let’s reexamine the orders table:

| order\_id | customer\_id | subscription\_id | purchase\_date |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 2017-01-01 |
| 2 | 2 | 2 | 2017-01-01 |
| 3 | 3 | 1 | 2017-01-01 |

Note that customer\_id (the primary key for customers) and subscription\_id (the primary key for subscriptions) both appear in this.

When the primary key for one table appears in a different table, it is called a **foreign key**.

So customer\_id is a primary key when it appears in customers, but a foreign key when it appears in orders.

In this example, our primary keys all had somewhat descriptive names. Generally, the primary key will just be called id. Foreign keys will have more descriptive names.

**Why is this important?** The most common types of joins will be joining a foreign key from one table with the primary key from another table. For instance, when we join orders and customers, we join on customer\_id, which is a foreign key in orders and the primary key in customers.

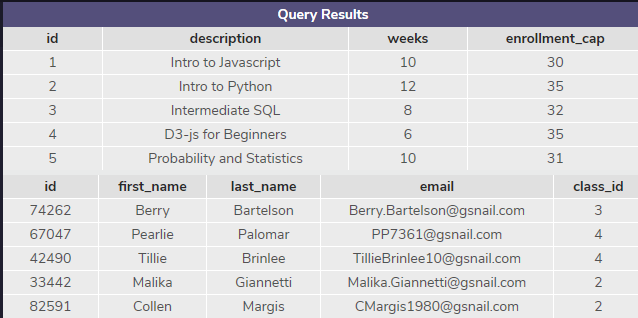
Suppose Columbia University has two tables in their database:

* The classes table contains information on the classes that the school offers. Its primary key is id.
* The students table contains information on all students in the school. Its primary key is id. It contains the foreign key class\_id, which corresponds to the primary key of classes.

Perform an inner join of classes and students using the primary and foreign keys described above, and select all the columns.

SELECT \* FROM classes;

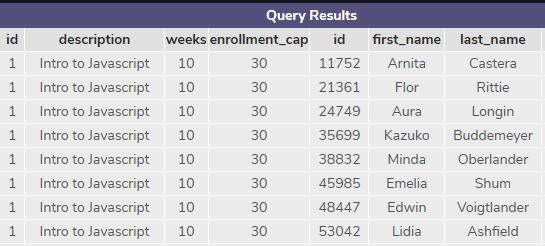
SELECT \* FROM students LIMIT 5;



SELECT \* FROM classes

JOIN students

ON classes.id = students.class\_id;



# Cross Join

Sometimes, we just want to combine all rows of one table with all rows of another table.

For instance, if we had a table of shirts and a table of pants, we might want to know all the possible combinations to create different outfits.

Our code might look like this:

SELECT shirts.shirt\_color,

pants.pants\_color

FROM shirts

CROSS JOIN pants;

* The first two lines select the columns shirt\_color and pants\_color.
* The third line pulls data from the table shirts.
* The fourth line performs a CROSS JOIN with pants.

Notice that cross joins don’t require an ON statement. You’re not really joining on any columns!

If we have 3 different shirts (white, grey, and olive) and 2 different pants (light denim and black), the results might look like this:

| shirt\_color | pants\_color |
| --- | --- |
| white | light denim |
| white | black |
| grey | light denim |
| grey | black |
| olive | light denim |
| olive | black |

3 shirts × 2 pants = 6 combinations!

A more common usage of CROSS JOIN is when we need to compare each row of a table to a list of values.

* start\_month: the first month where the customer subscribed to the print newspaper (i.e., 2 for February)
* end\_month: the final month where the customer subscribed to the print newspaper

Suppose we wanted to know how many users were subscribed during each month of the year. For each month (1, 2, 3) we would need to know if a user was subscribed. Follow the steps below to see how we can use a CROSS JOIN to solve this problem.

SELECT month,

COUNT(\*) AS 'subscribers'

FROM newspaper

CROSS JOIN months

WHERE start\_month <= month

AND end\_month >= month

GROUP BY month;

# Union

Sometimes we just want to stack one dataset on top of the other. Well, the UNION operator allows us to do that.

Suppose we have two tables and they have the same columns.

table1:

| pokemon | type |
| --- | --- |
| Bulbasaur | Grass |
| Charmander | Fire |
| Squirtle | Water |

table2:

| pokemon | type |
| --- | --- |
| Snorlax | Normal |

If we combine these two with UNION:

SELECT \*

FROM table1

UNION

SELECT \*

FROM table2;

The result would be:

| pokemon | type |
| --- | --- |
| Bulbasaur | Grass |
| Charmander | Fire |
| Squirtle | Water |
| Snorlax | Normal |

SQL has strict rules for appending data:

* Tables must have the same number of columns.
* The columns must have the same data types in the same order as the first table.

# With

Often times, we want to combine two tables, but one of the tables is the result of another calculation.

Let’s return to our magazine order example. Our marketing department might want to know a bit more about our customers. For instance, they might want to know how many magazines each customer subscribes to. We can easily calculate this using our orders table:

SELECT customer\_id,

COUNT(subscription\_id) AS 'subscriptions'

FROM orders

GROUP BY customer\_id;

This query is good, but a customer\_id helpful. customer’s name is needed for marketing.

We want to be able to join the results of this query with our customers table, which will tell us the name of each customer. We can do this by using a WITH clause.

WITH previous\_results AS (

SELECT ...

...

...

...

)

SELECT \*

FROM previous\_results

JOIN customers

ON \_\_\_\_\_ = \_\_\_\_\_;

* The WITH statement allows us to perform a separate query (such as aggregating customer’s subscriptions)
* previous\_results is the alias that we will use to reference any columns from the query inside of the WITH clause
* We can then go on to do whatever we want with this temporary table (such as join the temporary table with another table)

Essentially, we are putting a whole first query inside the parentheses () and giving it a name. After that, we can use this name as if it’s a table and write a new query **using** the first query.

WITH previous\_query AS (

SELECT customer\_id,

COUNT(subscription\_id) AS 'subscriptions'

FROM orders

GROUP BY customer\_id

)

SELECT customers.customer\_name,

previous\_query.subscriptions

FROM previous\_query

JOIN customers

ON previous\_query.customer\_id = customers.customer\_id;

