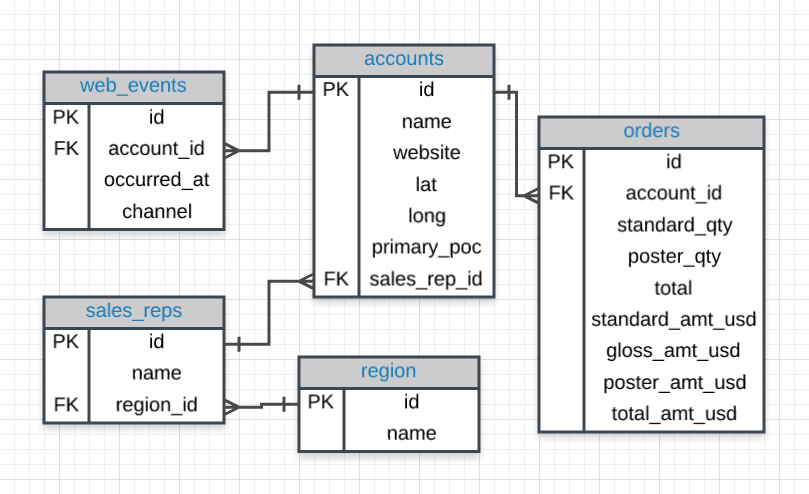
**Chapter 1: Introduction.**

**Entity Relationship Diagrams**

An **entity relationship diagram** (ERD) is a common way to view data in a database. Below is the ERD for the database we will use from Parch & Posey. These diagrams help you visualize the data you are analyzing including:

1. The names of the tables.
2. The columns in each table.
3. The way the tables work together.

**You can think of each of the boxes below as a spreadsheet.**



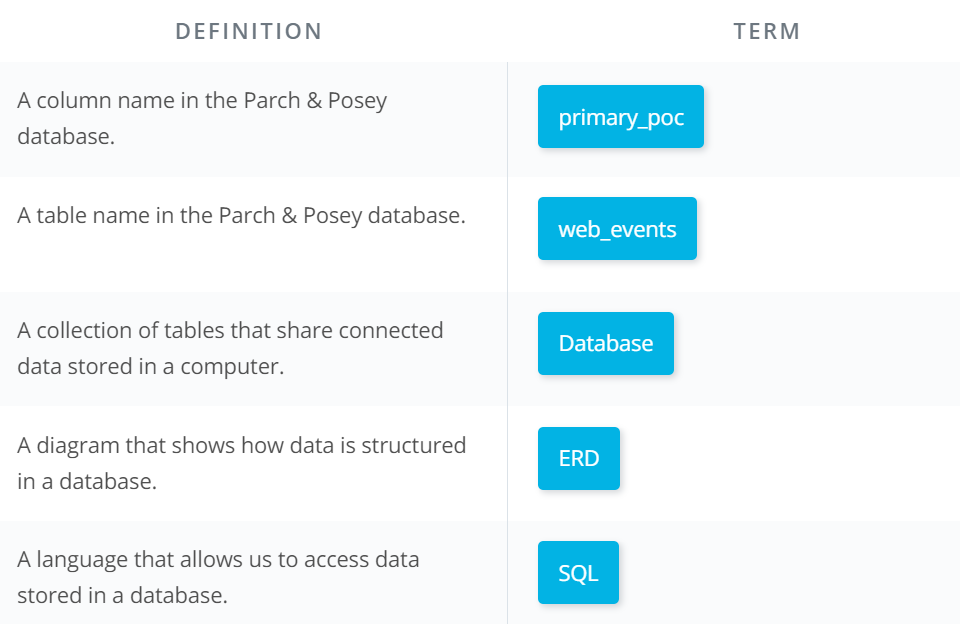
**What to Notice**

In the Parch & Posey database there are five tables (essentially 5 spreadsheets):

1. **web\_events**
2. **accounts**
3. **orders**
4. **sales\_reps**
5. **region**

You can think of each of these tables as an individual spreadsheet. Then the columns in each spreadsheet are listed below the table name. For example, the **region** table has two columns: id and name. Alternatively the **web\_events** table has four columns.

QUIZ 1



# Introduction – Why SQL

# I think it is an important distinction to say that SQL is a **language**. Hence, the last word of SQ****L**** being ****language****. SQL is used all over the place beyond the databases we will utilize in this class. With that being said, SQL is most popular for its interaction with databases.

There are some **major advantages** to using **traditional relational databases,** which we interact with using SQL. The five most apparent are:

* SQL is easy to understand.
* Traditional databases allow us to access data directly.
* Traditional databases allow us to audit and replicate our data.
* SQL is a great tool for analyzing multiple tables at once.
* SQL allows you to analyze more complex questions than dashboard tools like Google Analytics.

## SQL vs. NoSQL

You may have heard of NoSQL, which stands for not only SQL. Databases using NoSQL allow for you to write code that interacts with the data a bit differently than what we will do in this course. These NoSQL environments tend to be particularly popular for web based data, but less popular for data that lives in spreadsheets the way we have been analyzing data up to this point. One of the most popular NoSQL languages is called [MongoDB](https://www.mongodb.com/).

### Why Businesses Like Databases

1. **Data integrity is ensured** - only the data you want entered is entered, and only certain users are able to enter data into the database.
2. **Data can be accessed quickly** - SQL allows you to obtain results very quickly from the data stored in a database. Code can be optimized to quickly pull results.
3. **Data is easily shared** - multiple individuals can access data stored in a database, and the data is the same for all users allowing for consistent results for anyone with access to your database.

# Types of Databases

### SQL Databases

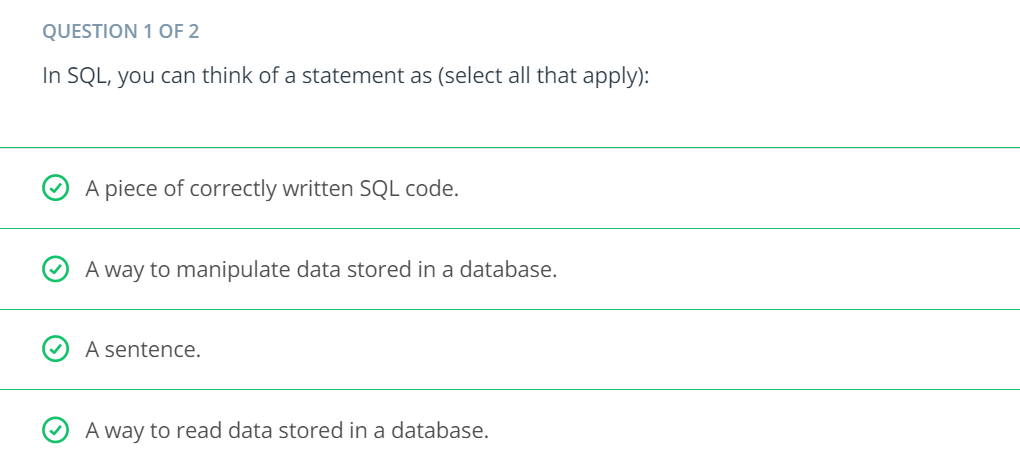
There are many different types of SQL databases designed for different purposes.

[**Postgres**](https://www.postgresql.org/) is a popular open-source database with a very complete library of analytical functions.

Some of the most popular databases include:

1. MySQL
2. Access
3. Oracle
4. Microsoft SQL Server
5. Postgres

You can also write SQL within other programming frameworks like Python, Scala, and HaDoop.



Here you were introduced to the SQL command that will be used in every query you write: SELECT ... FROM ....

1. **SELECT** indicates which column(s) you want to be given the data for.
2. **FROM** specifies from which table(s) you want to select the columns. Notice the columns need to exist in this table.

