**Beginning your SQL journey**

Now that you're familiar with the interface, let's get straight into it.

SQL, which stands for *Structured Query Language*, is a language for interacting with data stored in something called a *relational database*.

You can think of a relational database as a collection of tables. A table is just a set of rows and columns, like a spreadsheet, which represents exactly one type of entity. For example, a table might represent employees in a company or purchases made, but not both.

Each row, or *record*, of a table contains information about a single entity. For example, in a table representing employees, each row represents a single person. Each column, or *field*, of a table contains a single attribute for all rows in the table. For example, in a table representing employees, we might have a column containing first and last names for all employees.

The table of employees might look something like this:

| **id** | **name** | **age** | **nationality** |
| --- | --- | --- | --- |
| 1 | Jessica | 22 | Ireland |
| 2 | Gabriel | 48 | France |
| 3 | Laura | 36 | USA |

How many fields does the employees table above contain?

**SELECTing single columns**

While SQL can be used to create and modify databases, the focus of this course will be *querying* databases. A *query* is a request for data from a database table (or combination of tables). Querying is an essential skill for a data scientist, since the data you need for your analyses will often live in databases.

In SQL, you can select data from a table using a SELECT statement. For example, the following query selects the name column from the people table:

SELECT name

FROM people;

In this query, SELECT and FROM are called keywords. In SQL, keywords are not case-sensitive, which means you can write the same query as:

select name

from people;

That said, it's good practice to make SQL keywords uppercase to distinguish them from other parts of your query, like column and table names.

It's also good practice (but not necessary for the exercises in this course) to include a semicolon at the end of your query. This tells SQL where the end of your query is!

# SELECTing multiple columns

Well done! Now you know how to select single columns.

In the real world, you will often want to select multiple columns. Luckily, SQL makes this really easy. To select multiple columns from a table, simply separate the column names with commas!

For example, this query selects two columns, name and birthdate, from the people table:

SELECT name, birthdate

FROM people;

Sometimes, you may want to select all columns from a table. Typing out every column name would be a pain, so there's a handy shortcut:

SELECT \*

FROM people;

If you only want to return a certain number of results, you can use the LIMIT keyword to limit the number of rows returned:

SELECT \*

FROM people

LIMIT 10;

# SELECT DISTINCT

Often your results will include many duplicate values. If you want to select all the unique values from a column, you can use the DISTINCT keyword.

This might be useful if, for example, you're interested in knowing which languages are represented in the films table:

SELECT DISTINCT language

FROM films;

# Learning to COUNT

What if you want to count the number of employees in your employees table? The COUNT() function lets you do this by returning the number of rows in one or more columns.

For example, this code gives the number of rows in the people table:

SELECT COUNT(\*)

FROM people;

# Practice with COUNT

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

For example, to count the number of birth dates present in the people table:

SELECT COUNT(birthdate)

FROM people;

It's also common to combine COUNT() with DISTINCT to count the number of distinct values in a column.

For example, this query counts the number of distinct birth dates contained in the people table:

SELECT COUNT(DISTINCT birthdate)

FROM people;

# Filtering results

Congrats on finishing the first chapter! You now know how to select columns and perform basic counts. This chapter will focus on filtering your results.

In SQL, the WHERE keyword allows you to filter based on both text and numeric values in a table. There are a few different comparison operators you can use:

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

For example, you can filter text records such as title. The following code returns all films with the title 'Metropolis':

SELECT title

FROM films

WHERE title = 'Metropolis';

Notice that the WHERE clause always comes after the FROM statement!

**Note that in this course we will use <> and not != for the not equal operator, as per the SQL standard.**

# Simple filtering of text

Remember, the WHERE clause can also be used to filter text results, such as names or countries.

For example, this query gets the titles of all films which were filmed in China:

SELECT title

FROM films

WHERE country = 'China';

Now it's your turn to practice using WHERE with text values!

**Important: in PostgreSQL (the version of SQL we're using), you must use single quotes with WHERE.**

# WHERE AND

Often, you'll want to select data based on multiple conditions. You can build up your WHERE queries by combining multiple conditions with the AND keyword.

For example,

SELECT title

FROM films

WHERE release\_year > 1994

AND release\_year < 2000;

gives you the titles of films released between 1994 and 2000.

Note that you need to specify the column name separately for every AND condition, so the following would be invalid:

SELECT title

FROM films

WHERE release\_year > 1994 AND < 2000;

You can add as many AND conditions as you need!

# WHERE AND OR

What if you want to select rows based on multiple conditions where some but not all of the conditions need to be met? For this, SQL has the OR operator.

For example, the following returns all films released in either 1994 or 2000:

SELECT title

FROM films

WHERE release\_year = 1994

OR release\_year = 2000;

Note that you need to specify the column for every OR condition, so the following is invalid:

SELECT title

FROM films

WHERE release\_year = 1994 OR 2000;

When combining AND and OR, be sure to enclose the individual clauses in parentheses, like so:

SELECT title

FROM films

WHERE (release\_year = 1994 OR release\_year = 1995)

AND (certification = 'PG' OR certification = 'R');

Otherwise, due to SQL's precedence rules, you may not get the results you're expecting!

# WHERE AND OR (2)

You now know how to select rows that meet **some** but not **all** conditions by combining AND and OR.

For example, the following query selects all films that were released in 1994 or 1995 which had a rating of PG or R.

SELECT title

FROM films

WHERE (release\_year = 1994 OR release\_year = 1995)

AND (certification = 'PG' OR certification = 'R');

Now you'll write a query to get the title and release year of films released in the 90s which were in French or Spanish and which took in more than $2M 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!

# BETWEEN

As you've learned, you can use the following query to get titles of all films released in and between 1994 and 2000:

SELECT title

FROM films

WHERE release\_year >= 1994

AND release\_year <= 2000;

Checking for ranges like this is very common, so in SQL the BETWEEN keyword provides a useful shorthand for filtering values within a specified range. This query is equivalent to the one above:

SELECT title

FROM films

WHERE release\_year

BETWEEN 1994 AND 2000;

It's important to remember that BETWEEN is inclusive, meaning the beginning and end values are included in the results!

# BETWEEN (2)

Similar to the WHERE clause, the BETWEEN clause can be used with multiple AND and OR operators, so you can build up your queries and make them even more powerful!

For example, suppose we have a table called kids. We can get the names of all kids between the ages of 2 and 12 from the United States:

SELECT name

FROM kids

WHERE age BETWEEN 2 AND 12

AND nationality = 'USA';

Take a go at using BETWEEN with AND on the films data 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!

# WHERE IN

As you've seen, WHERE is very useful for filtering results. However, if you want to filter based on many conditions, WHERE can get unwieldy. For example:

SELECT name

FROM kids

WHERE age = 2

OR age = 4

OR age = 6

OR age = 8

OR age = 10;

Enter the IN operator! The IN operator allows you to specify multiple values in a WHERE clause, making it easier and quicker to specify multiple OR conditions! Neat, right?

So, the above example would become simply:

SELECT name

FROM kids

WHERE age IN (2, 4, 6, 8, 10);

# Introduction to NULL and IS NULL

In SQL, NULL represents a missing or unknown value. You can check for NULL values using the expression IS NULL. For example, to count the number of missing birth dates in the people table:

SELECT COUNT(\*)

FROM people

WHERE birthdate IS NULL;

As you can see, IS NULL is useful when combined with WHERE to figure out what data you're missing.

Sometimes, you'll want to filter out missing values so you only get results which are not NULL. To do this, you can use the IS NOT NULL operator.

For example, this query gives the names of all people whose birth dates are not missing in the people table.

SELECT name

FROM people

WHERE birthdate IS NOT NULL;

# LIKE and NOT LIKE

As you've seen, the WHERE clause can be used to filter text data. However, so far you've only been able to filter by specifying the exact text you're interested in. In the real world, often you'll want to search for a pattern rather than a specific text string.

In SQL, the LIKE operator can be used in a WHERE clause to search for a pattern in a column. To accomplish this, you use something called a wildcard as a placeholder for some other values. There are two wildcards you can use with LIKE:

The % wildcard will match zero, one, or many characters in text. For example, the following query matches companies like 'Data', 'DataC' 'DataCamp', 'DataMind', and so on:

SELECT name

FROM companies

WHERE name LIKE 'Data%';

The \_ wildcard will match a single character. For example, the following query matches companies like 'DataCamp', 'DataComp', and so on:

SELECT name

FROM companies

WHERE name LIKE 'DataC\_mp';

You can also use the NOT LIKE operator to find records that *don't* match the pattern you specify.

# Aggregate functions

Often, you will want to perform some calculation on the data in a database. SQL provides a few functions, called aggregate functions, to help you out with this.

For example,

SELECT AVG(budget)

FROM films;

gives you the average value from the budget column of the films table. Similarly, the MAX() function returns the highest budget:

SELECT MAX(budget)

FROM films;

The SUM() function returns the result of adding up the numeric values in a column:

SELECT SUM(budget)

FROM films;

You can probably guess what the MIN() function does!

# Combining aggregate functions with WHERE

Aggregate functions can be combined with the WHERE clause to gain further insights from your data.

For example, to get the total budget of movies made in the year 2010 or later:

SELECT SUM(budget)

FROM films

WHERE release\_year >= 2010;

# A note on arithmetic

In addition to using aggregate functions, you can perform basic arithmetic with symbols like +, -, \*, and /.

So, for example, this gives a result of 12:

SELECT (4 \* 3);

However, the following gives a result of 1:

SELECT (4 / 3);

What's going on here?

SQL assumes that if you divide an integer by an integer, you want to get an integer back. So be careful when dividing!

If you want more precision when dividing, you can add decimal places to your numbers. For example,

SELECT (4.0 / 3.0) AS result;

gives you the result you would expect: 1.333.

# It's AS simple AS aliasing

You may have noticed in the first exercise of this chapter that the column name of your result was just the name of the function you used. For example,

SELECT MAX(budget)

FROM films;

gives you a result with one column, named max. But what if you use two functions like this?

SELECT MAX(budget), MAX(duration)

FROM films;

Well, then you'd have two columns named max, which isn't very useful!

To avoid situations like this, SQL allows you to do something called aliasing. Aliasing simply means you assign a temporary name to something. To alias, you use the AS keyword, which you've already seen earlier in this course

For example, in the above example we could use aliases to make the result clearer:

SELECT MAX(budget) AS max\_budget,

MAX(duration) AS max\_duration

FROM films;

Aliases are helpful for making results more readable!

# Even more aliasing

Let's practice your newfound aliasing skills some more before moving on!

**Recall:** SQL assumes that if you divide an integer by an integer, you want to get an integer back.

This means that the following will erroneously result in 400.0:

SELECT 45 / 10 \* 100.0;

This is because 45 / 10 evaluates to an integer (4), and not a decimal number like we would expect.

So when you're dividing make sure at least one of your numbers has a decimal place:

SELECT 45 \* 100.0 / 10;

The above now gives the correct answer of 450.0 since the numerator (45 \* 100.0) of the division is now a decimal!

# ORDER BY

Congratulations on making it this far! You now know how to select and filter your results.

In this chapter you'll learn how to sort and group your results to gain further insight. Let's go!

In SQL, the ORDER BY keyword is used to sort results in ascending or descending order according to the values of one or more columns.

By default ORDER BY will sort in ascending order. If you want to sort the results in descending order, you can use the DESC keyword. For example,

SELECT title

FROM films

ORDER BY release\_year DESC;

gives you the titles of films sorted by release year, from newest to oldest.

# Sorting single columns (DESC)

To order results in descending order, you can put the keyword DESC after your ORDER BY. For example, to get all the names in the people table, in reverse alphabetical order:

SELECT name

FROM people

ORDER BY name DESC;

# Sorting multiple columns

ORDER BY can also be used to sort on multiple columns. It will sort by the first column specified, then sort by the next, then the next, and so on. For example,

SELECT birthdate, name

FROM people

ORDER BY birthdate, name;

sorts on birth dates first (oldest to newest) and then sorts on the names in alphabetical order. **The order of columns is important!**

Try using ORDER BY to sort multiple columns! Remember, to specify multiple columns you separate the column names with a comma.

# GROUP BY

Now you know how to sort results! Often you'll need to aggregate results. For example, you might want to count the number of male and female employees in your company. Here, what you want is to group all the males together and count them, and group all the females together and count them. In SQL, GROUP BY allows you to group a result by one or more columns, like so:

SELECT sex, count(\*)

FROM employees

GROUP BY sex;

This might give, for example:

| **sex** | **count** |
| --- | --- |
| male | 15 |
| female | 19 |

Commonly, GROUP BY is used with aggregate functions like COUNT() or MAX(). Note that GROUP BY always goes after the FROM clause!

# GROUP BY practice

As you've just seen, combining aggregate functions with GROUP BY can yield some powerful results!

