@Auther: Gabir N. Yousef
@Date: 26th of Aug 2019
@Quizzes: https://github.com/gabir-yusuf/100DaysOfDataScience/tree/master/2_SQL%20Joins

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.

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?

However, most analysts are working with a database that was already set up with the necessary properties in place. As analysts of data, you don't really need to think too much about data **normalization**. You just need to be able to pull the data from the database, so you can start making insights. This will be our focus in this lesson.

Write Your First JOIN

Below we see an example of a query using a **JOIN** statement. Let's discuss what the different clauses of this query mean.

```
SELECT orders.*
FROM orders
JOIN accounts
ON orders.account_id = accounts.id;
```

As we've learned, the **SELECT** clause indicates which column(s) of data you'd like to see in the output (For Example, orders.* gives us all the columns in orders table in the output). The **FROM** clause indicates the first table from which we're pulling data, and the **JOIN** indicates the second table. The **ON** clause specifies the column on which you'd like to merge the two tables together. Try running this query yourself below.

Additional Information

If we wanted to only pull individual elements from either the **orders** or **accounts** table, we can do this by using the exact same information in the **FROM** and **ON** statements. However, in your **select statement**, you will need to know how to specify tables and columns in the **SELECT** statement:

1. The table name is always **before** the period.
2. The column you want from that table is always **after** the period.

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 only pulls two columns, not all the information in these two tables. Alternatively, the below query pulls all the columns from *both* the **accounts** and **orders** table.

```
SELECT *
FROM orders
JOIN accounts
ON orders.account_id = accounts.id;
```

And the first query you ran pull all the information from *only* the **orders** table:

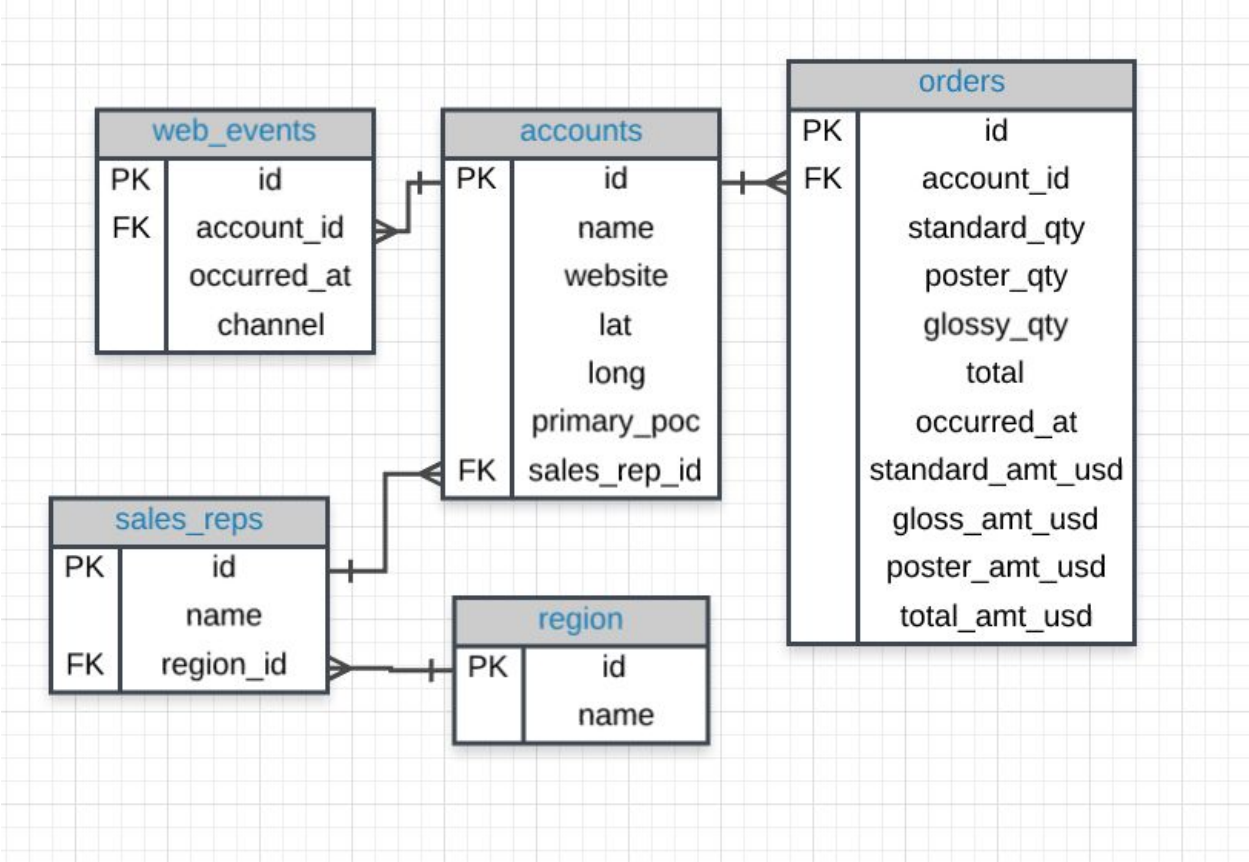
```
SELECT orders.*
FROM orders
JOIN accounts
ON orders.account_id = accounts.id;
```