If you want to be provided with the data from all columns in the table, you use "\*", like so:

* SELECT \* FROM orders

Note that using SELECT does not *create* a new table with these columns in the database, it just provides the data to you as the results, or output, of this command.

**LIMIT** statement is useful when you want to see just the first few rows of a table. This can be much faster for loading than if we load the entire dataset.

The **LIMIT** command is always the very last part of a query. An example of showing just the first 10 rows of the orders table with all of the columns might look like the following:

**SELECT** \*

**FROM** orders

**LIMIT** 10;

select occurred\_at, account\_id, channel

from web\_events

limit 15;

**ORDER BY** statement allows us to sort our results using the data in any column. If you are familiar with Excel or Google Sheets, using **ORDER BY** is similar to sorting a sheet using a column. A key difference, however, is that **using ORDER BY in a SQL query only has temporary effects, for the results of that query, unlike sorting a sheet by column in Excel or Sheets.**

### Pro Tip

Remember DESC can be added after the column in your **ORDER BY** statement to sort in descending order, as the default is to sort in ascending order.

**More Examples**

**SELECT** **id**, account\_id, total\_amt\_usd

**FROM** orders

**ORDER** **BY** account\_id, total\_amt\_usd **DESC**;

**WHERE statement.**

We can display subsets of tables based on conditions that must be met. You can also think of the **WHERE** command as filtering the data.

**Example**:

Pulls the first 10 rows and all columns from the orders table that have a *total\_amt\_usd* less than 500.

**SELECT** \*

**FROM** orders

**WHERE** total\_amt\_usd < 500

**LIMIT** 10;

Commonly when we are using **WHERE** with non-numeric data fields, we use the **LIKE**, **NOT**, or **IN** operators.

## Derived Columns

Creating a new column that is a combination of existing columns is known as a **derived** column (or "calculated" or "computed" column). Usually you want to give a name, or "alias," to your new column using the **AS** keyword.

* + This derived column, and its alias, are generally only temporary, existing just for the duration of your query. The next time you run a query and access this table, the new column will not be there.

 these familiar mathematical operators will be useful:

1. \* (Multiplication)
2. + (Addition)
3. - (Subtraction)
4. / (Division)

Example

**SELECT** **id**, (standard\_amt\_usd/total\_amt\_usd)\*100 **AS** std\_percent, total\_amt\_usd

**FROM** orders

**LIMIT** 10;

### Introduction to Logical Operators

In the next concepts, you will be learning about **Logical Operators**. **Logical Operators** include:

1. **LIKE** This allows you to perform operations similar to using **WHERE** and =, but for cases when you might **not** know **exactly** what you are looking for.
2. **IN** This allows you to perform operations similar to using **WHERE** and =, but for more than one condition.
3. **NOT** This is used with **IN** and **LIKE** to select all of the rows **NOT LIKE** or **NOT IN** a certain condition.
4. **AND & BETWEEN** These allow you to combine operations where all combined conditions must be true.
5. **OR** This allows you to combine operations where at least one of the combined conditions must be true.

**The LIKE** **operator.**

It is extremely useful for working with text. You will use **LIKE** within a **WHERE** clause. The **LIKE** operator is frequently used with %.

>>> The % tells us that we might want any number of characters leading up to a particular set of characters or following a certain set of characters, as we saw with the **google** syntax above.

Example:

1. All the companies whose names start with 'C'

**SELECT** **name**

**FROM** accounts

**WHERE** **name** **LIKE** 'C%';

1. All companies whose names contain the string 'one' somewhere in the name.

**SELECT** **name**

**FROM** accounts

**WHERE** **name** **LIKE** '%one%';

1. All companies whose names end with 's'.

**SELECT** **name** **FROM** accounts

**WHERE** **name** **LIKE** '%s';

**The IN operator.**

The **IN** operator is useful for working with both numeric and text columns. This operator allows you to use an =, but for more than one item of that particular column.

We can check one, two or many column values for which we want to pull data, but all within the same query.

Examples:

1. Use the accounts table to find the account name, primary\_poc, and sales\_rep\_id for Walmart, Target, and Nordstrom.

**SELECT** **name**, primary\_poc, sales\_rep\_id

**FROM** accounts

**WHERE** **name** **IN** ('Walmart', 'Target', 'Nordstrom');

1. Use the web\_events table to find all information regarding individuals who were contacted via the channel of organic or adwords.

**SELECT** \*

**FROM** web\_events

**WHERE** channel **IN** ('organic', 'adwords');

**The** **NOT** **operator**.

The **NOT** operator is an extremely useful operator for working with the previous two operators we introduced: **IN** and **LIKE**.

By specifying **NOT LIKE** or **NOT IN**, we can grab all of the rows that do not meet a particular criteria.

Example *NOT IN:*

Use the accounts table to find the account name, primary\_poc, and sales\_rep\_id for all stores except Walmart, Target, and Nordstrom.

**SELECT** **name**, primary\_poc, sales\_rep\_id

**FROM** accounts

**WHERE** **name** **NOT** **IN** ('Walmart', 'Target', 'Nordstrom');

Example *NOT LIKE:*

All the companies whose names do not start with 'C'.

**SELECT** **name**

**FROM** accounts

**WHERE** **name** **NOT** **LIKE** 'C%';

**AND & BETWEEN.**

The **AND** operator is used within a **WHERE** statement to consider more than one logical clause at a time. Each time you link a new statement with an **AND**, you will need to specify the column you are interested in looking at.

This operator works with all of the operations we have seen so far including arithmetic operators (+, \*, -, /). **LIKE**, **IN**, and **NOT** logic can also be linked together using the **AND** operator.

# BETWEEN Operator

Sometimes we can make a cleaner statement using **BETWEEN** than we can using **AND**. Particularly this is true when we are using the same column for different parts of our **AND** statement.

Instead of writing :

WHERE column >= 6 AND column <= 10

we can instead write, equivalently:

WHERE column BETWEEN 6 AND 10

**Example 1:**

When you use the BETWEEN operator in SQL, do the results include the values of your endpoints, or not? Figure out the answer to this important question by writing a query that displays the order date and gloss\_qty data for all orders where gloss\_qty is between 24 and 29. Then look at your output to see if the BETWEEN operator included the begin and end values or not.

**SELECT** occurred\_at, gloss\_qty

**FROM** orders

**WHERE** gloss\_qty **BETWEEN** 24 **AND** 29;

*You should notice that there are a number of rows in the output of this query where the gloss\_qty values are 24 or 29. So the answer to the question is that yes, the BETWEEN operator in SQL is inclusive; that is, the endpoint values are included. So the BETWEEN statement in this query is equivalent to having written "WHERE gloss\_qty >= 24 AND gloss\_qty <= 29."*

**Example 2:**

When you use the BETWEEN operator in SQL, do the results include the values of your endpoints, or not? Figure out the answer to this important question by writing a query that displays the order date and gloss\_qty data for all orders where gloss\_qty is between 24 and 29. Then look at your output to see if the BETWEEN operator included the begin and end values or not.

**SELECT** occurred\_at, gloss\_qty

**FROM** orders

**WHERE** gloss\_qty **BETWEEN** 24 **AND** 29;

*You should notice that there are a number of rows in the output of this query where the gloss\_qty values are 24 or 29. So the answer to the question is that yes, the BETWEEN operator in SQL is inclusive; that is, the endpoint values are included. So the BETWEEN statement in this query is equivalent to having written "WHERE gloss\_qty >= 24 AND gloss\_qty <= 29."*

**OR Statement.**

The **OR** operator can combine multiple statements. Each time you link a new statement with an **OR**, you will need to specify the column you are interested in looking at.

You may link as many statements as you would like to consider at the same time.