A word of warning: SQL will return an error if you try to SELECT a field that is not in your GROUP BY clause without using it to calculate some kind of value about the entire group.

Note that you can combine GROUP BY with ORDER BY to group your results, calculate something about them, and then order your results. For example,

SELECT sex, count(\*)

FROM employees

GROUP BY sex

ORDER BY count DESC;

might return something like

| **sex** | **count** |
| --- | --- |
| female | 19 |
| male | 15 |

because there are more females at our company than males. Note also that ORDER BY always goes after GROUP BY.

# GROUP BY practice (2)

Now practice your new skills by combining GROUP BY and ORDER BY with some more aggregate functions!

Make sure to always put the ORDER BY clause at the end of your query. You can't sort values that you haven't calculated yet!

# HAVING a great time

In SQL, aggregate functions can't be used in WHERE clauses. For example, the following query is invalid:

SELECT release\_year

FROM films

GROUP BY release\_year

WHERE COUNT(title) > 10;

This means that if you want to filter based on the result of an aggregate function, you need another way! That's where the HAVING clause comes in. For example,

SELECT release\_year

FROM films

GROUP BY release\_year

HAVING COUNT(title) > 10;

shows only those years in which more than 10 films were released.

# All together now

Time to practice using ORDER BY, GROUP BY and HAVING together.

Now you're going to write a query that returns the average budget and average gross earnings for films in each year after 1990, if the average budget is greater than $60 million.

This is going to be a big query, but you can handle it!

# A taste of things to come

Congrats on making it to the end of the course! By now you should have a good understanding of the basics of SQL.

There's one more concept we're going to introduce. You may have noticed that all your results so far have been from just one table, e.g. films or people.

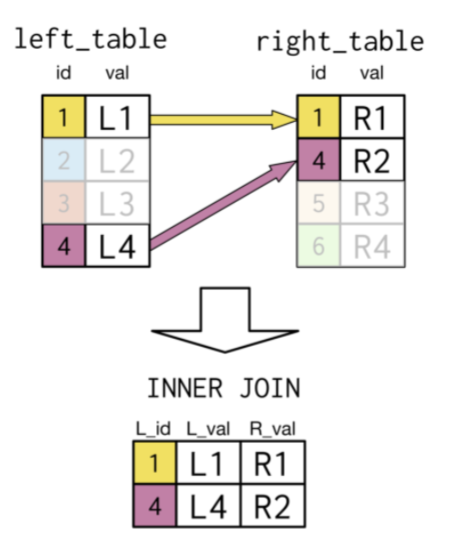
In the real world however, you will often want to query multiple tables. For example, what if you want to see the IMDB score for a particular movie?

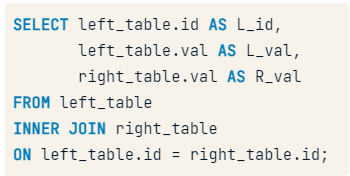
In this case, you'd want to get the ID of the movie from the films table and then use it to get IMDB information from the reviews table. In SQL, this concept is known as a **join**, and a basic join is shown in the editor to the right.

The query in the editor gets the IMDB score for the film To Kill a Mockingbird! Cool right?

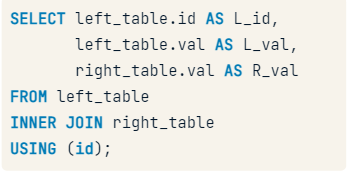
As you can see, joins are incredibly useful and important to understand for anyone using SQL.

We have a whole follow-up course dedicated to them called [**Joining Data in PostgreSQL**](https://www.datacamp.com/courses/joining-data-in-postgresql) for you to hone your database skills further!





İf the keywords are the same you can use USING





# Inner join

Although this courses focuses on PostgreSQL, you'll find that these joins and the material here applies to different forms of SQL as well.

Throughout this course, you'll be working with the countries database containing information about the most populous world cities as well as country-level economic data, population data, and geographic data. This countries database also contains information on languages spoken in each country.

You can see the different tables in this database by clicking on the corresponding tabs. Click through them to get a sense for the types of data that each table contains before you continue with the course! Take note of the fields that appear to be shared across the tables.

Recall from the video the basic syntax for an INNER JOIN, here including all columns in **both** tables:

SELECT \*

FROM left\_table

INNER JOIN right\_table

ON left\_table.id = right\_table.id;

You'll start off with a SELECT statement and then build up to an INNER JOIN with the cities and countries tables.

* Modify the SELECT statement to keep only the name of the city, the name of the country, and the name of the region the country resides in.
* Alias the name of the city AS city and the name of the country AS country.

# Inner join (2)

Instead of writing the full table name, you can use table aliasing as a shortcut. For tables you also use AS to add the alias immediately after the table name with a space. Check out the aliasing of cities and countries below.

SELECT c1.name AS city, c2.name AS country

FROM cities AS c1

INNER JOIN countries AS c2

ON c1.country\_code = c2.code;

Notice that to select a field in your query that appears in multiple tables, you'll need to identify which table/table alias you're referring to by using a . in your SELECT statement.

You'll now explore a way to get data from both the countries and economies tables to examine the inflation rate for both 2010 and 2015.

Sometimes it's easier to write SQL code out of order: you write the SELECT statement after you've done the JOIN.

# Inner join (3)

The ability to combine multiple joins in a single query is a powerful feature of SQL, e.g:

SELECT \*

FROM left\_table

INNER JOIN right\_table

ON left\_table.id = right\_table.id

INNER JOIN another\_table

ON left\_table.id = another\_table.id;

As you can see here it becomes tedious to continually write long table names in joins. This is when it becomes useful to alias each table using the first letter of its name (e.g. countries AS c)! It is standard practice to alias in this way and, if you choose to alias tables or are asked to specifically for an exercise in this course, you should follow this protocol.

Now, for each country, you want to get the country name, its region, the fertility rate, and the unemployment rate for both 2010 and 2015.

Note that results should work throughout this course with or without table aliasing unless specified differently.

# Inner join with using

When joining tables with a common field name, e.g.

SELECT \*

FROM countries

INNER JOIN economies

ON countries.code = economies.code

You can use USING as a shortcut:

SELECT \*

FROM countries

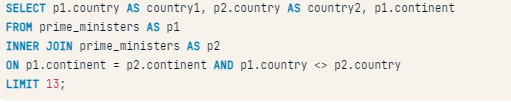
INNER JOIN economies

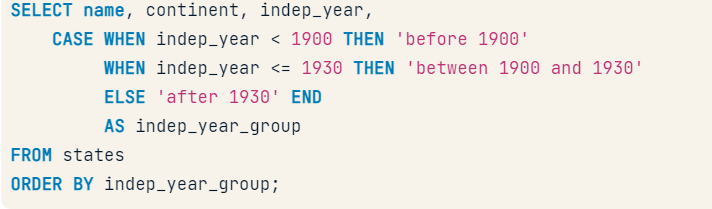
USING(code)

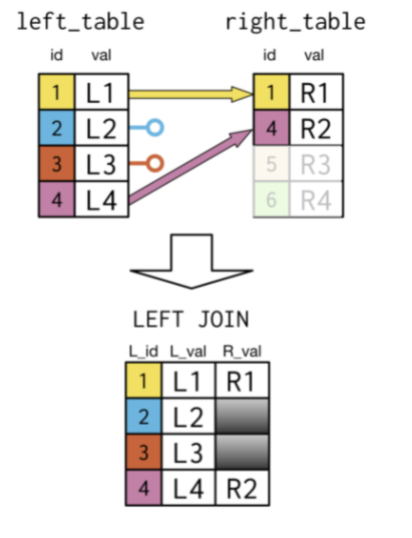
# Case when and then

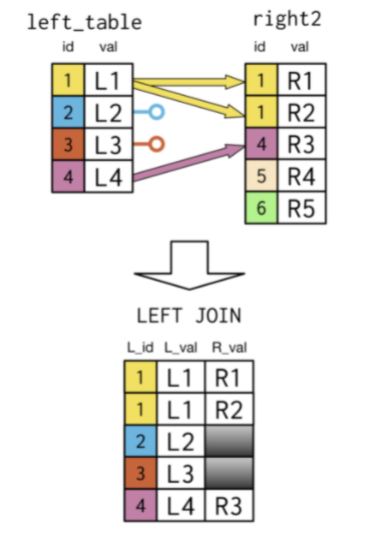
Often it's useful to look at a numerical field not as raw data, but instead as being in different categories or groups.

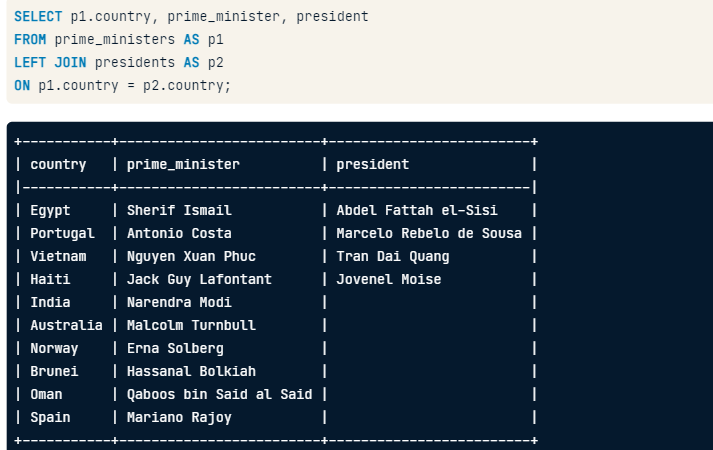
You can use CASE with WHEN, THEN, ELSE, and END to define a new grouping field.

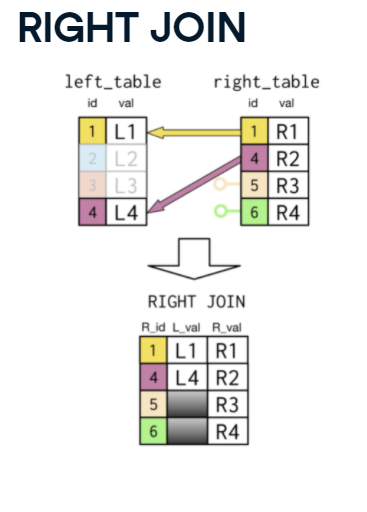












# Left Join

Now you'll explore the differences between performing an inner join and a left join using the cities and countries tables.

You'll begin by performing an inner join with the cities table on the left and the countries table on the right. Remember to alias the name of the city field as city and the name of the country field as country.

You will then change the query to a left join. Take note of how many records are in each query here!

# Left join (2)

Next, you'll try out another example comparing an inner join to its corresponding left join. Before you begin though, take note of how many records are in both the countries and languages tables below.

You will begin with an inner join on the countries table on the left with the languages table on the right. Then you'll change the code to a left join in the next bullet.

Note the use of multi-line comments here using /\* and \*/.

/\*

Select country name AS country, the country's local name,

the language name AS language, and

the percent of the language spoken in the country

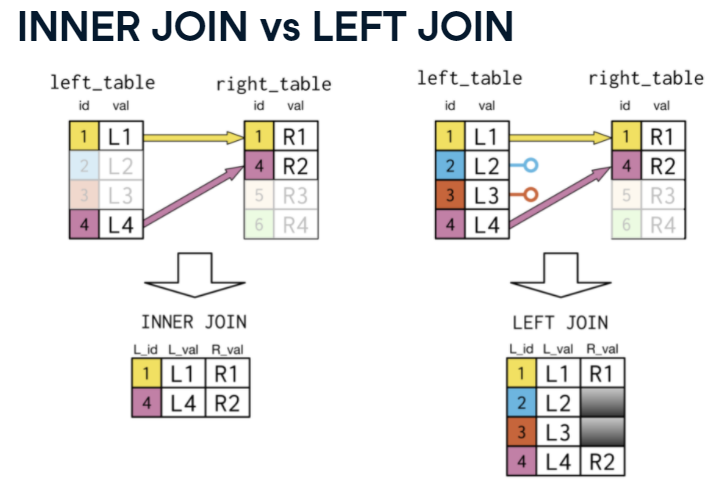
\*/

# Left join (3)

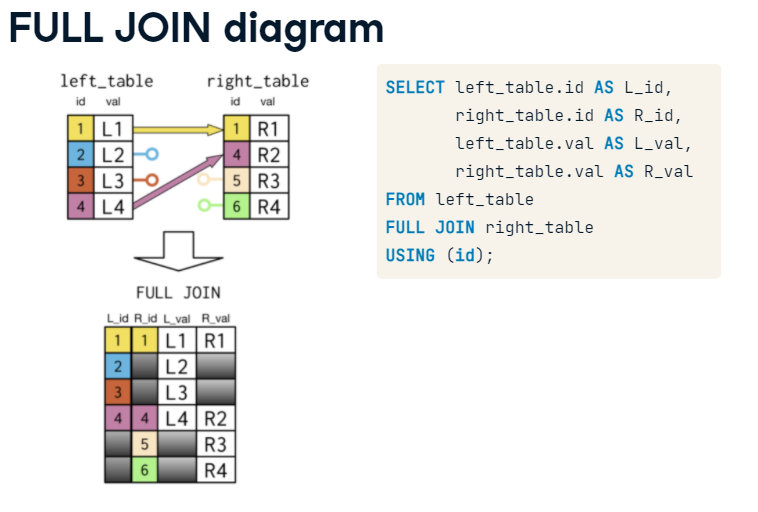
You'll now revisit the use of the AVG() function introduced in our introductory SQL course. You will use it in combination with left join to determine the average gross domestic product (GDP) per capita **by region** in 2010.