Joining tables allows you access to each of the tables in the **SELECT** statement through the table name, and the columns will always follow a **.** after the table name.

Entity Relationship Diagrams

From the last lesson, you might remember that an **entity relationship diagram** (ERD) is a common way to view data in a database. It is also a key element to understanding how we can pull data from multiple tables.

It will be beneficial to have an idea of what the ERD looks like for Parch & Posey handy, so I have posted it again below. **You might even print a copy to have with you as you work through the exercises in the remaining content.**



Tables & Columns

In the Parch & Posey database there are 5 tables:

1. **web_events**
2. **accounts**
3. **orders**
4. **sales_reps**
5. **Region**

You will notice some of the columns in the tables have **PK** or **FK** next to the column name, while other columns don't have a label at all.

If you look a little closer, you might notice that the **PK** is associated with the first column in every table. The **PK** here stands for the **primary key**. **A primary key exists in every table, and it is a column that has a unique value for every row.**

If you look at the first few rows of any of the tables in our database, you will notice that this first, **PK**, column is always unique. For this database, it is always called `id`, but that is not true of all databases.

Keys

Primary Key (PK)

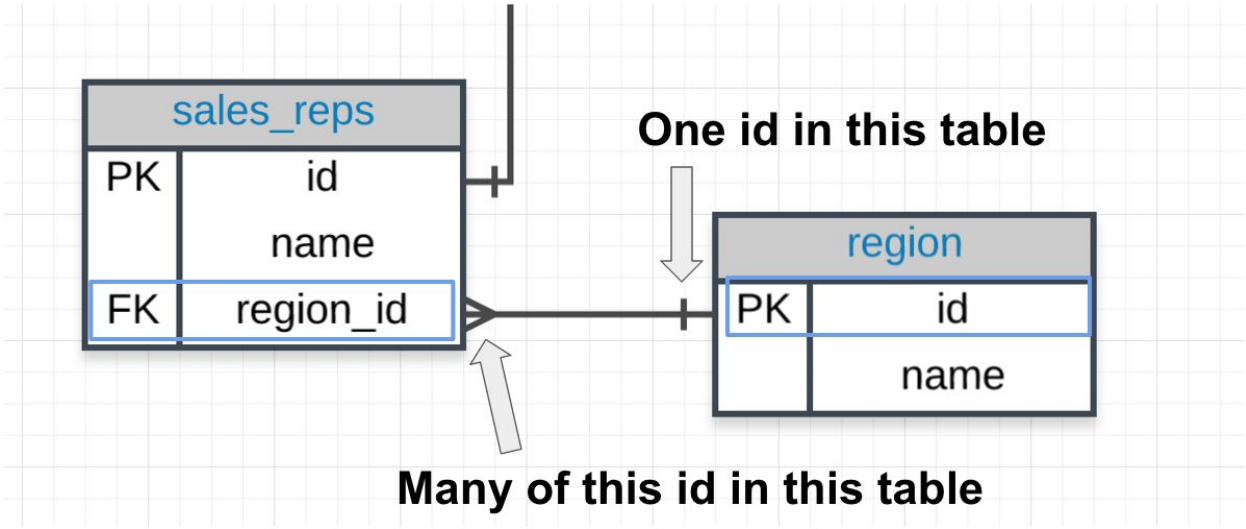
A **primary key** is a unique column in a particular table. This is the first column in each of our tables. Here, those columns are all called **id**, but that doesn't necessarily have to be the name. **It is common that the primary key is the first column in our tables in most databases.**

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.