This operator works with all of the operations we have seen so far including arithmetic operators (+, \*, -, /), **LIKE**, **IN**, **NOT**, **AND**, and **BETWEEN** logic can all be linked together using the **OR** operator.

**Example 1:**

Find list of **orders** ids where either gloss\_qty or poster\_qty is greater than 4000. Only include the id field in the resulting table.

**SELECT** **id**

**FROM** orders

**WHERE** gloss\_qty > 4000 **OR** poster\_qty > 4000;

**Example 2:**

Find all the company names that start with a 'C' or 'W', and the primary contact **contains** 'ana' or 'Ana', but it doesn't contain 'eana'.

**SELECT** \*

**FROM** accounts

**WHERE** (**name** **LIKE** 'C%' **OR** **name** **LIKE** 'W%')

**AND** ((primary\_poc **LIKE** '%ana%' **OR** primary\_poc **LIKE** '%Ana%')

**AND** primary\_poc **NOT** **LIKE** '%eana%');

**Chapter 2: SQL Joins**

**Database Normalization**

When creating a database, it is really important to think about how data will be stored. This is known as **normalization**, and it is a huge part of most SQL classes. If you are in charge of setting up a new database, it is important to have a thorough understanding of database **normalization**.

There are essentially three ideas that are aimed at database normalization:

1. Are the tables storing logical groupings of the data?
2. Can I make changes in a single location, rather than in many tables for the same information?
3. Can I access and manipulate data quickly and efficiently?

>> The whole purpose of **JOIN** statements is to allow us to pull data from more than one table at a time.

>> Again - **JOIN**s are useful for allowing us to pull data from multiple tables. This is both simple and powerful all at the same time.

>> We use **ON** clause to specify a **JOIN** condition which is a logical statement to combine the table in FROM and JOIN statements.

# First JOIN

Pull all the information from only the **orders** table:

**SELECT** orders.\*

**FROM** orders

**JOIN** accounts

**ON** orders.account\_id = accounts.**id**;

For example, if we want to pull only the **account name** and the dates in which that account placed an order, but none of the other columns, we can do this with the following query:

**SELECT** accounts.**name**, orders.occurred\_at

**FROM** orders

**JOIN** accounts

**ON** orders.account\_id = accounts.**id**;

This query pulls all the columns from *both* the **accounts** and **orders** table.

**SELECT** \*

**FROM** orders

**JOIN** accounts

**ON** orders.account\_id = accounts.**id**;

# Keys

### Primary Key (PK)

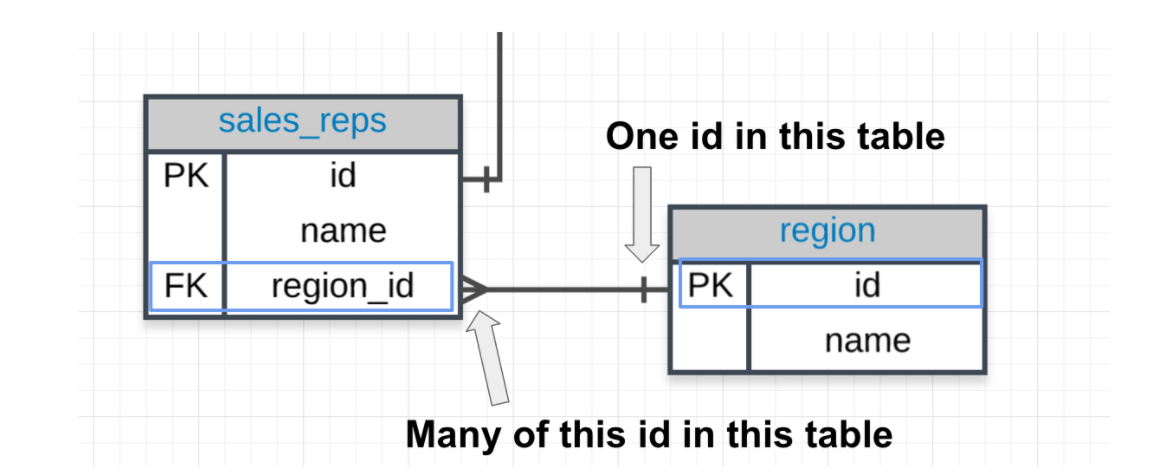
A **primary key** is a unique column in a particular table. This is the first column in each of our tables. Here, those columns are all called **id**, but that doesn't necessarily have to be the name. **It is common that the primary key is the first column in our tables in most databases.**

### Foreign Key (FK)

A **foreign key** is a column in one table that is a primary key in a different table. We can see in the Parch & Posey ERD that the foreign keys are:

1. **region\_id**
2. **account\_id**
3. **sales\_rep\_id**

Each of these is linked to the **primary key** of another table. An example is shown in the image below:



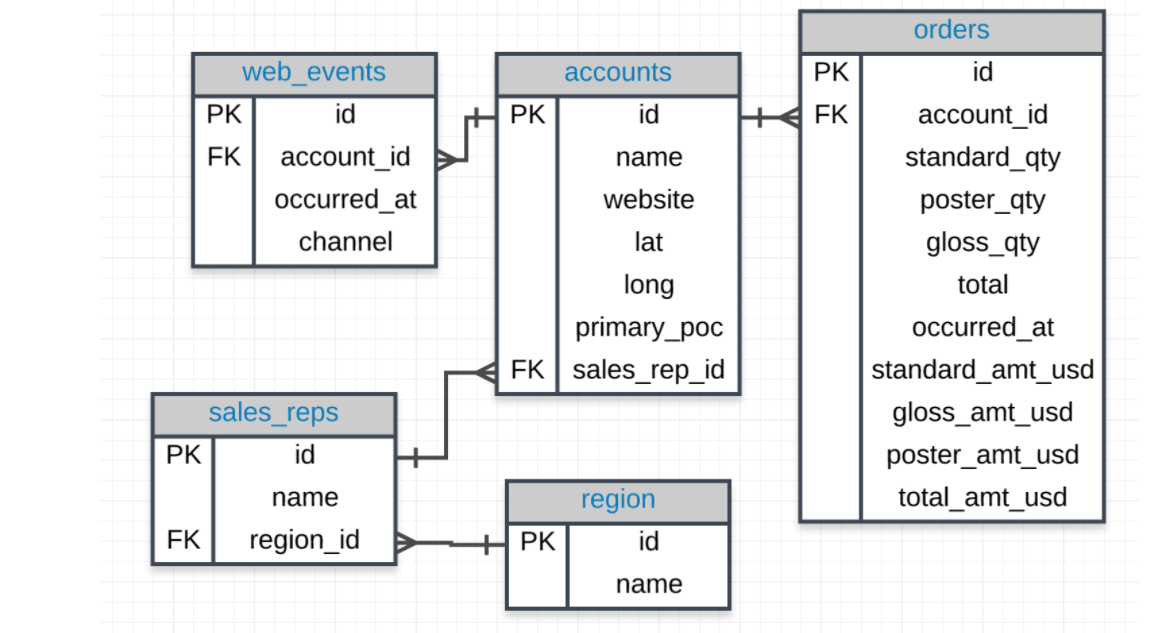
### **Primary - Foreign Key Link**

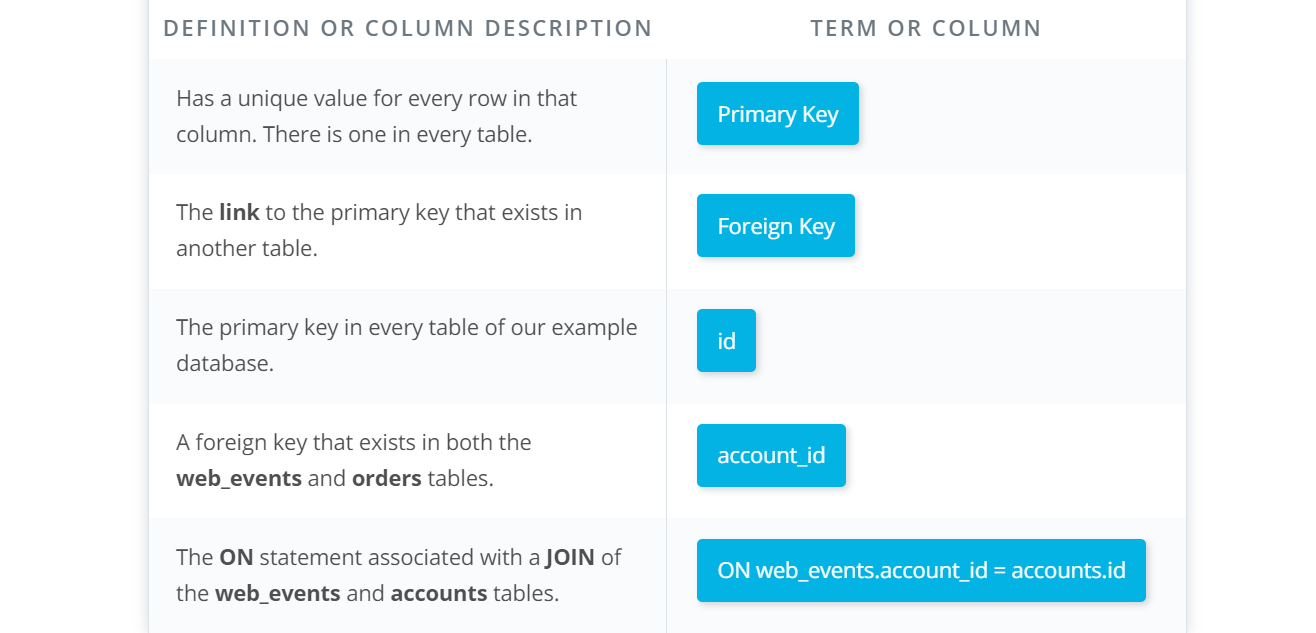
In the above image you can see that:

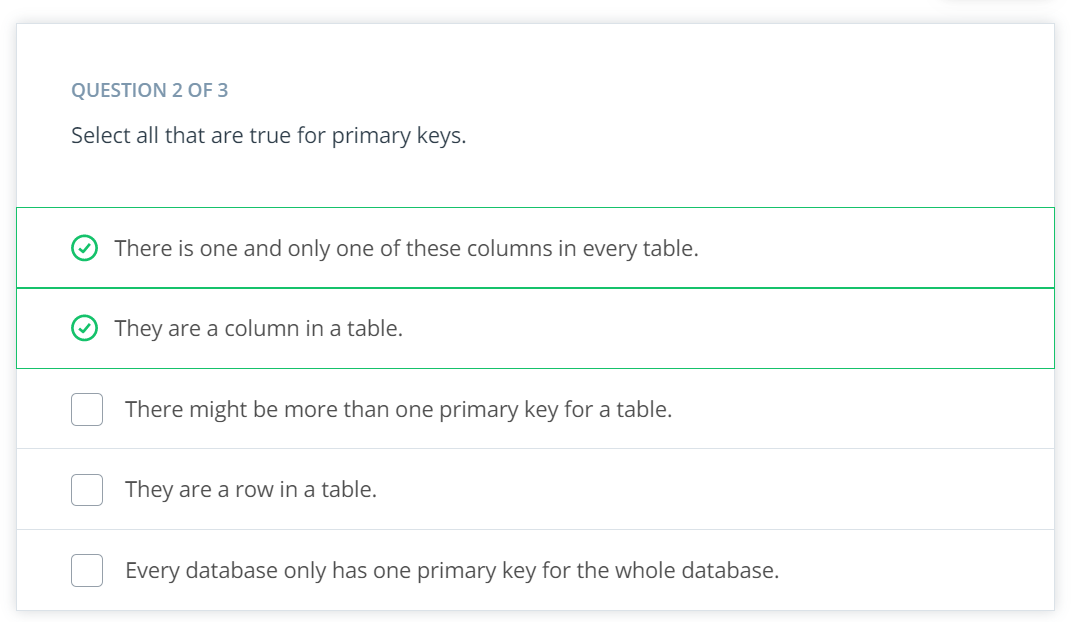
1. The **region\_id** is the foreign key.
2. The region\_id is **linked** to id - this is the primary-foreign key link that connects these two tables.
3. The crow's foot shows that the **FK** can actually appear in many rows in the **sales\_reps** table.
4. While the single line is telling us that the **PK** shows that id appears only once per row in this table.

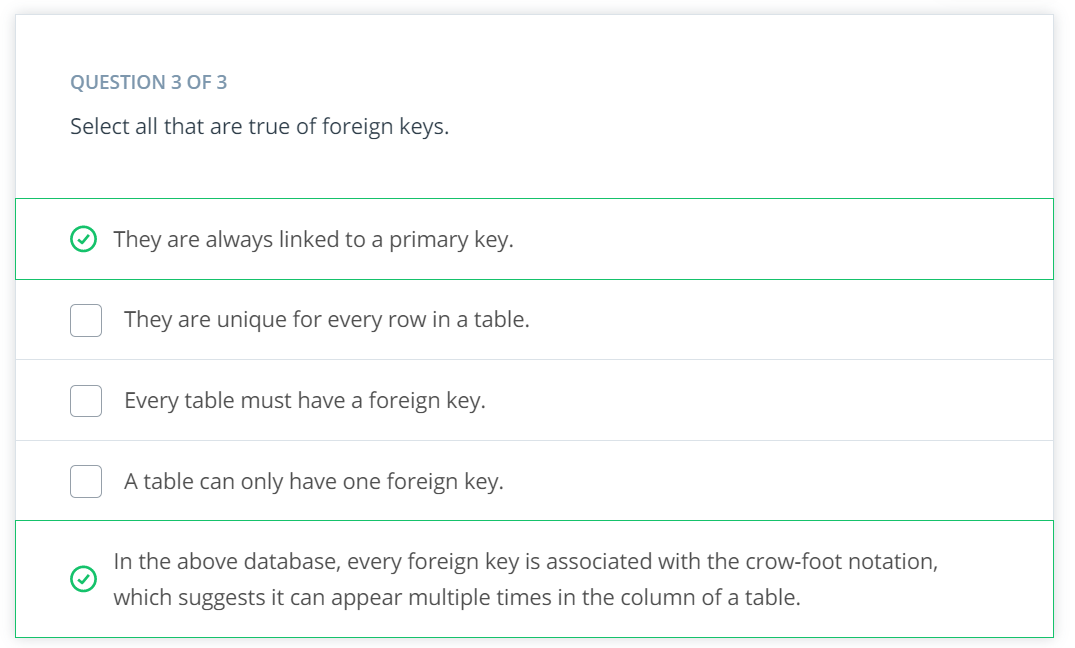
If you look through the rest of the database, you will notice this is always the case for a primary-foreign key relationship.