# Right join

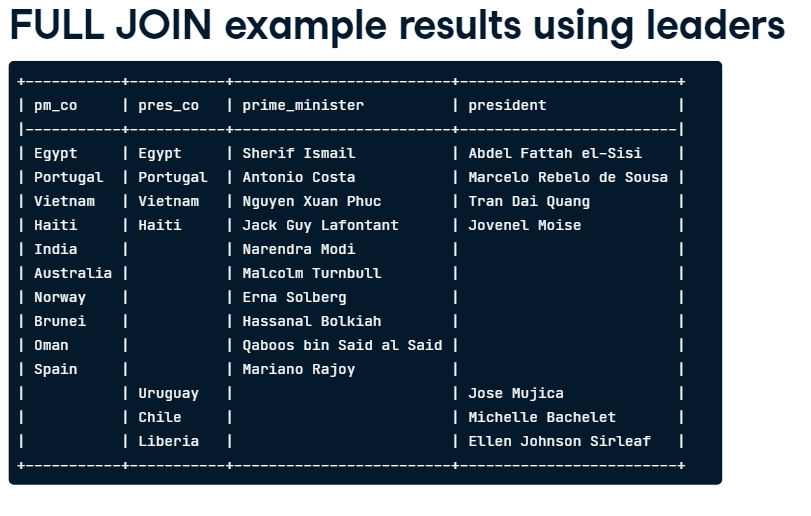
Right joins aren't as common as left joins. One reason why is that you can always write a right join as a left join.











# Full join

In this exercise, you'll examine how your results differ when using a full join versus using a left join versus using an inner join with the countries and currencies tables.

You will focus on the North American region and also where the name of the country is missing. Dig in to see what we mean!

Begin with a full join with countries on the left and currencies on the right. The fields of interest have been SELECTed for you throughout this exercise.

Then complete a similar left join and conclude with an inner join.

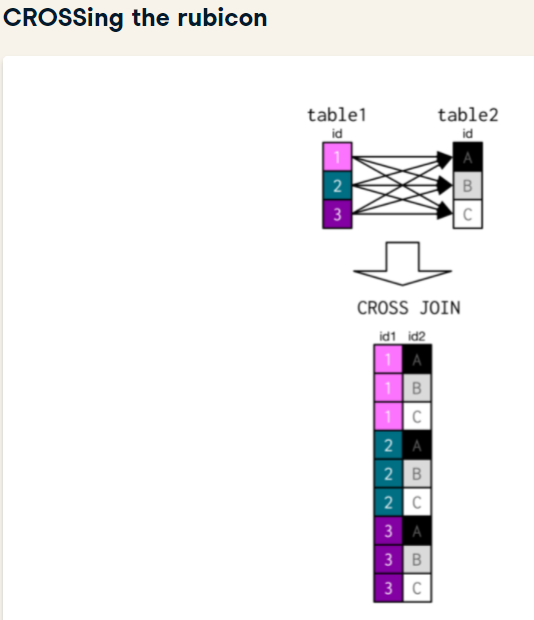
# Full join (2)

You'll now investigate a similar exercise to the last one, but this time focused on using a table with more records on the left than the right. You'll work with the languages and countries tables.

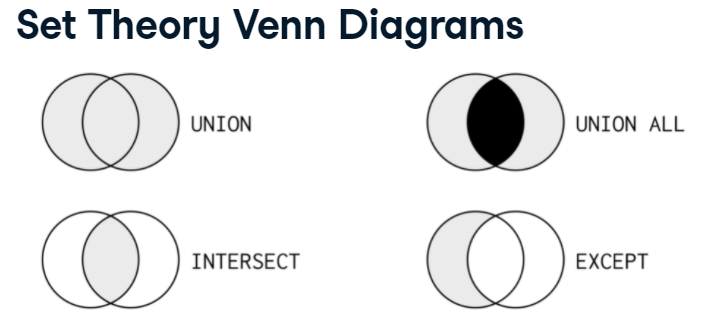
Begin with a full join with languages on the left and countries on the right. Appropriate fields have been selected for you again here.

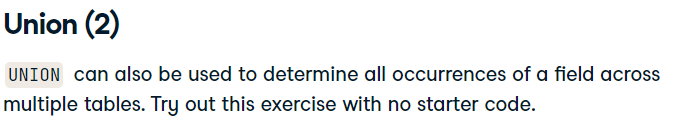
# Full join (3)

You'll now explore using two consecutive full joins on the three tables you worked with in the previous two exercises.





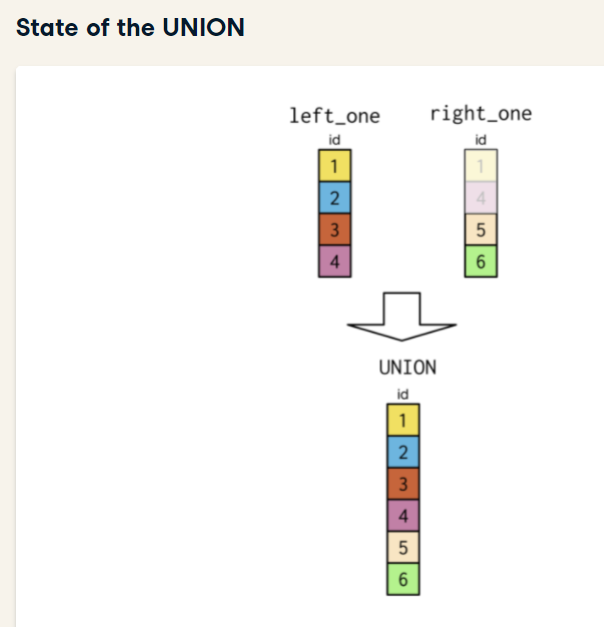


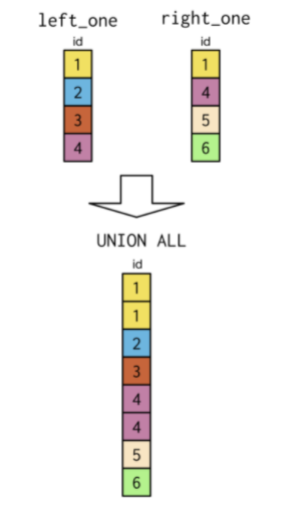


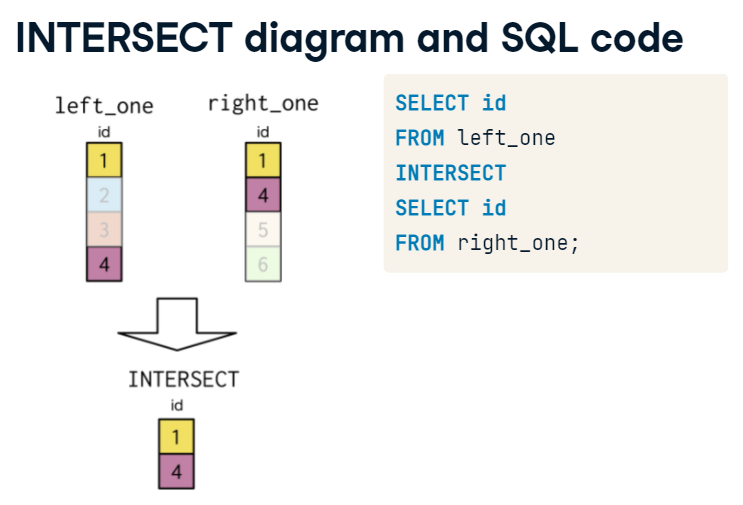
# Union all

As you saw, duplicates were removed from the previous two exercises by using UNION.

To include duplicates, you can use UNION ALL.



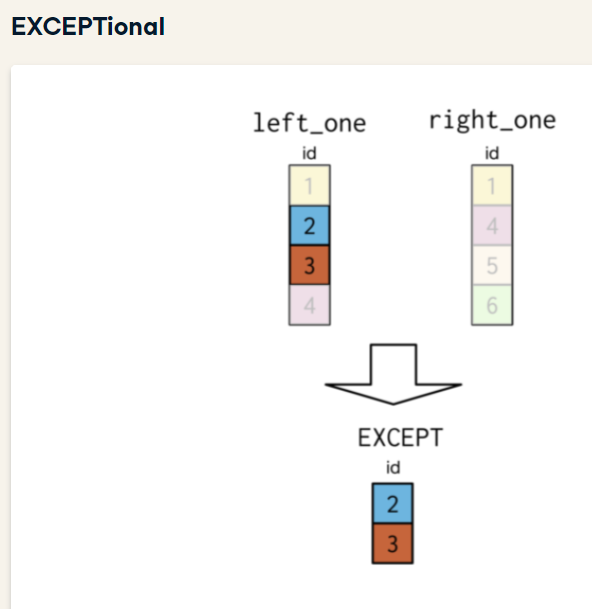




# Intersect

UNION ALL will extract all records from two tables, while INTERSECT will only return records that both tables have in common. In this exercise, you will create a similar query as before, however, this time you will look at the records in common for country code and year for the economies and populations tables.

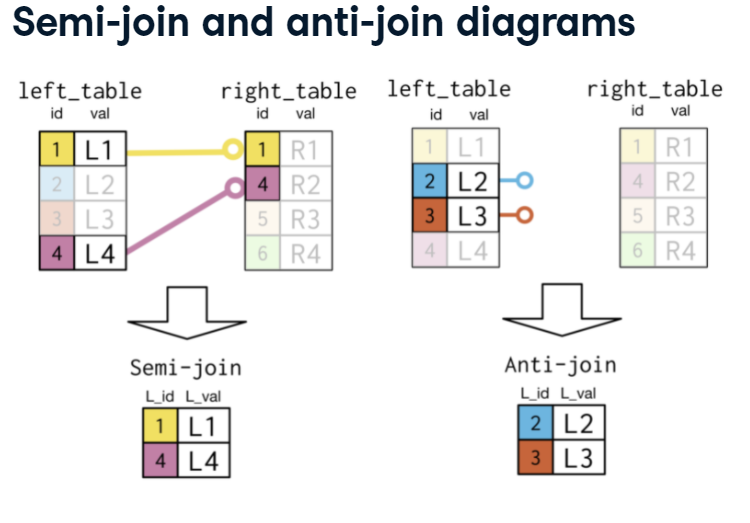
Note the number of records from the result of this query compared to the similar UNION ALL query result of 814 records.











# Relating semi-join to a tweaked inner join

Let's revisit the code from the previous exercise, which retrieves languages spoken in the Middle East.

SELECT DISTINCT name

FROM languages

WHERE code IN

(SELECT code

FROM countries

WHERE region = 'Middle East')

ORDER BY name;

Sometimes problems solved with semi-joins can also be solved using an inner join.

SELECT languages.name AS language

FROM languages

INNER JOIN countries

ON languages.code = countries.code

WHERE region = 'Middle East'

ORDER BY language;

This inner join isn't quite right. What is missing from this second code block to get it to match with the correct answer produced by the first block?

# Diagnosing problems using anti-join

Another powerful join in SQL is the anti-join. It is particularly useful in identifying which records are causing an incorrect number of records to appear in join queries.

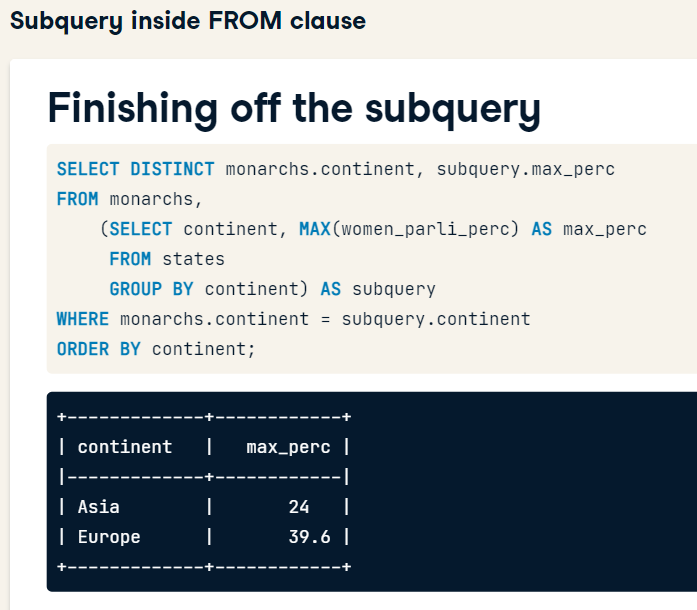
You will also see another example of a subquery here, as you saw in the first exercise on semi-joins. Your goal is to identify the currencies used in Oceanian countries!