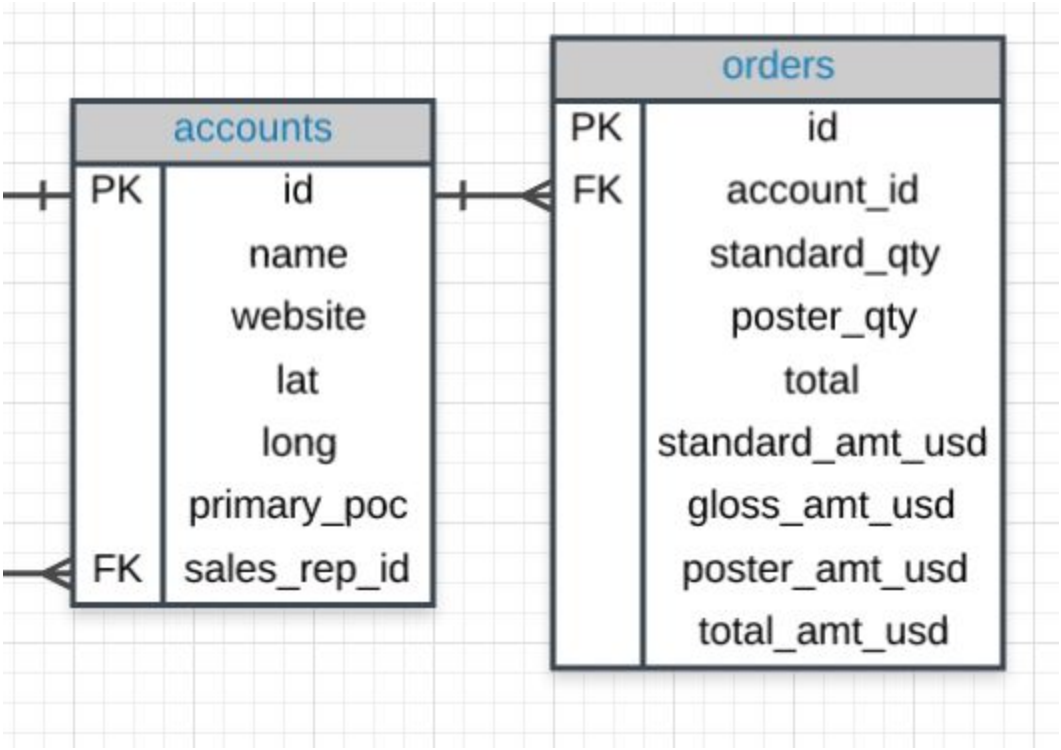
4. While the single line is telling us that the **PK** shows that id appears only once per row in this table.
5.

JOIN Revisited

Let's look back at the first JOIN you wrote.

```
SELECT orders.*
FROM orders
JOIN accounts
ON orders.account_id = accounts.id;
```

Here is the ERD for these two tables:



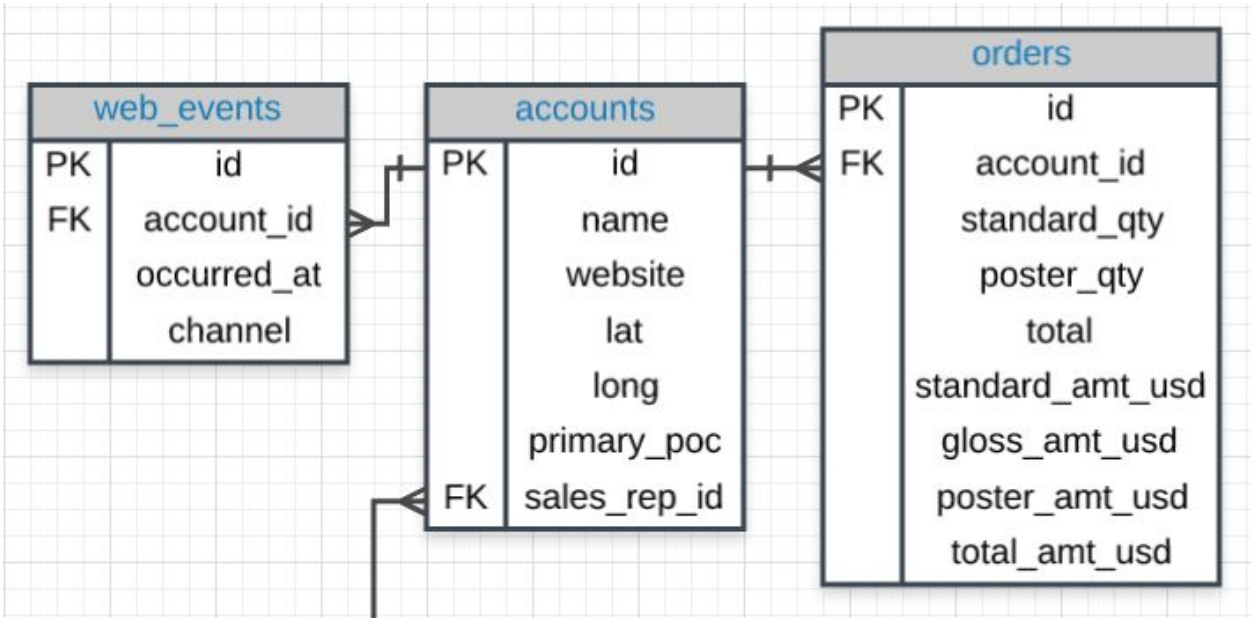
Notice

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
```

Alternatively, we can create a **SELECT** statement that could pull specific columns from any of the three tables. Again, our **JOIN** holds a table, and **ON** is a link for our **PK** to equal the **FK**.

To pull specific columns, the **SELECT** statement will need to specify the table that you are wishing to pull the column from, as well as the column name. We could pull only three columns in the above by changing the select statement to the below, but maintaining the rest of the JOIN information:

```
SELECT web_events.channel, accounts.name, orders.total
```

We could continue this same process to link all of the tables if we wanted. For efficiency reasons, we probably don't want to do this unless we actually need information from all of the tables.

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
```

Frequently, you might also see these statements without the **AS** statement. 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

JOIN Check In

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.

Our new **JOIN**s allow us to pull rows that might only exist in one of the two tables. This will introduce a new data type called **NULL**. This data type will be discussed in detail in the next lesson.

Quick Note

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** we learned about in the previous video.

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.

Again this returns rows that **do not match** one another from the two tables. The use cases for a full outer join are **very rare**.

You can see examples of outer joins at the link here and a description of the rare use cases here. We will not spend time on these given the few instances you might need to use them.

Similar to the above, you might see the language **FULL OUTER JOIN**, which is the same as **OUTER JOIN**.

Notes:

- A **LEFT JOIN** and **RIGHT JOIN** do the same thing if we change the tables that are in the **FROM** and **JOIN** statements.
- A **LEFT JOIN** will **at least** return all the rows that are in an **INNER JOIN**.
- **JOIN** and **INNER JOIN** are the same.
- A **LEFT OUTER JOIN** is the same as **LEFT JOIN**.

LEFT and RIGHT JOIN Solutions

This section is a walkthrough of those final two problems in the previous concept. First, another look at the two tables we are working with:

Country

countryid	countryName
1	India
2	Nepal
3	United States
4	Canada
5	Sri Lanka
6	Brazil

State

stateid	countryid	stateName
1	1	Maharashtra
2	1	Punjab
3	2	Kathmandu
4	3	California
5	3	Texas
6	4	Alberta