# Helpful ERD For Answering the Below Questions





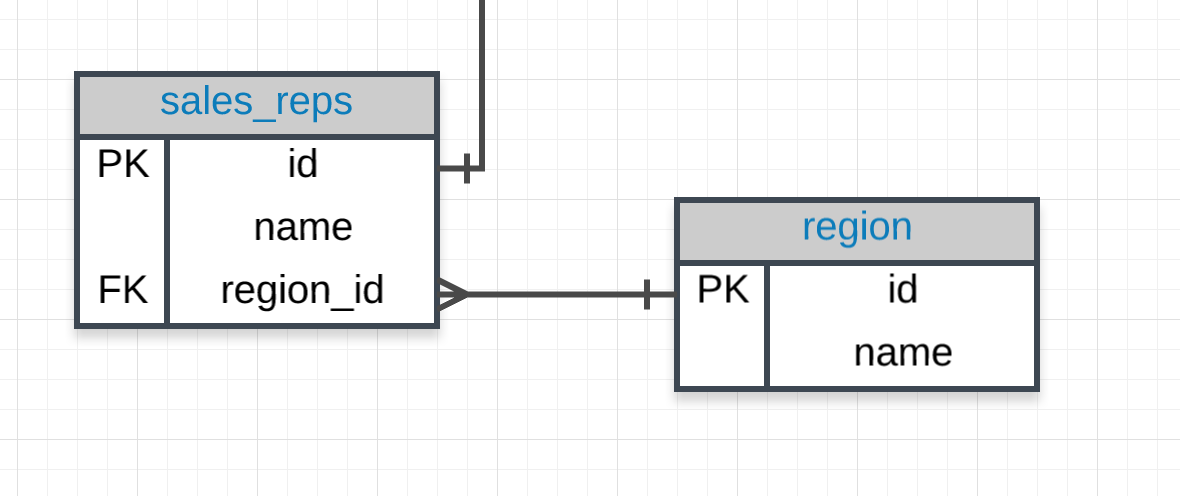


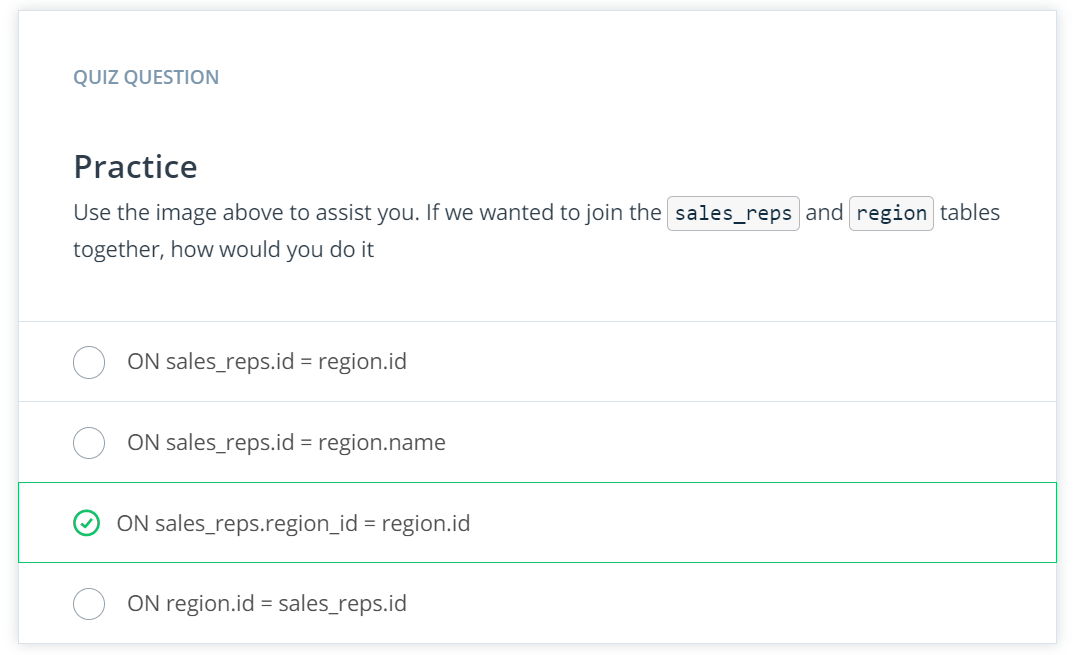


# Notice

Notice our SQL query has the two tables we would like to join - one in the **FROM** and the other in the **JOIN**. Then in the **ON**, we will **ALWAYs** have the **PK** equal to the **FK**:

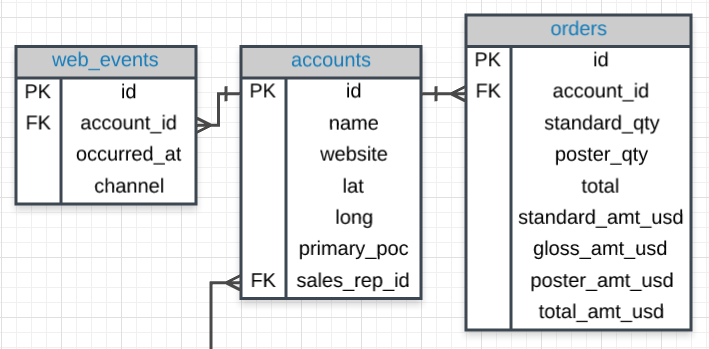
The way we join any two tables is in this way: linking the **PK** and **FK** (generally in an **ON** statement).





# JOIN More than Two Tables

This same logic can actually assist in joining more than two tables together. Look at the three tables below.



# The Code.

If we wanted to join all three of these tables, we could use the same logic. The code below pulls all of the data from all of the joined tables.

**SELECT** \*

**FROM** web\_events

**JOIN** accounts

**ON** web\_events.account\_id = accounts.**id**

**JOIN** orders

**ON** accounts.**id** = orders.account\_id

Alias.

When we **JOIN** tables together, it is nice to give each table an **alias**. Frequently an alias is just the first letter of the table name. You actually saw something similar for column names in the **Arithmetic Operators** concept.

Example:

**FROM** tablename AS t1

**JOIN** tablename2 AS t2

Before, you saw something like:

**SELECT** col1 + col2 **AS** total, col3

Each of the above could be written in the following way instead, and they would still produce the **exact same results**:

**FROM** tablename t1

**JOIN** tablename2 t2

and

**SELECT** col1 + col2 total, col3

# Aliases for Columns in Resulting Table

While aliasing tables is the most common use case. It can also be used to alias the columns selected to have the resulting table reflect a more readable name.

Example:

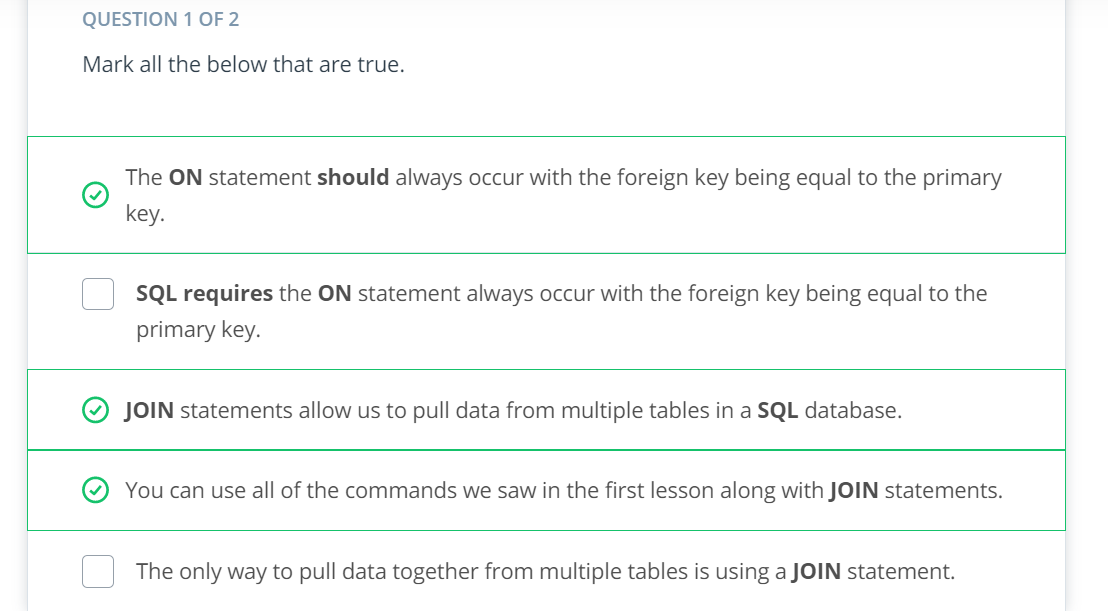
**Select** t1.column1 aliasname, t2.column2 aliasname2