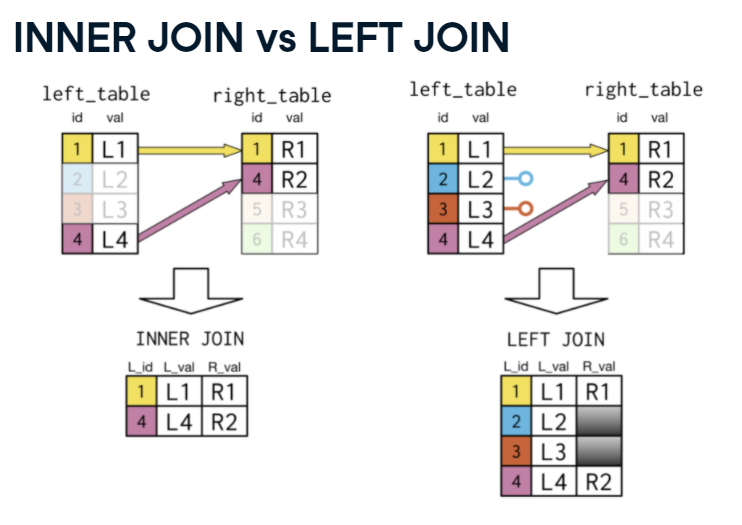




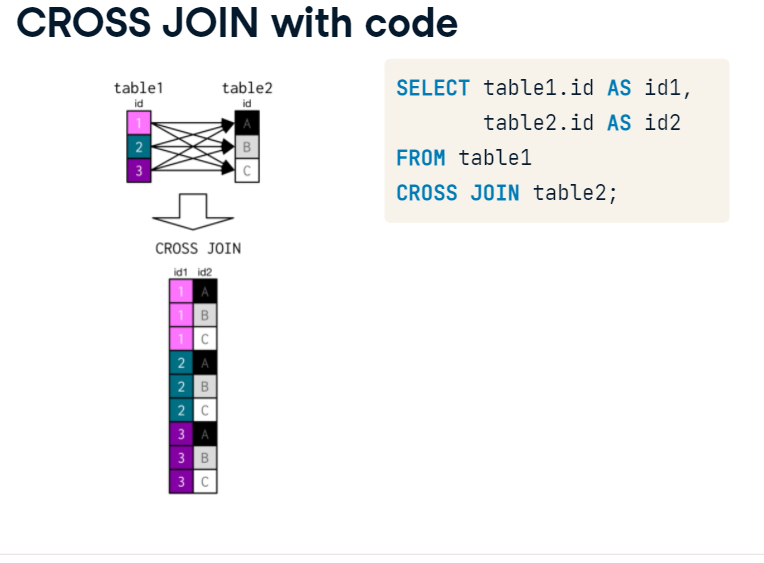


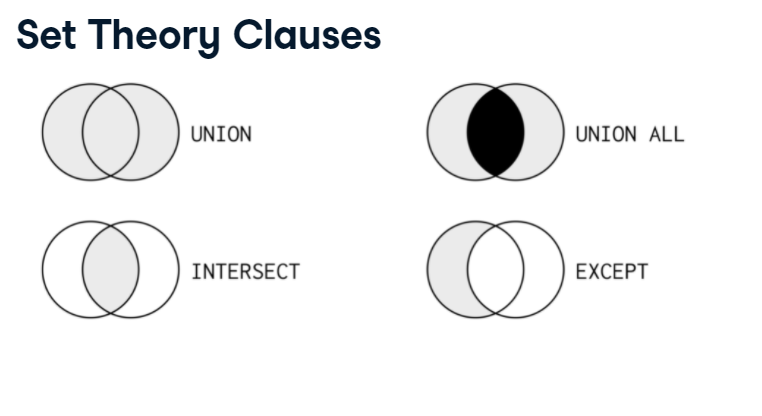


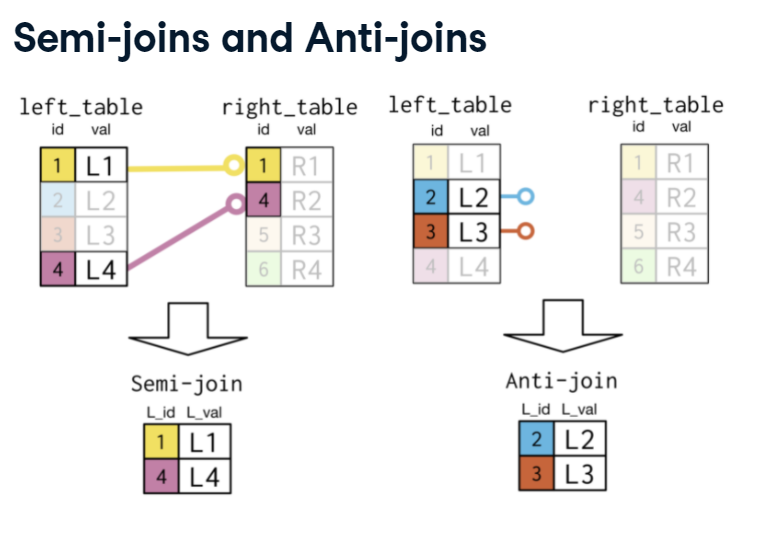


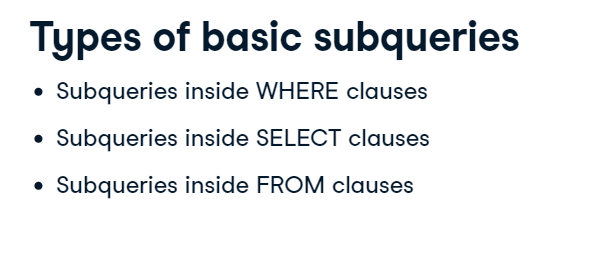




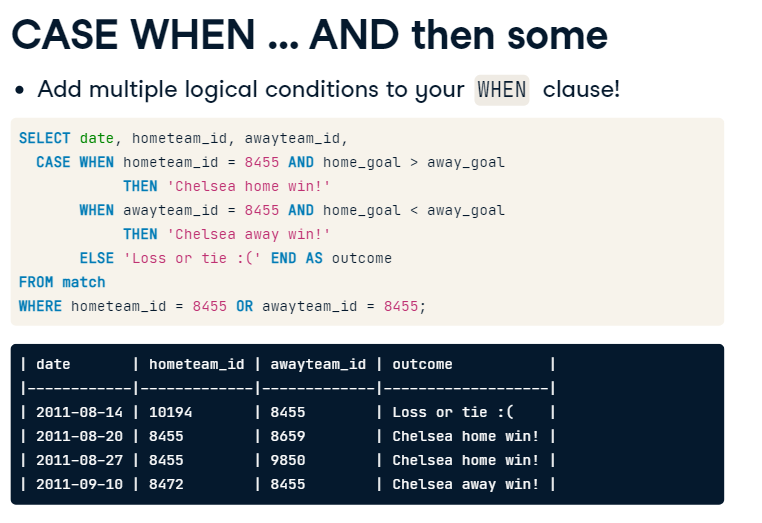




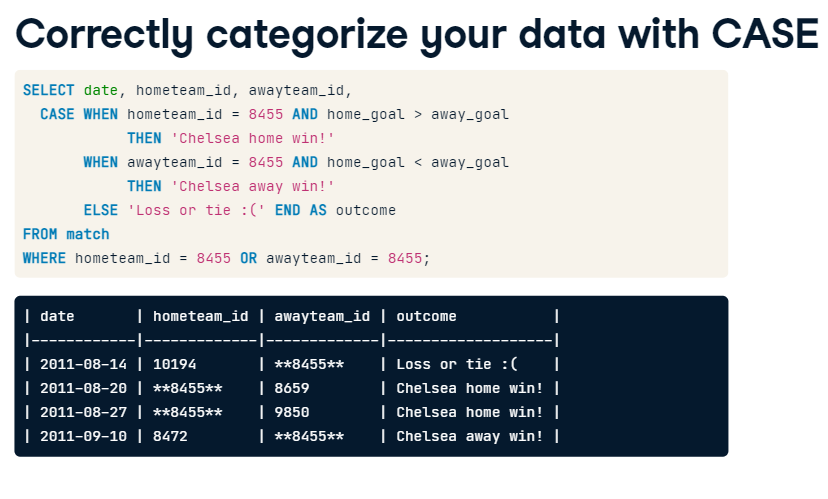


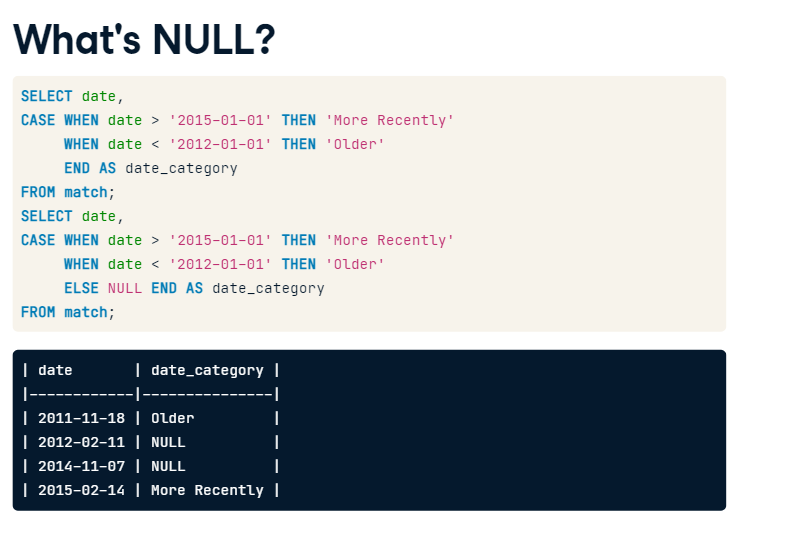






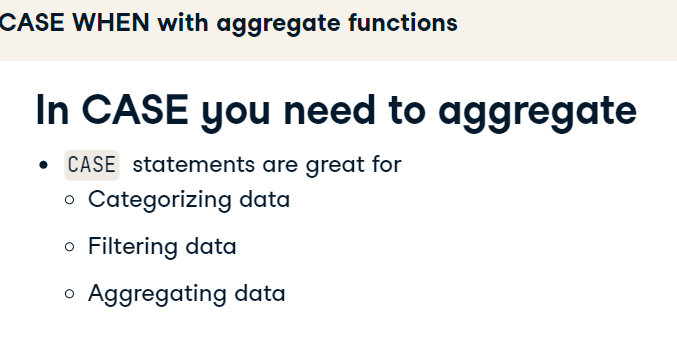






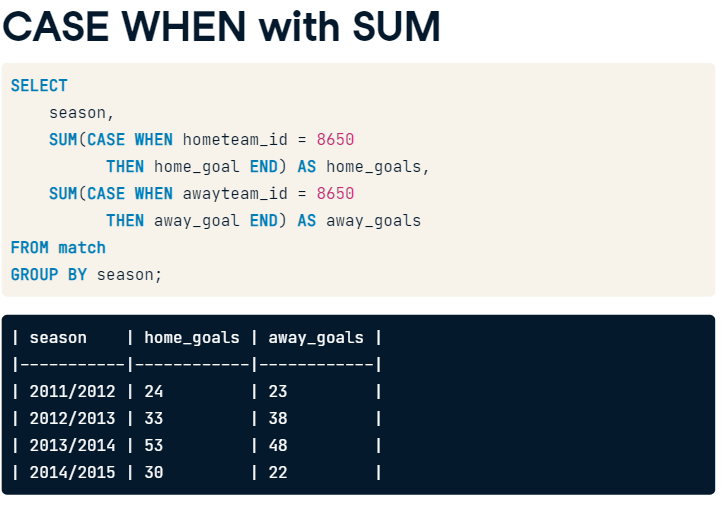




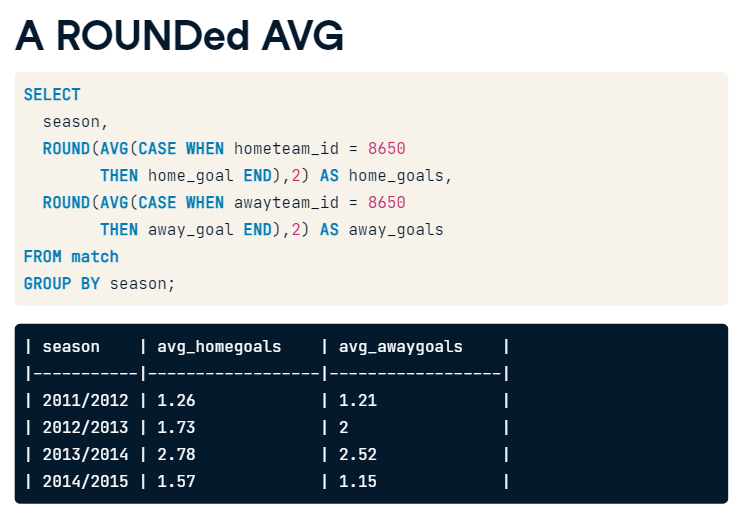


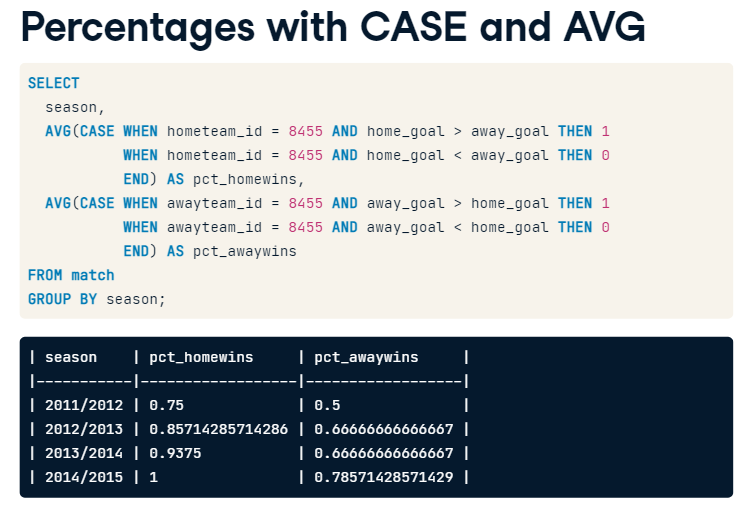


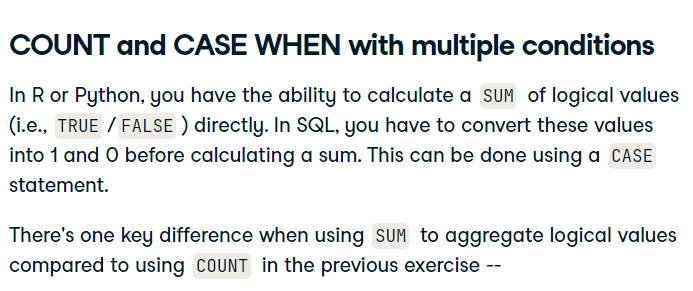