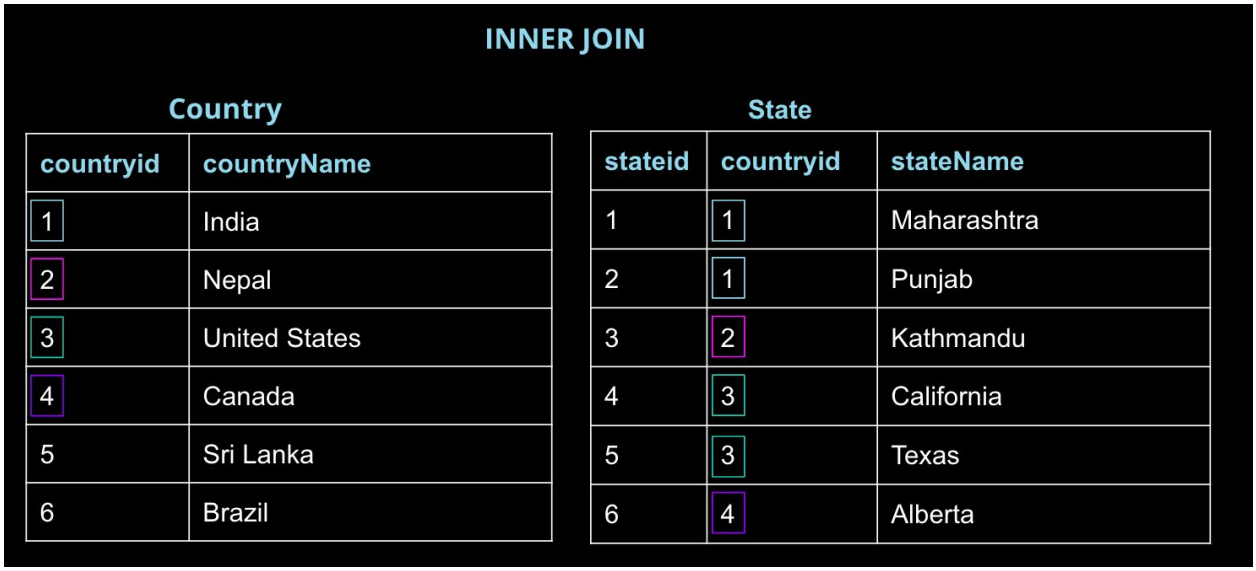
INNER JOIN Question

The questions are aimed to assure you have a conceptual idea of what is happening with **LEFT** and **INNER JOIN**s before you need to use them for more difficult problems.

For an **INNER JOIN** like the one here:

```
SELECT c.countryid, c.countryName, s.stateName
FROM Country c
JOIN State s
ON c.countryid = s.countryid;
```

We are essentially **JOIN**ing the matching **PK**-**FK** links from the two tables, as shown in the below image.



The resulting table will look like:

countryid	countryName	stateName
1	India	Maharashtra

1	India	Punjab
2	Nepal	Kathmandu
3	United States	California
3	United States	Texas
4	Canada	Alberta

LEFT JOIN Question

The questions are aimed to assure you have a conceptual idea of what is happening with **LEFT** and **INNER JOIN**s before you need to use them for more difficult problems.

For a **LEFT JOIN** like the one here:

```
SELECT c.countryid, c.countryName, s.stateName
FROM Country c
LEFT JOIN State s
ON c.countryid = s.countryid;
```

We are essentially **JOIN**ing the matching **PK**-**FK** links from the two tables, as we did before, but we are also pulling all the additional rows from the **Country** table even if they don't have

a match in the **State**table. Therefore, we obtain all the rows of the **INNER JOIN**, but we also get additional rows from the table in the **FROM**.

The resulting table will look like:

countryid	countryName	stateName
1	India	Maharashtra
1	India	Punjab
2	Nepal	Kathmandu
3	United States	California

