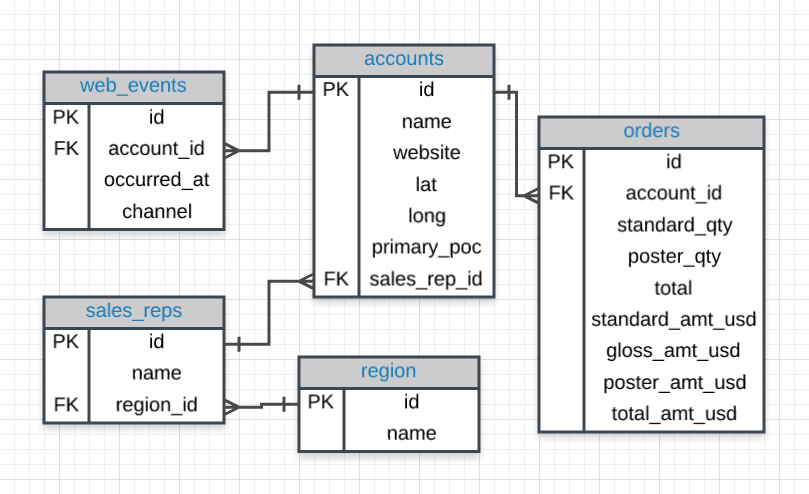
**Entity Relationship Diagrams**

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.**



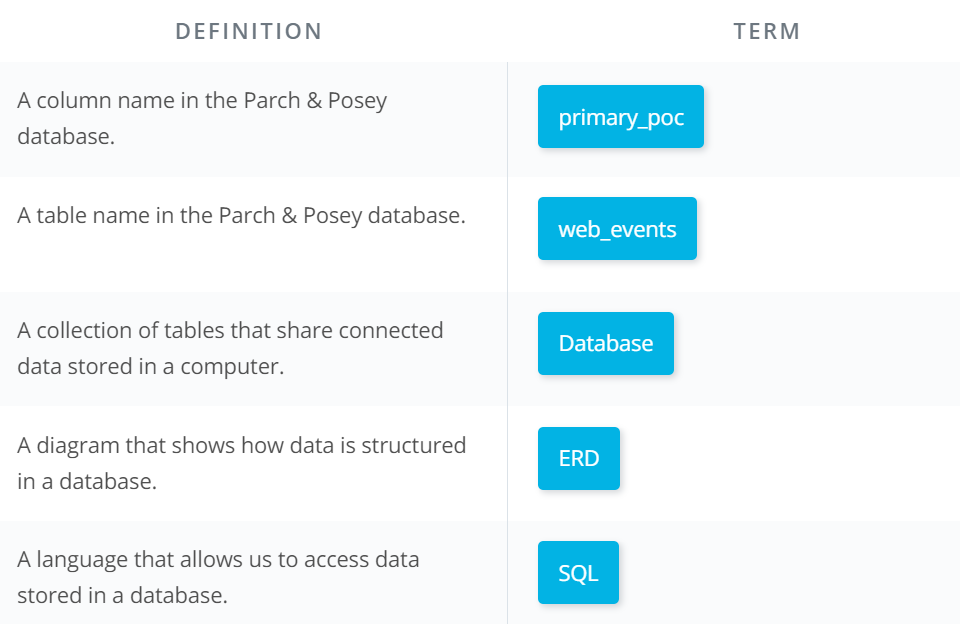
**What to Notice**

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

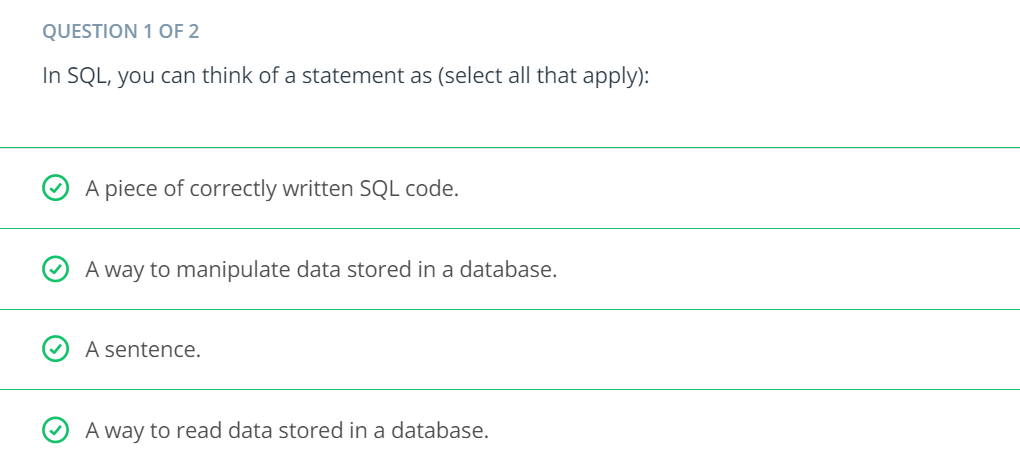
There are many different types of SQL databases designed for different purposes.

[**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."*