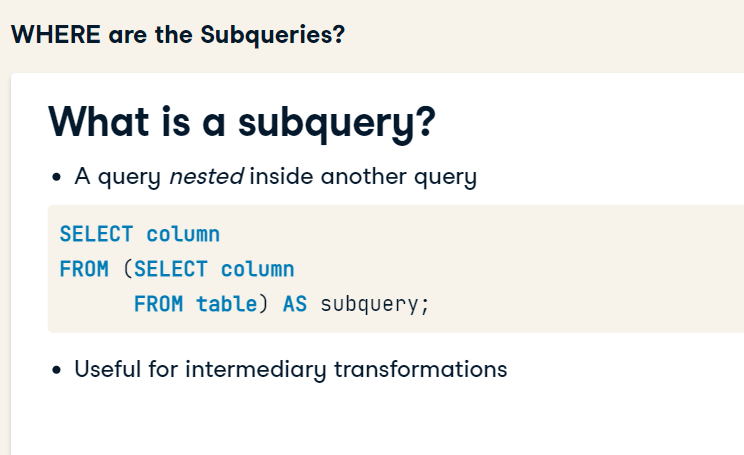


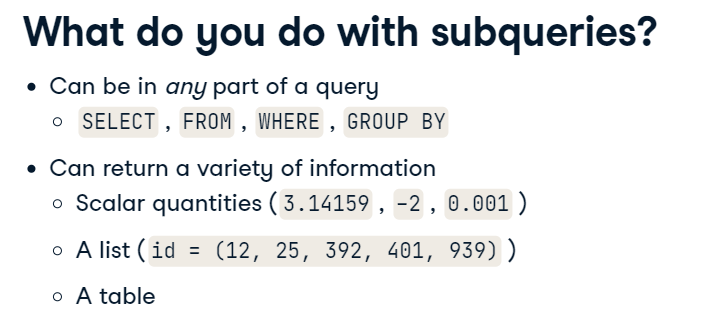


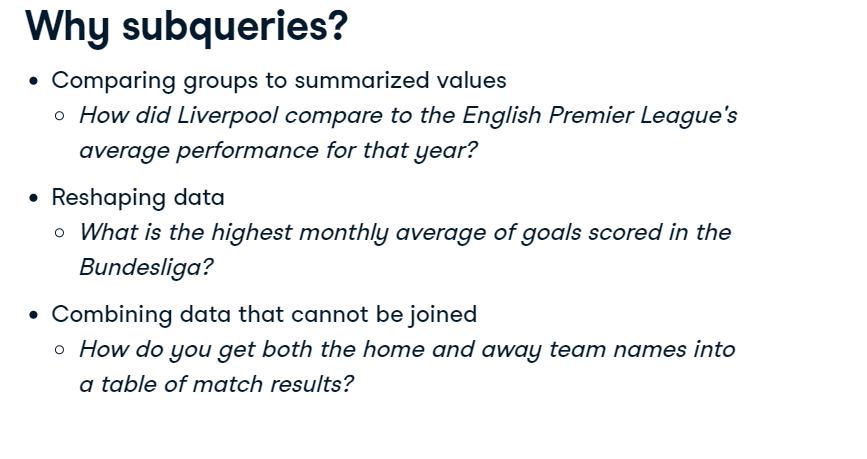


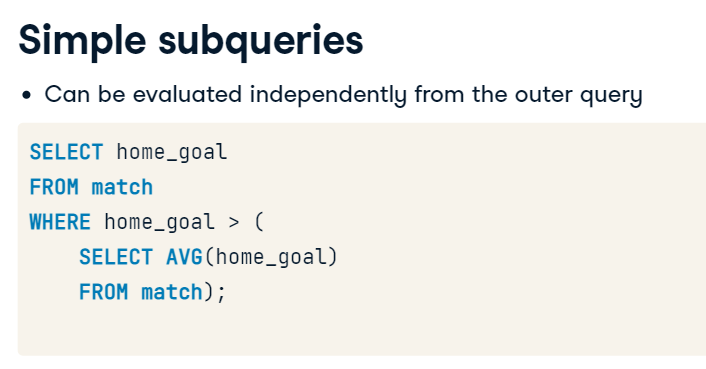


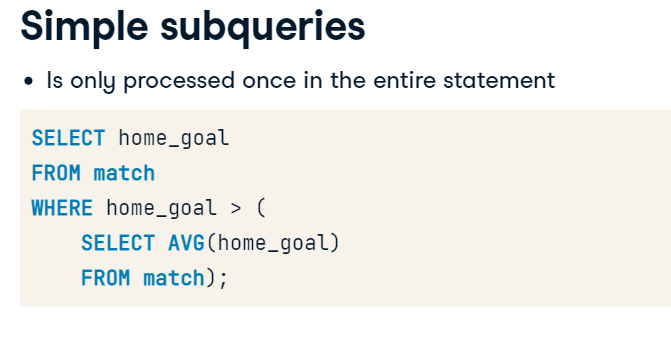






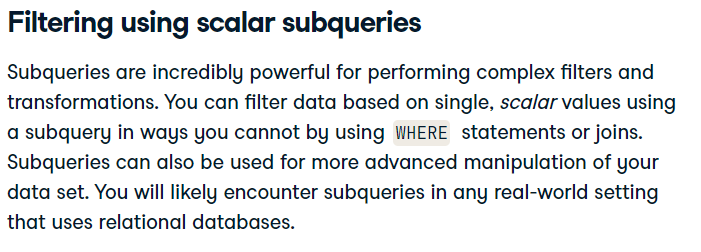


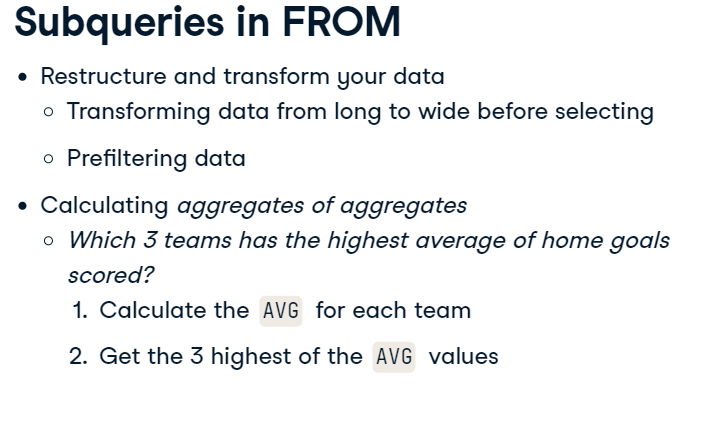


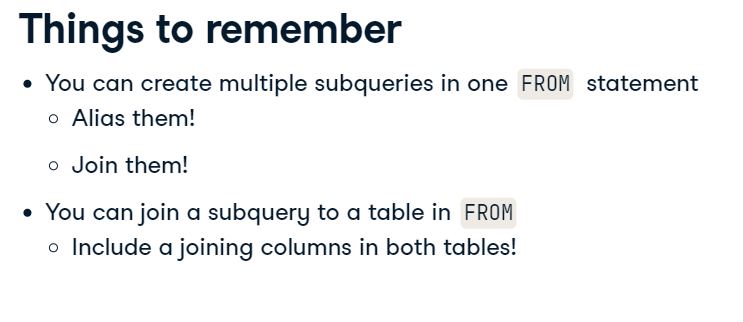


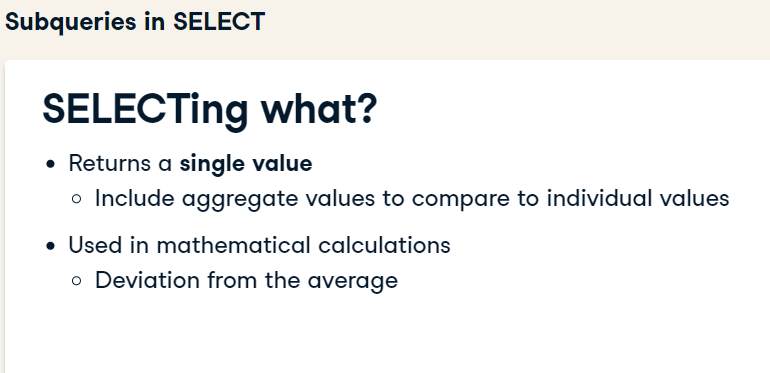


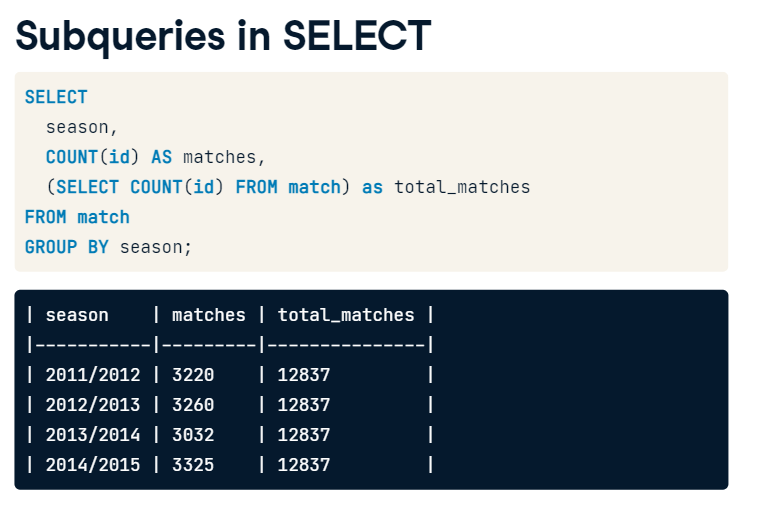






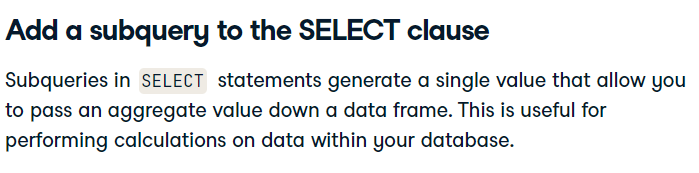


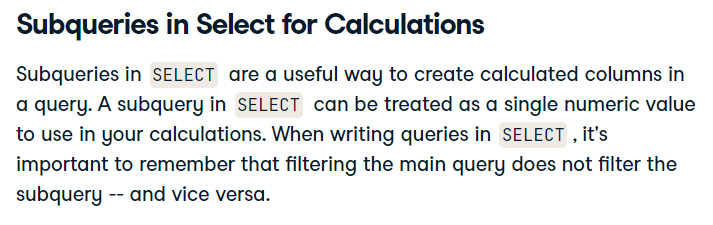


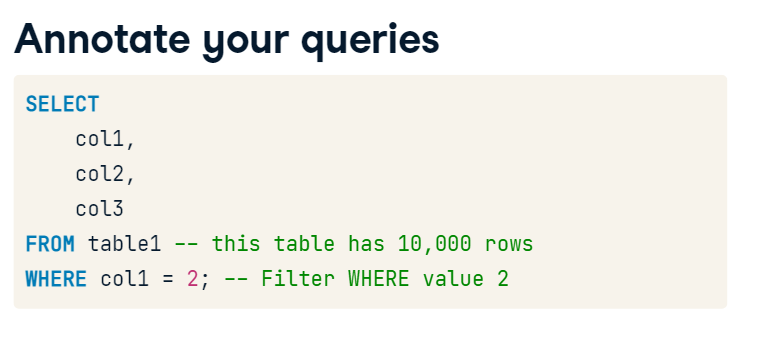