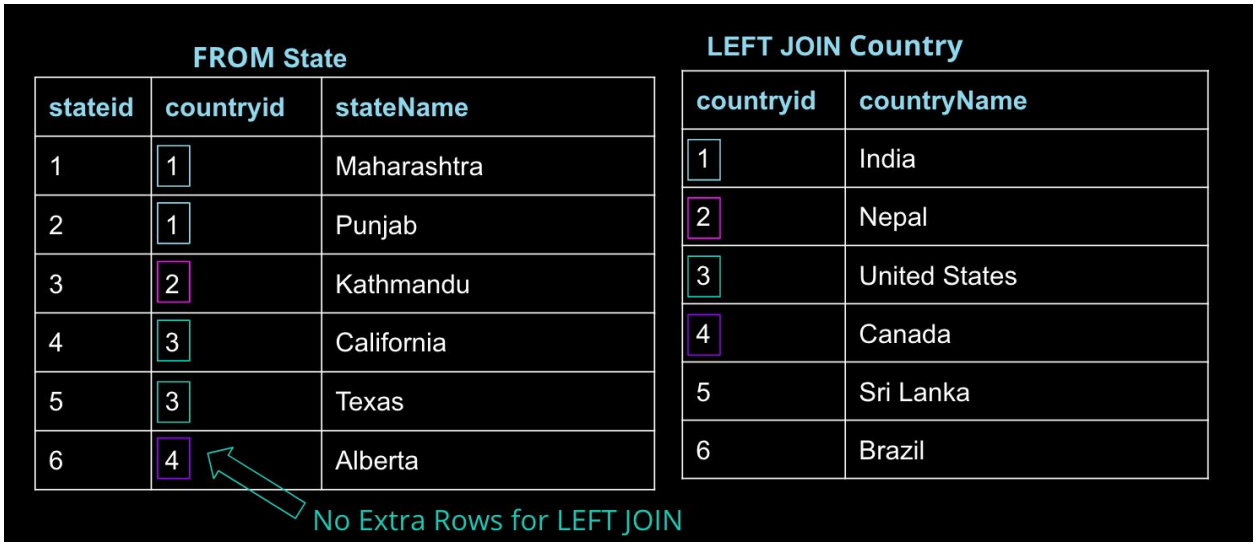
	3	United States	Texas
	4	Canada	Alberta
	5	Sri Lanka	NULL
	6	Brazil	NULL

FINAL LEFT JOIN Note

If we were to flip the tables, we would actually obtain the same exact result as the **JOIN** statement:

```
SELECT c.countryid, c.countryName, s.stateName
FROM State s
LEFT JOIN Country c
ON c.countryid = s.countryid;
```

This is because if **State** is on the **LEFT** table, all of the rows exist in the **RIGHT** table again.



The resulting table will look like:

countryid	countryName	stateName
1	India	Maharashtra
1	India	Punjab
2	Nepal	Kathmandu
3	United States	California
3	United States	Texas

4	Canada	Alberta

Note:

`If you have two or more columns in your SELECT that have the same name after the table name such as accounts.name and sales_reps.name you will need to alias them. Otherwise it will only show one of the columns. You can alias them like accounts.name AS AcountName, sales_rep.name AS SalesRepName`

Recap

Primary and Foreign Keys

You learned a key element for **JOIN**ing tables in a database has to do with primary and foreign keys:

- **primary keys** - are unique for every row in a table. These are generally the first column in our database (like you saw with the **id** column for every table in the Parch & Posey database).
- **foreign keys** - are the **primary key** appearing in another table, which allows the rows to be non-unique.

Choosing the set up of data in our database is very important, but not usually the job of a data analyst. This process is known as **Database Normalization**.

JOINs

In this lesson, you learned how to combine data from multiple tables using **JOIN**s. The three **JOIN**statements you are most likely to use are:

1. **JOIN** - an **INNER JOIN** that only pulls data that exists in both tables.
2. **LEFT JOIN** - pulls all the data that exists in both tables, as well as all of the rows from the table in the **FROM** even if they do not exist in the **JOIN** statement.

3. **RIGHT JOIN** - pulls all the data that exists in both tables, as well as all of the rows from the table in the **JOIN** even if they do not exist in the **FROM** statement.

There are a few more advanced **JOIN**s that we did not cover here, and they are used in very specific use cases. UNION and UNION ALL, CROSS JOIN, and the tricky SELF JOIN. These are more advanced than this course will cover, but it is useful to be aware that they exist, as they are useful in special cases.

Alias

You learned that you can alias tables and columns using **AS** or not using it. This allows you to be more efficient in the number of characters you need to write, while at the same time you can assure that your column headings are informative of the data in your table.