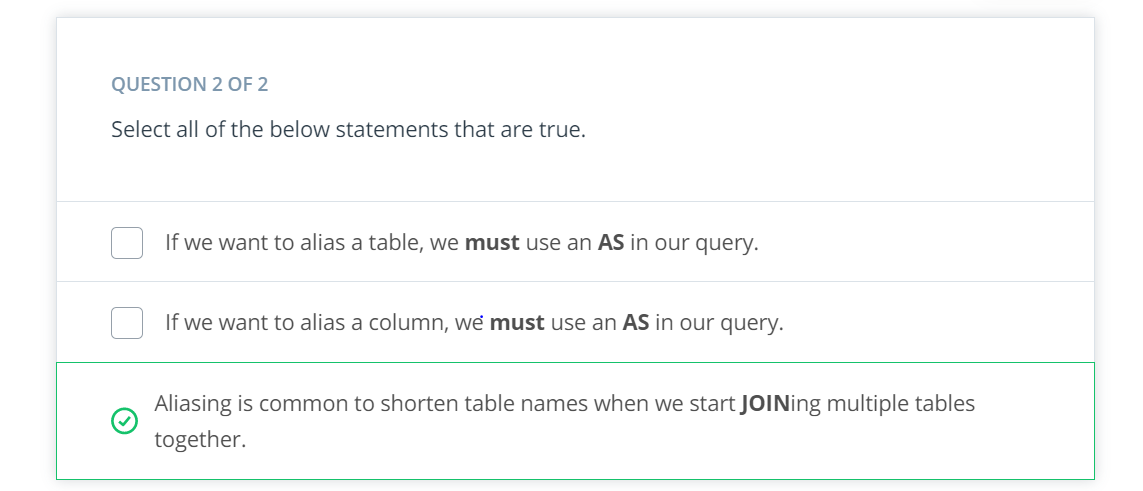
**FROM** tablename **AS** t1

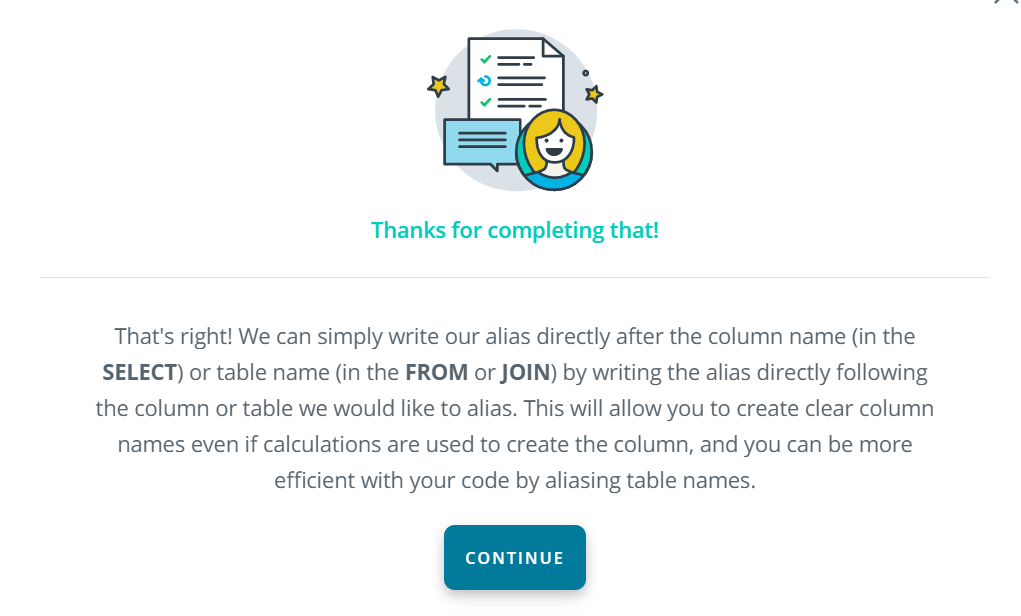
**JOIN** tablename2 **AS** t2

The alias name fields will be what shows up in the returned table instead of t1.column1 and t2.column2

| **aliasname** | **aliasname2** |
| --- | --- |
| example row | example row |
| example row | example row |







**Solutions**

1. Provide a table for all the for all **web\_events** associated with account **name** of Walmart. There should be three columns. Be sure to include the primary\_poc, time of the event, and the channel for each event. Additionally, you might choose to add a fourth column to assure only Walmart events were chosen.

**SELECT** a.primary\_poc, w.occurred\_at, w.channel, a.**name**

**FROM** web\_events w

**JOIN** accounts a

**ON** w.account\_id = a.**id**

**WHERE** a.**name** = 'Walmart';

1. Provide a table that provides the **region** for each **sales\_rep** along with their associated **accounts**. Your final table should include three columns: the region **name**, the sales rep **name**, and the account **name**. Sort the accounts alphabetically (A-Z) according to account name.

**SELECT** r.**name** region, s.**name** rep, a.**name** **account**

**FROM** sales\_reps s

**JOIN** region r

**ON** s.region\_id = r.**id**

**JOIN** accounts a

**ON** a.sales\_rep\_id = s.**id**

**ORDER** **BY** a.**name**;

1. Provide the **name** for each region for every **order**, as well as the account **name** and the **unit price** they paid (total\_amt\_usd/total) for the order. Your final table should have 3 columns: **region name**, **account name**, and **unit price**. A few accounts have 0 for **total**, so I divided by (total + 0.01) to assure not dividing by zero.

**SELECT** r.**name** region, a.**name** **account**, o.total\_amt\_usd/(o.total + 0.01) unit\_price

**FROM** region r

**JOIN** sales\_reps s

**ON** s.region\_id = r.**id**

**JOIN** accounts a

**ON** a.sales\_rep\_id = s.**id**

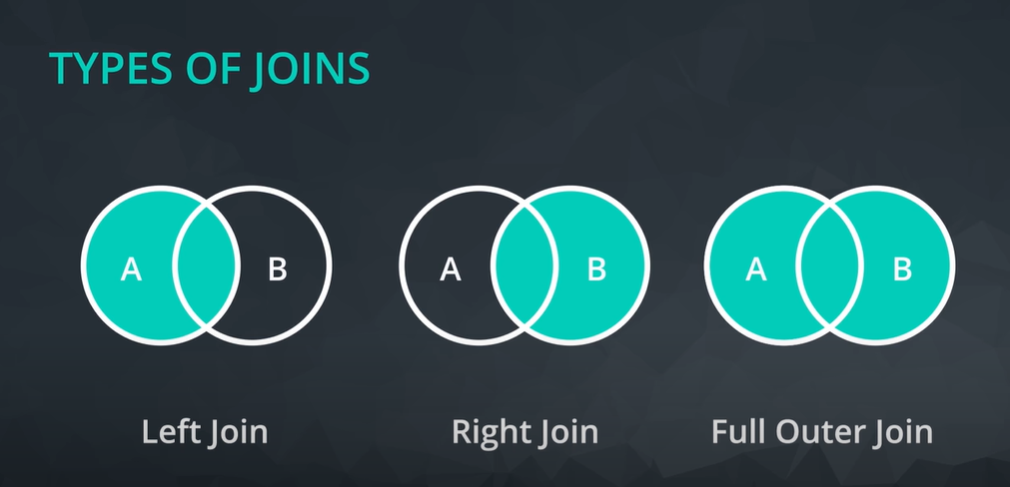
**JOIN** orders o

**ON** o.account\_id = a.**id**;

### **JOINs**

Notice each of these new **JOIN** statements pulls all the same rows as an **INNER JOIN**, which you saw by just using **JOIN**, but they also potentially pull some additional rows.