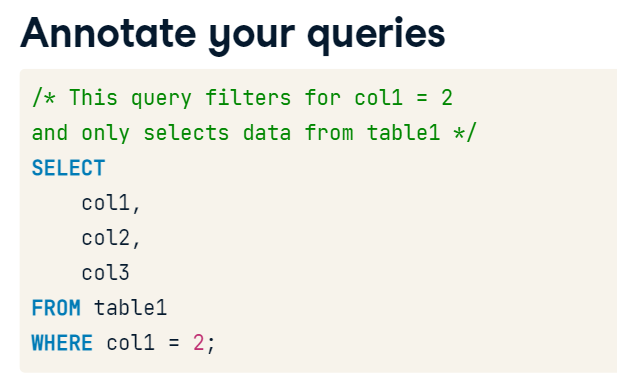






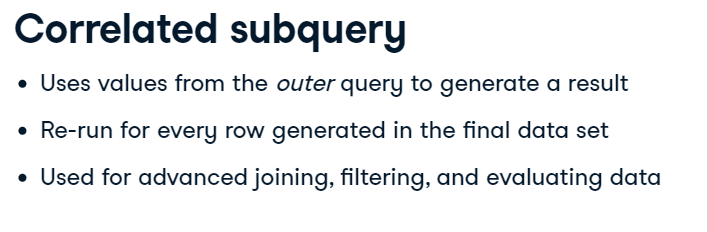


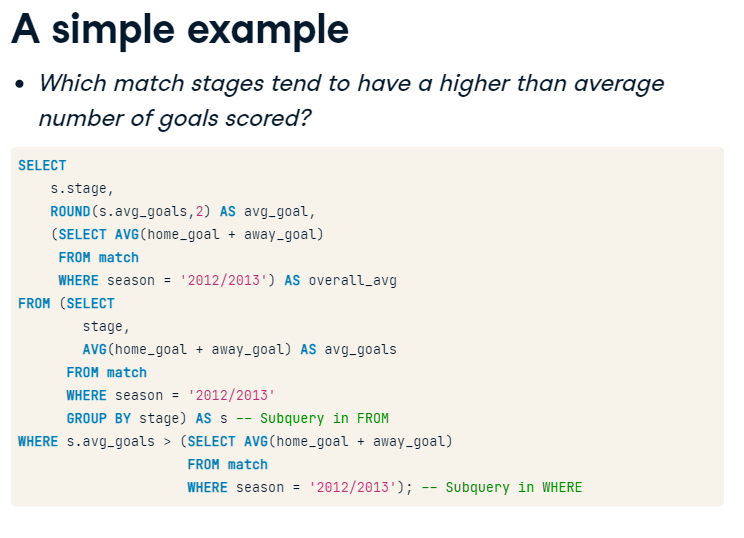


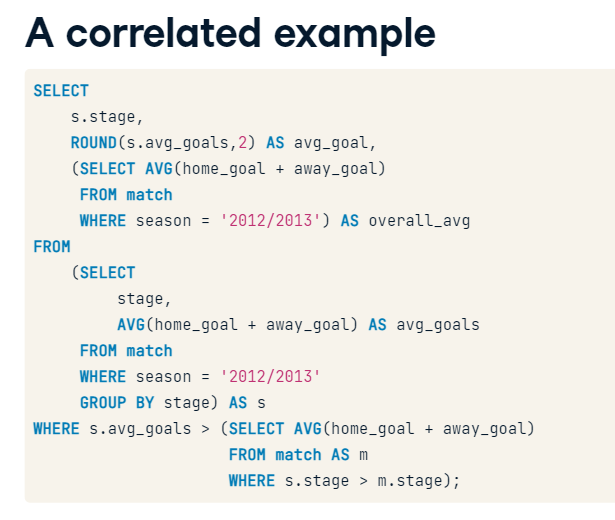


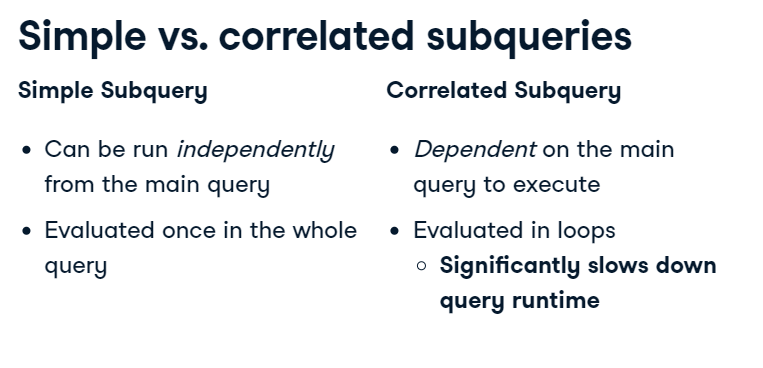


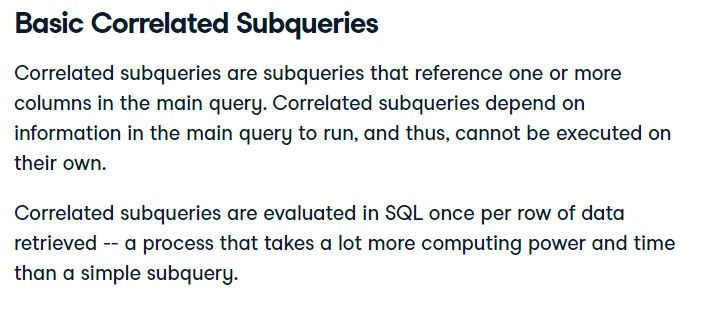


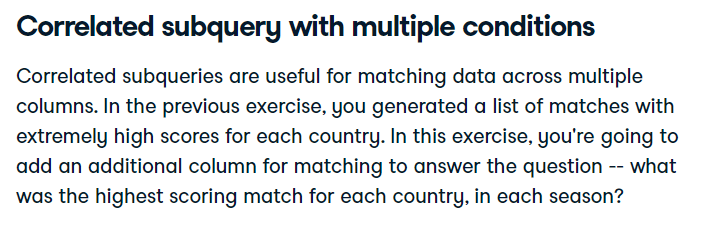


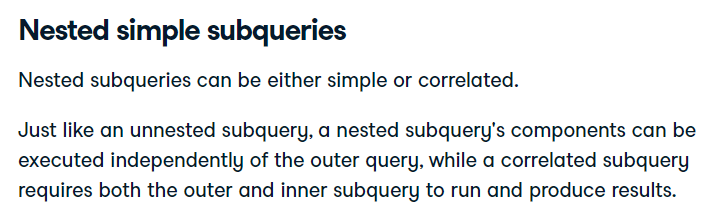


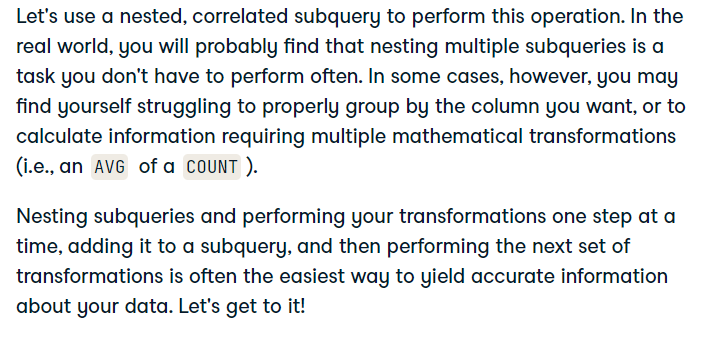






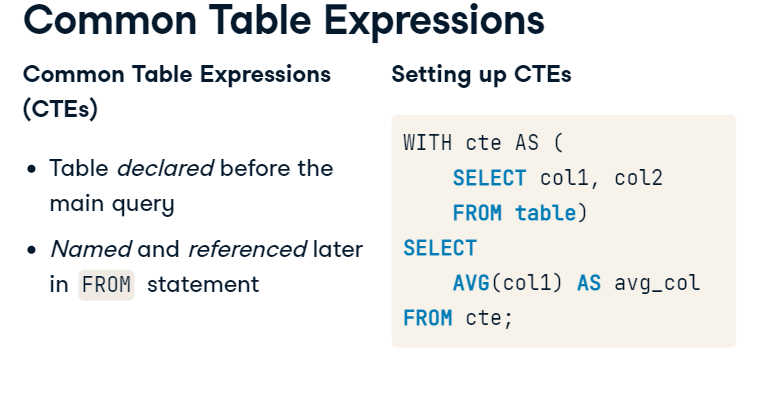


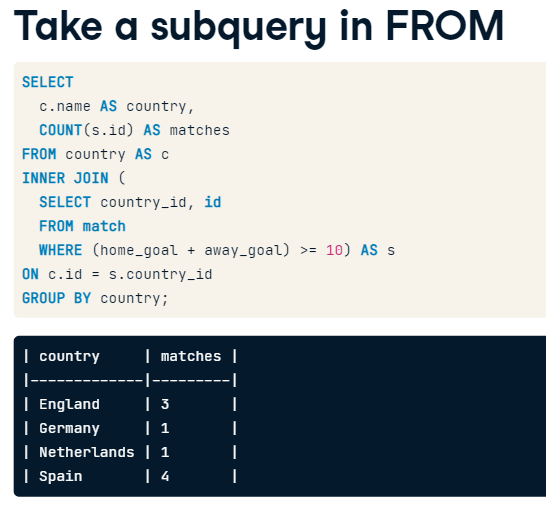




[mtkerdogan@gmail.com](mailto:mtkerdogan@gmail.com)

[bkeles74@gmail.com](mailto:bkeles74@gmail.com)