If there is not matching information in the **JOIN**ed table, then you will have columns with empty cells. These empty cells introduce a new data type called **NULL**.



### INNER JOINs

Notice **every** JOIN we have done up to this point has been an **INNER JOIN**. That is, we have always pulled rows only if they exist as a match across two tables.

You might see the SQL syntax of

LEFT OUTER **JOIN**

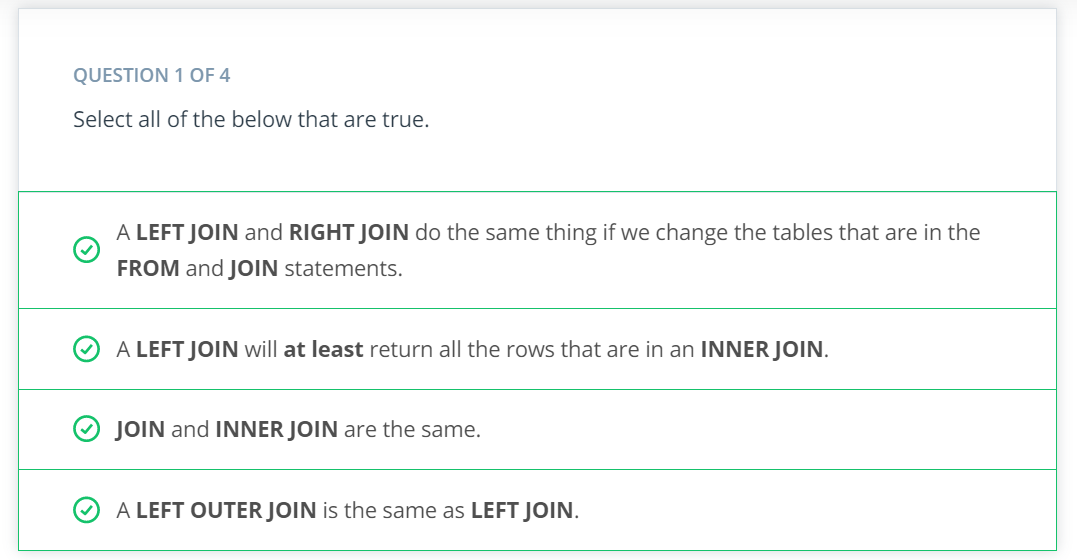
OR

RIGHT OUTER **JOIN**

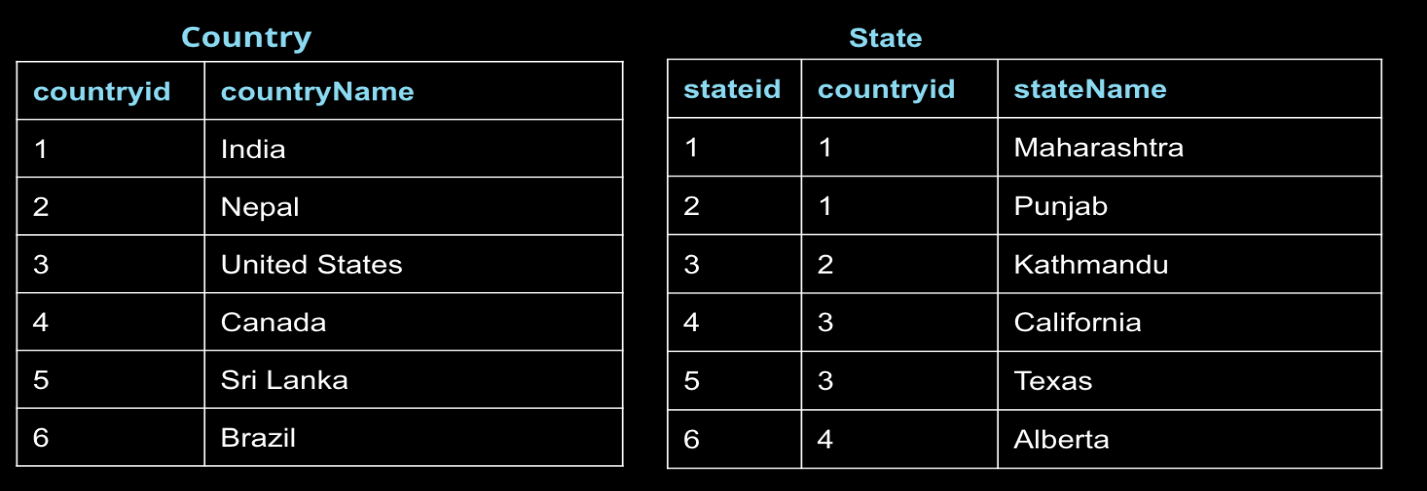
These are the exact same commands as the **LEFT JOIN** and **RIGHT JOIN**

### **OUTER JOINS**

The last type of join is a full outer join. This will return the inner join result set, as well as any unmatched rows from either of the two tables being joined.



The two small tables below are will test your knowledge of **JOIN**s.



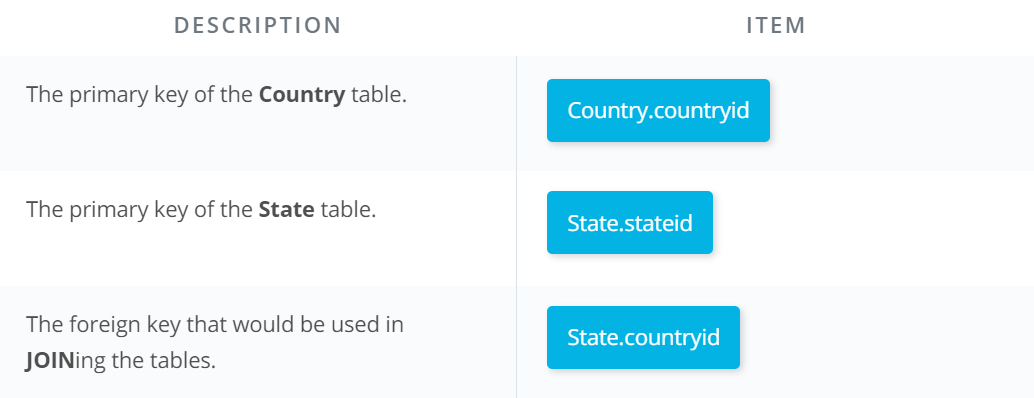
**Country** has 6 rows and 2 columns:

* **countryid** and **countryName**

**State** has 6 rows and 3 columns:

* **stateid**, **countryid**, and **stateName**

Use the above tables to determine the solution to the following questions.



### QUESTION 3 OF 4

Match the results of the query to the description (Use table above).