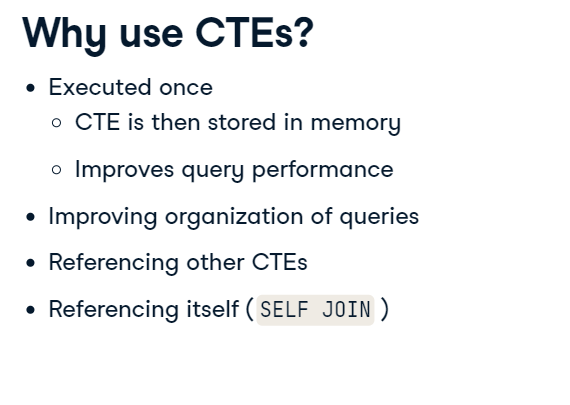


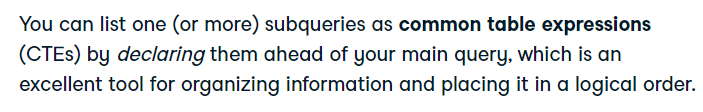


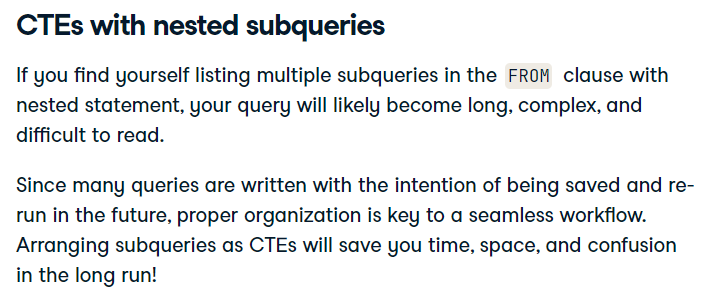


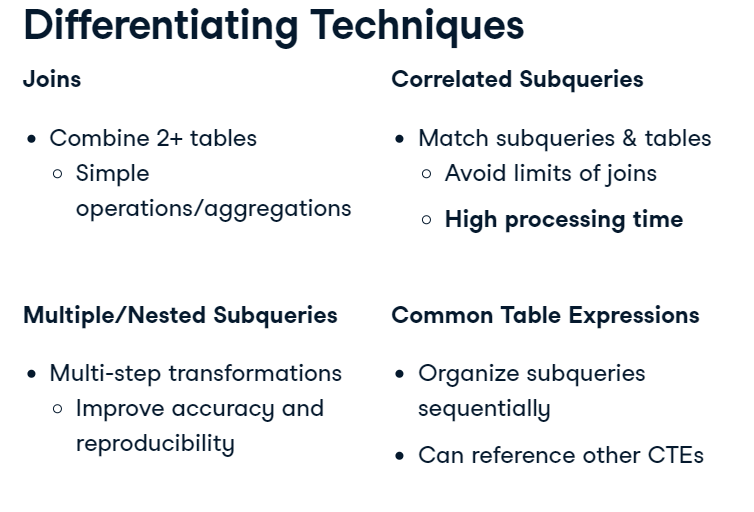


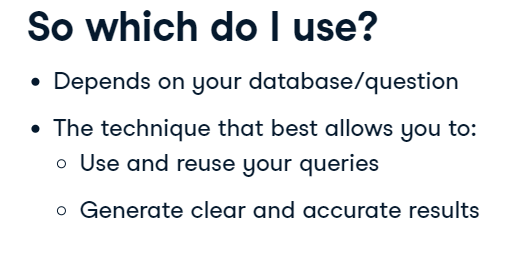


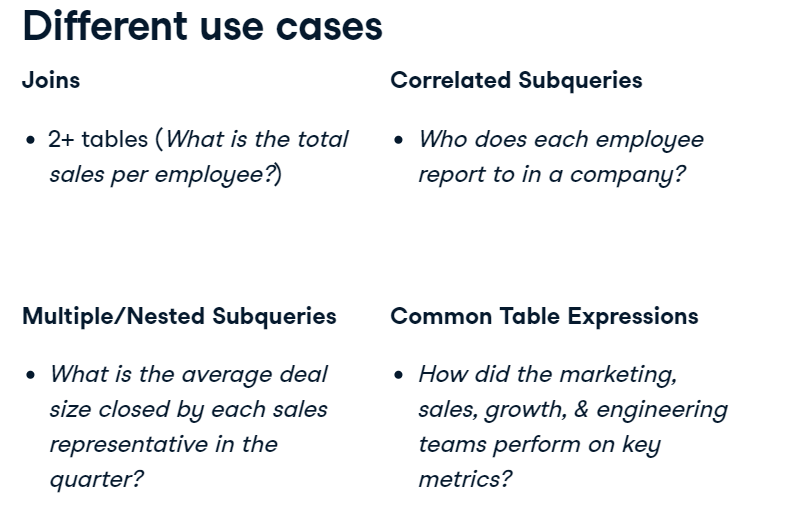


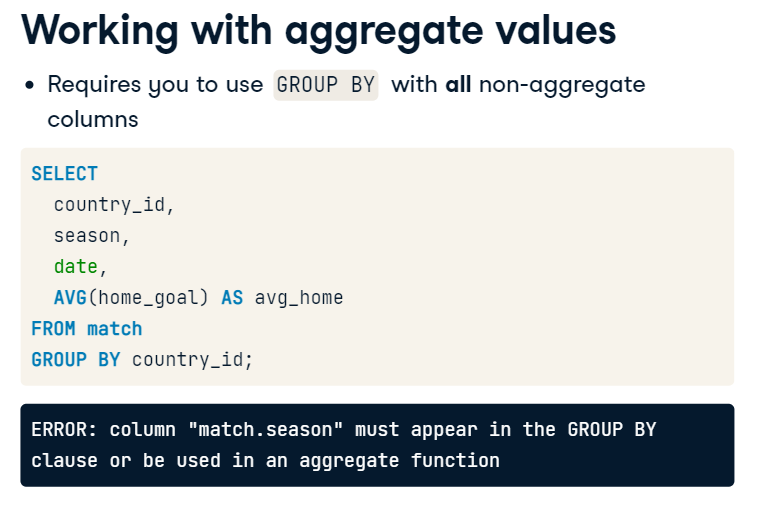


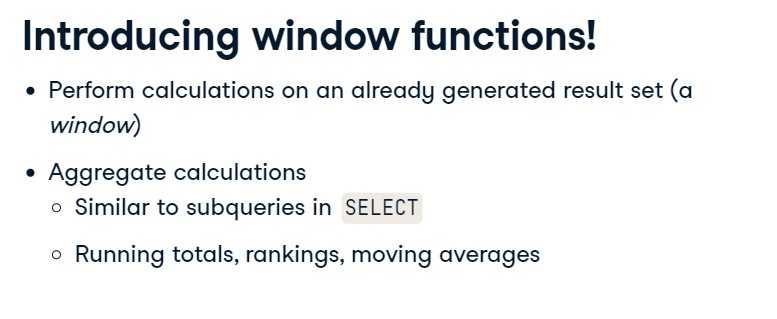


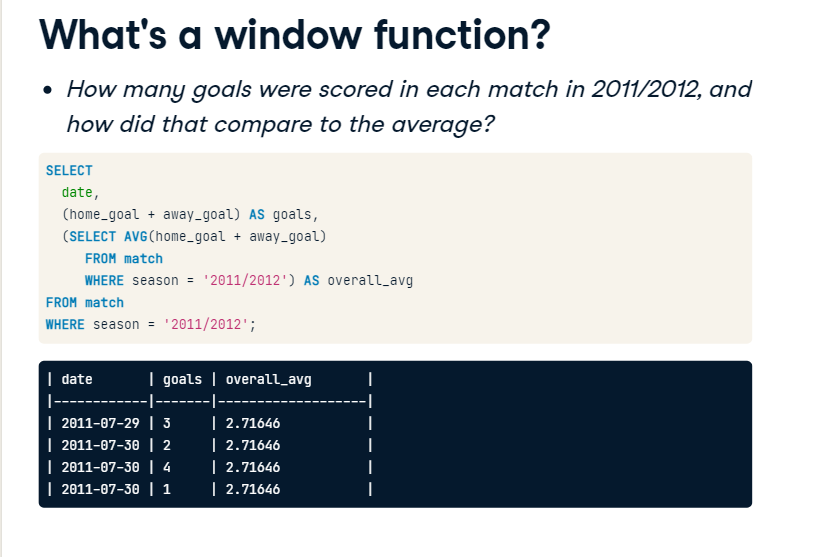


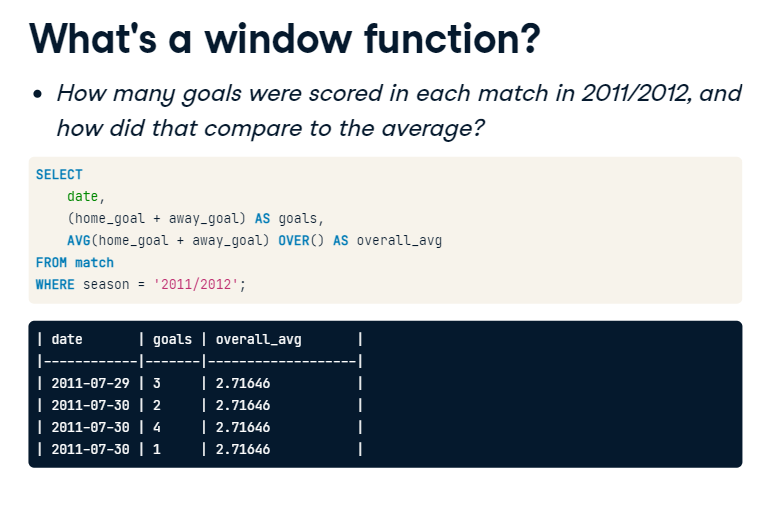




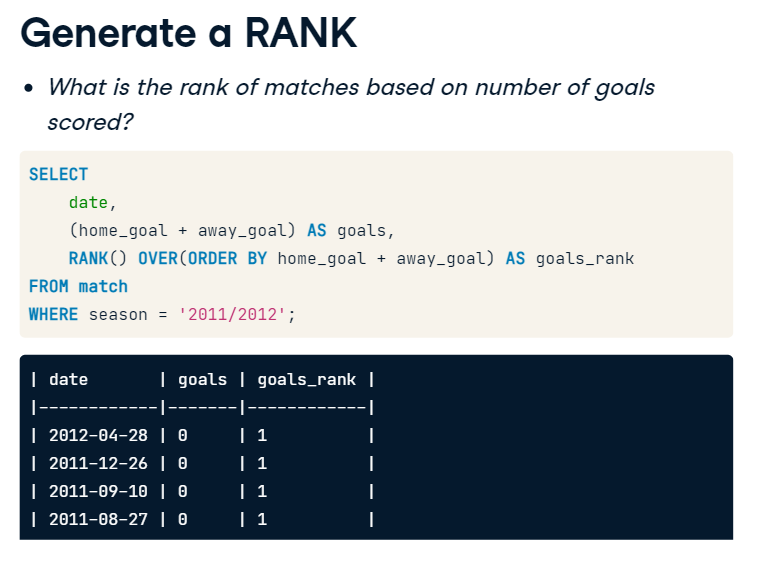




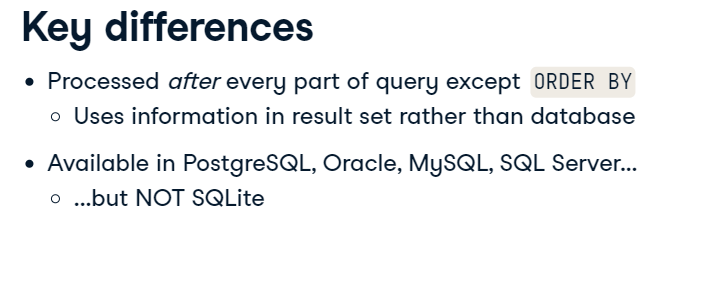


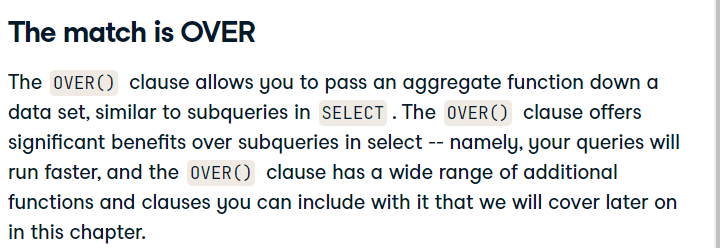


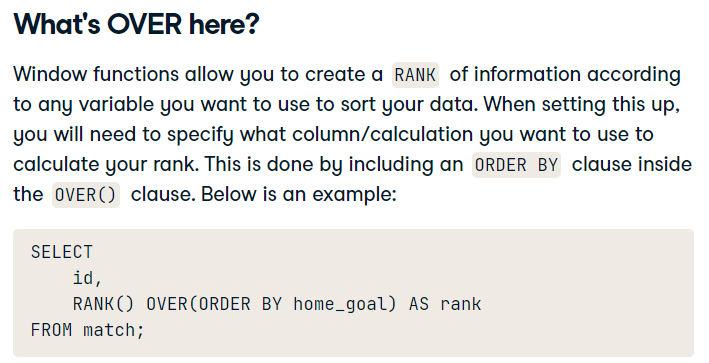


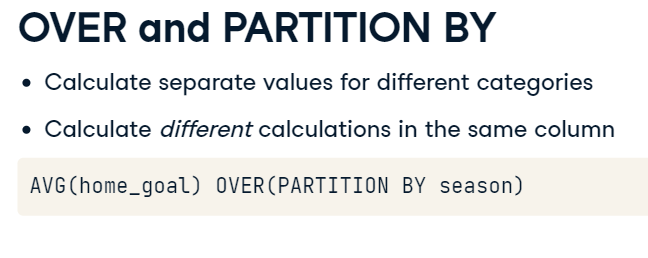


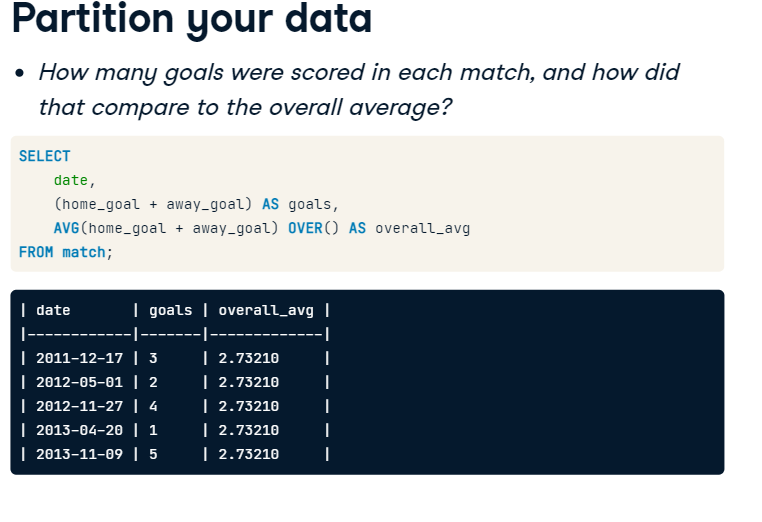


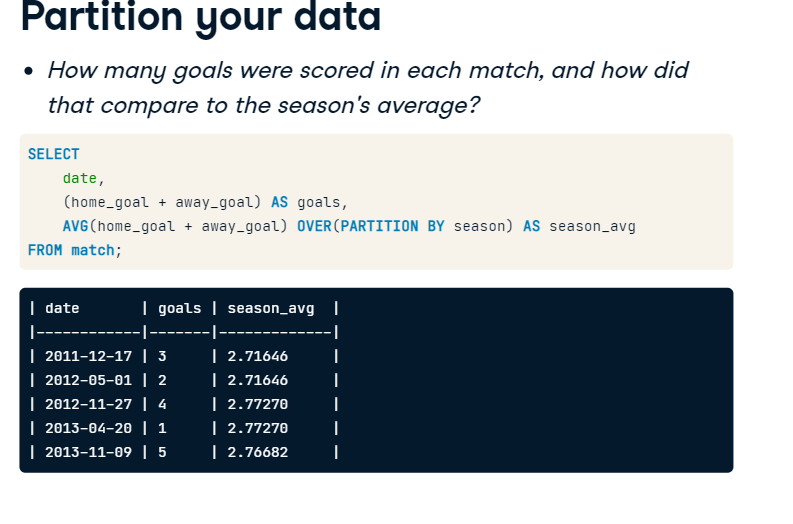


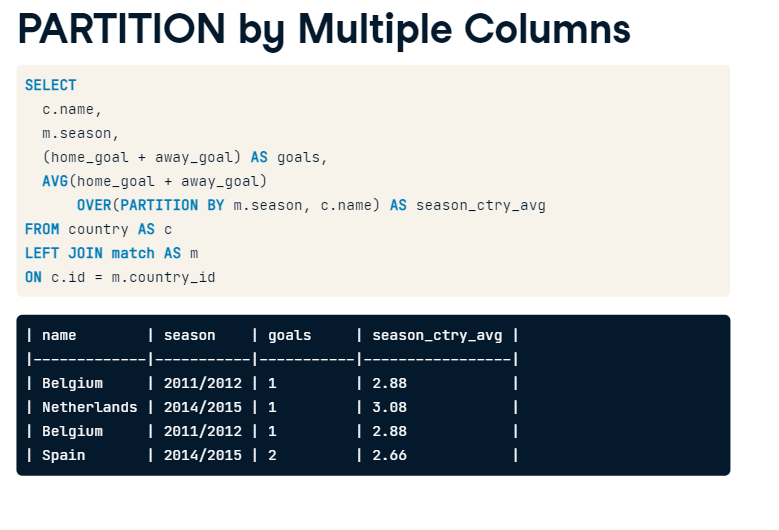


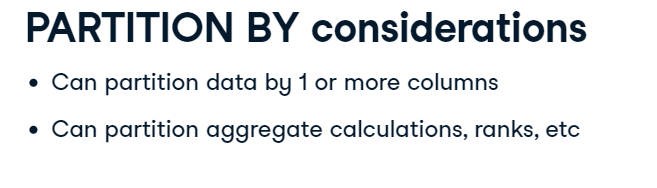


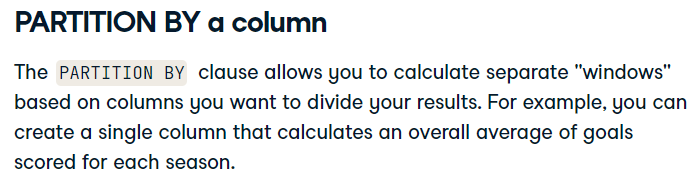


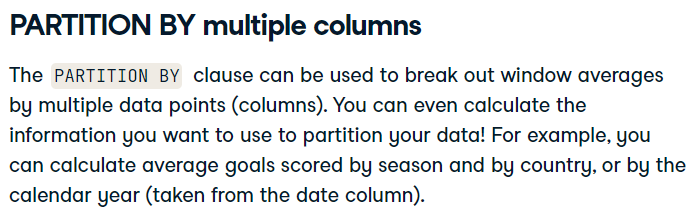


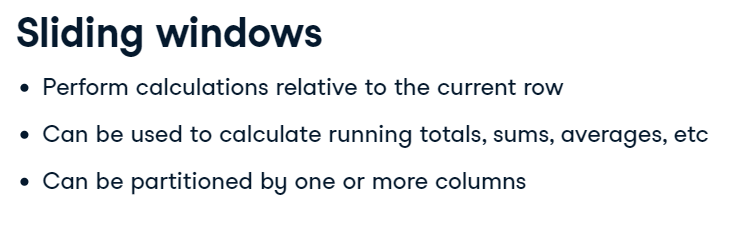


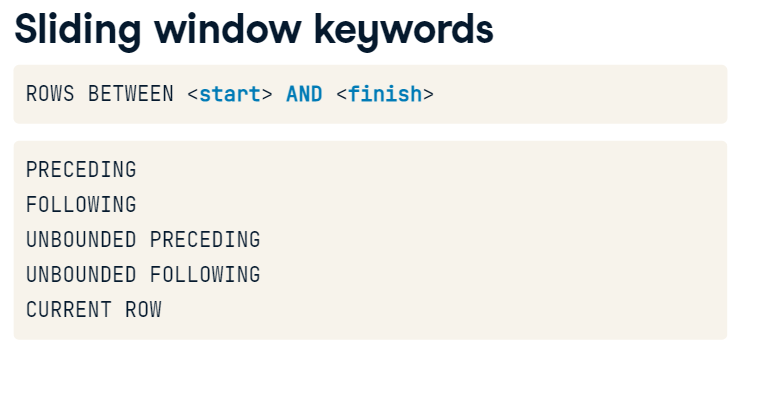


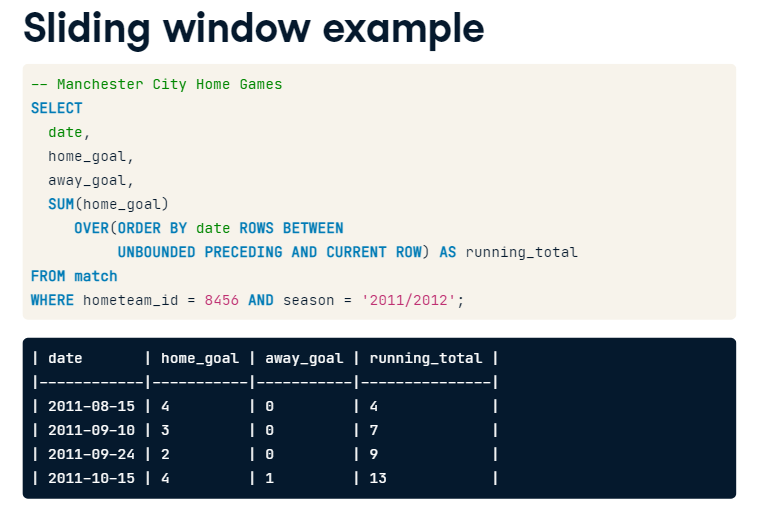


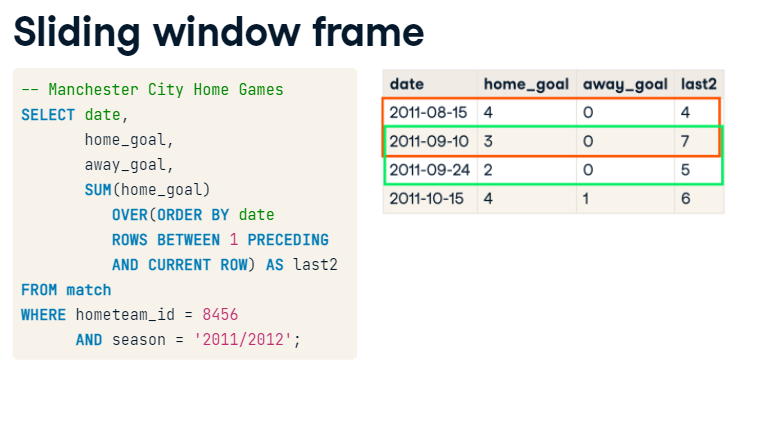


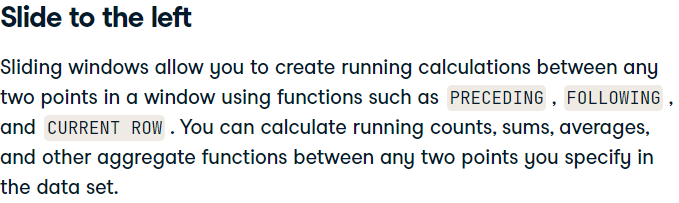


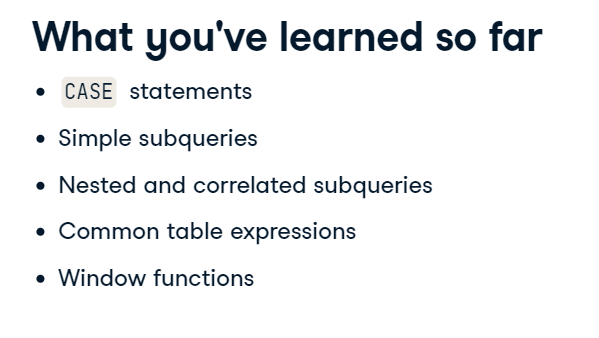


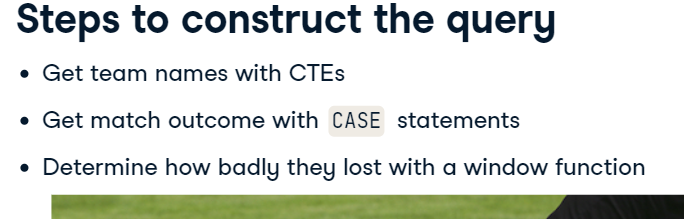












SELECT

    m.id,

    t.team\_long\_name,

    -- Identify matches as home/away wins or ties

    CASE WHEN m.home\_goal > m.away\_goal THEN 'MU Win'

        WHEN m.home\_goal < m.away\_goal THEN 'MU Loss'

        ELSE 'Tie' END AS outcome

FROM match AS m

-- Left join team on the home team ID and team API id

LEFT JOIN team AS t

ON m.hometeam\_id = t.team\_api\_id

WHERE

    -- Filter for 2014/2015 and Manchester United as the home team