**SELECT** **c**.countryid, **c**.countryName, s.stateName

**FROM** Country **c**

**JOIN** State s

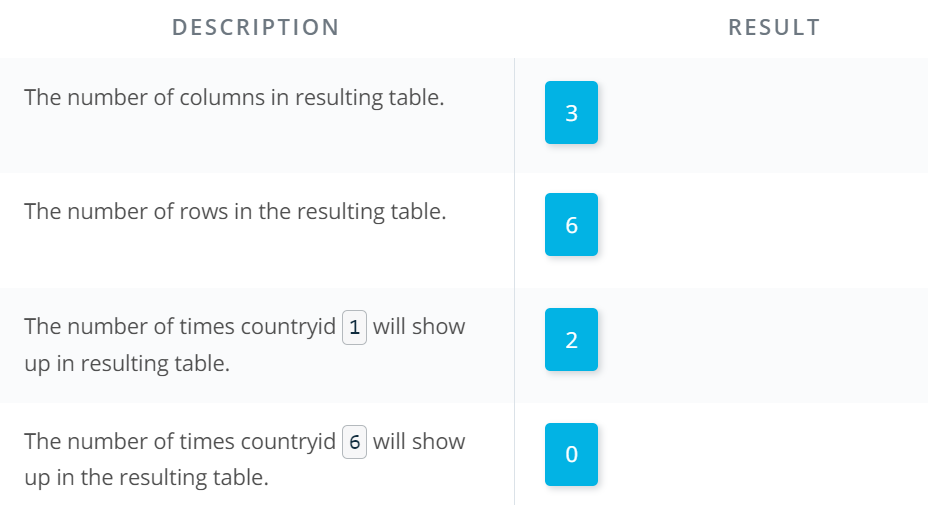
**ON** **c**.countryid = s.countryid;

**SELECT** **c**.countryid, **c**.countryName, s.stateName

**FROM** Country **c**

**LEFT** **JOIN** State s

**ON** **c**.countryid = s.countryid;



NOTE: Since this is a **JOIN** (**INNER JOIN** technically), we only get rows that show up in both tables. Therefore our resulting table will essentially look like the right table with the **countryName** pulled in as a column. Since 1, 2, 3, and 4 are **countryid**s in both tables, this information will be pulled together. The **countryid**s of 5 and 6 only show up in the **Country** table. Therefore, these will be dropped.

### QUESTION 4 OF 4

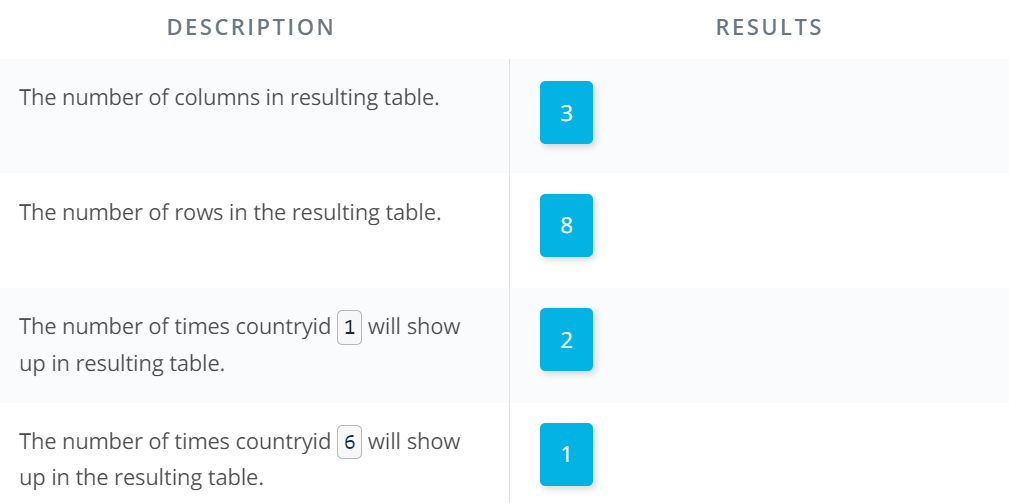
Match the results of the query to the description (Use table above).

**SELECT** **c**.countryid, **c**.countryName, s.stateName

**FROM** Country **c**

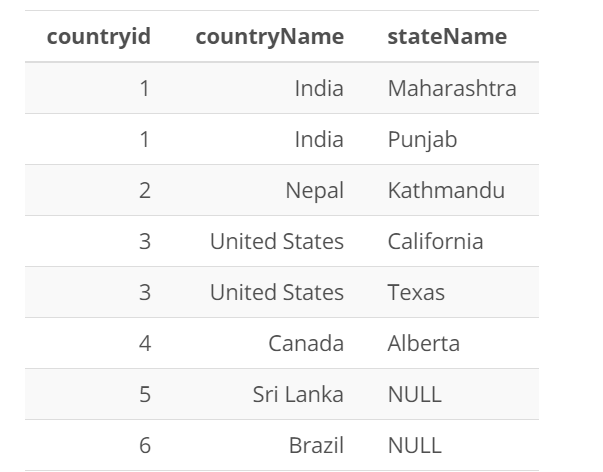
**LEFT** **JOIN** State s

**ON** **c**.countryid = s.countryid;



NOTE: We have a column for each of the identified elements in our **SELECT** statement. We will have all of the same rows as in a **JOIN** statement, but we also will obtain the additional two rows in the **Country** table that are not in the **State** table for **Sri Lanka** and **Brazil**.

The resulting table will look like:



**INTRODUCTION TO AGGREGATION.**

**Introduction to NULLs.**

**NULLs** are a datatype that specifies where no data exists in SQL. They are often ignored in our aggregation functions, which you will get a first look at in the next concept using **COUNT**.

**NULL**s are different than a zero - they are cells where data does not exist.

When identifying **NULL**s in a **WHERE** clause, we write **IS NULL** or **IS NOT NULL**. We don't use =, because **NULL** isn't considered a value in SQL. Rather, it is a property of the data.

# NULLs - Expert Tip

There are two common ways in which you are likely to encounter **NULL**s:

* **NULL**s frequently occur when performing a **LEFT** or **RIGHT JOIN**. You saw in the last lesson - when some rows in the left table of a left join are not matched with rows in the right table, those rows will contain some **NULL** values in the result set.
* **NULL**s can also occur from simply missing data in our database.

### **Expert Tip.**

Functionally, **MIN** and **MAX** are similar to **COUNT** in that they can be used on non-numerical columns. Depending on the column type, **MIN** will return the lowest number, earliest date, or non-numerical value as early in the alphabet as possible. As you might suspect, **MAX** does the opposite—it returns the highest number, the latest date, or the non-numerical value closest alphabetically to “Z.”

### **MEDIAN - Expert Tip**

One quick note that a median might be a more appropriate measure of center for this data, but finding the median happens to be a pretty difficult thing to get using SQL alone — so difficult that finding a median is occasionally asked as an interview question.

**GROUP BY**

The key takeaways here:

* **GROUP BY** can be used to aggregate data within subsets of the data. For example, grouping for different accounts, different regions, or different sales representatives.
* Any column in the **SELECT** statement that is not within an aggregator must be in the **GROUP BY** clause.
* The **GROUP BY** always goes between **WHERE** and **ORDER BY**.
* **ORDER BY** works like **SORT** in spreadsheet software.

### **GROUP BY - Expert Tip**

Before we dive deeper into aggregations using **GROUP BY** statements, it is worth noting that SQL evaluates the aggregations before the **LIMIT** clause. If you don’t group by any columns, you’ll get a 1-row result—no problem there. If you group by a column with enough unique values that it exceeds the **LIMIT** number, the aggregates will be calculated, and then some rows will simply be omitted from the results.

### GROUP BY QUIZ

1. What was the smallest order placed by each **account** in terms of **total usd**. Provide only two columns - the account **name** and the **total usd**. Order from smallest dollar amounts to largest.

**SELECT** a.**name**, **MIN**(total\_amt\_usd) smallest\_order

**FROM** accounts a

**JOIN** orders o

**ON** a.**id** = o.account\_id

**GROUP** **BY** a.**name**

**ORDER** **BY** smallest\_order;

Sort of strange we have a bunch of orders with no dollars. We might want to look into those.

1. Find the number of **sales reps** in each region. Your final table should have two columns - the **region** and the number of **sales\_reps**. Order from fewest reps to most reps.

**SELECT** r.**name**, **COUNT**(\*) num\_reps

**FROM** region r

**JOIN** sales\_reps s

**ON** r.**id** = s.region\_id

**GROUP** **BY** r.**name**

**ORDER** **BY** num\_reps;