    season = '2014/2015'

    AND t.team\_long\_name = 'Manchester United';

SELECT

    m.id,

    t.team\_long\_name,

    -- Identify matches as home/away wins or ties

    CASE WHEN m.home\_goal > m.away\_goal  THEN 'MU Loss'

        WHEN m.home\_goal < m.away\_goal  THEN 'MU Win'

        ELSE 'Tie' END AS outcome

-- Join team table to the match table

FROM match AS m

LEFT JOIN team AS t

ON m.awayteam\_id = t.team\_api\_id

WHERE

    -- Filter for 2014/2015 and Manchester United as the away team

    season = '2014/2015'

    AND t.team\_long\_name = 'Manchester United';

-- Set up the home team CTE

WITH home AS (

  SELECT m.id, t.team\_long\_name,

    CASE WHEN m.home\_goal > m.away\_goal THEN 'MU Win'

       WHEN m.home\_goal < m.away\_goal THEN 'MU Loss'

         ELSE 'Tie' END AS outcome

  FROM match AS m

  LEFT JOIN team AS t ON m.hometeam\_id = t.team\_api\_id),

-- Set up the away team CTE

away AS (

  SELECT m.id, t.team\_long\_name,

    CASE WHEN m.home\_goal > m.away\_goal THEN 'MU Win'

       WHEN m.home\_goal < m.away\_goal THEN 'MU Loss'

         ELSE 'Tie' END AS outcome

  FROM match AS m

  LEFT JOIN team AS t ON m.awayteam\_id = t.team\_api\_id)

-- Select team names, the date and goals

SELECT DISTINCT

    m.date,

    home.team\_long\_name AS home\_team,

    away.team\_long\_name AS away\_team,

    m.home\_goal,

    m.away\_goal

-- Join the CTEs onto the match table

FROM match AS m

LEFT JOIN home ON m.id = home.id

LEFT JOIN away ON m.id = away.id

WHERE m.season = '2014/2015'

      AND (home.team\_long\_name = 'Manchester United'

           OR away.team\_long\_name = 'Manchester Unite