

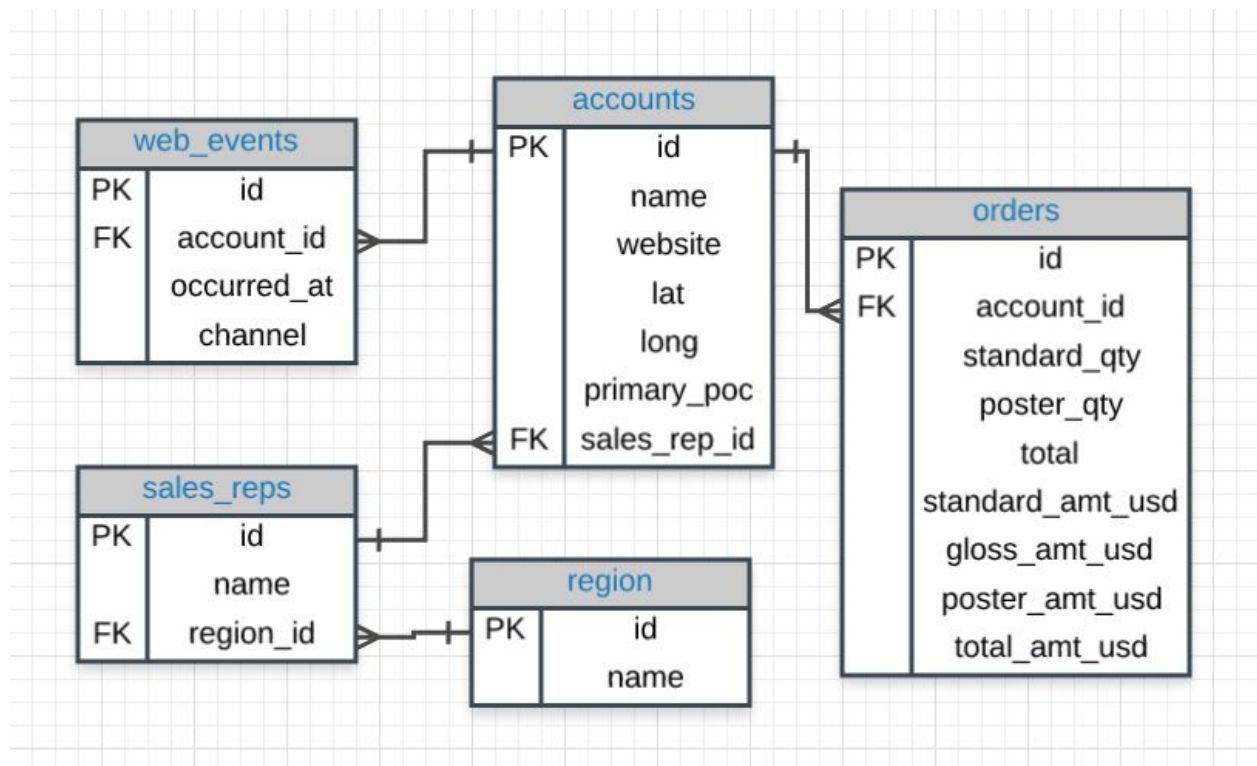
# Entity Relationship Diagrams

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.

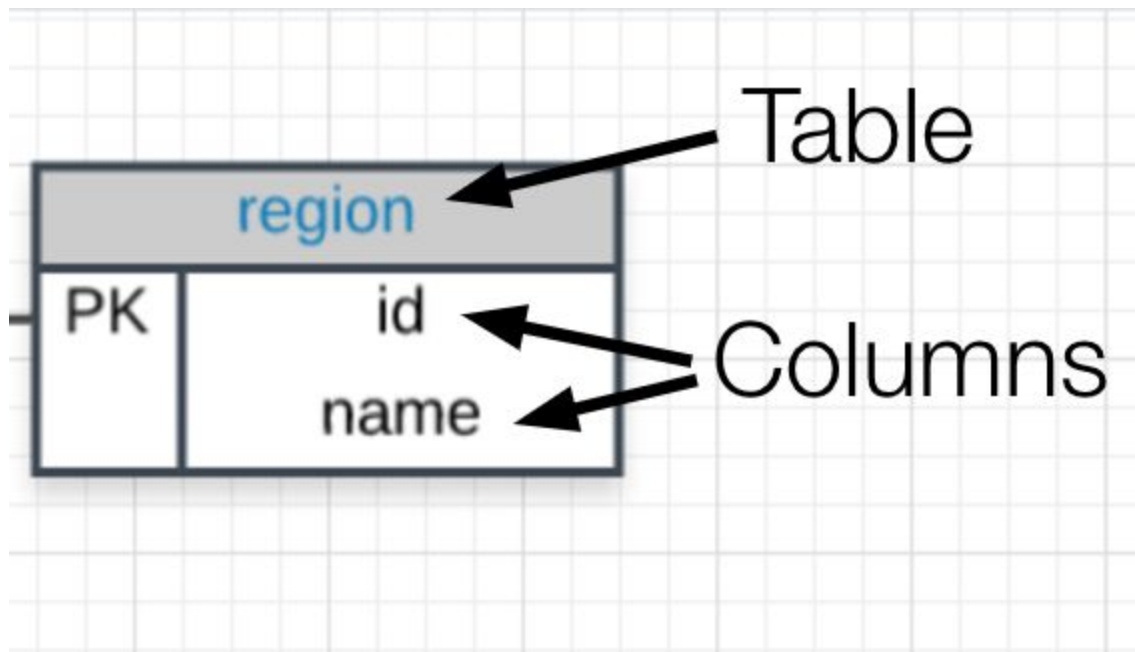


## What to Notice

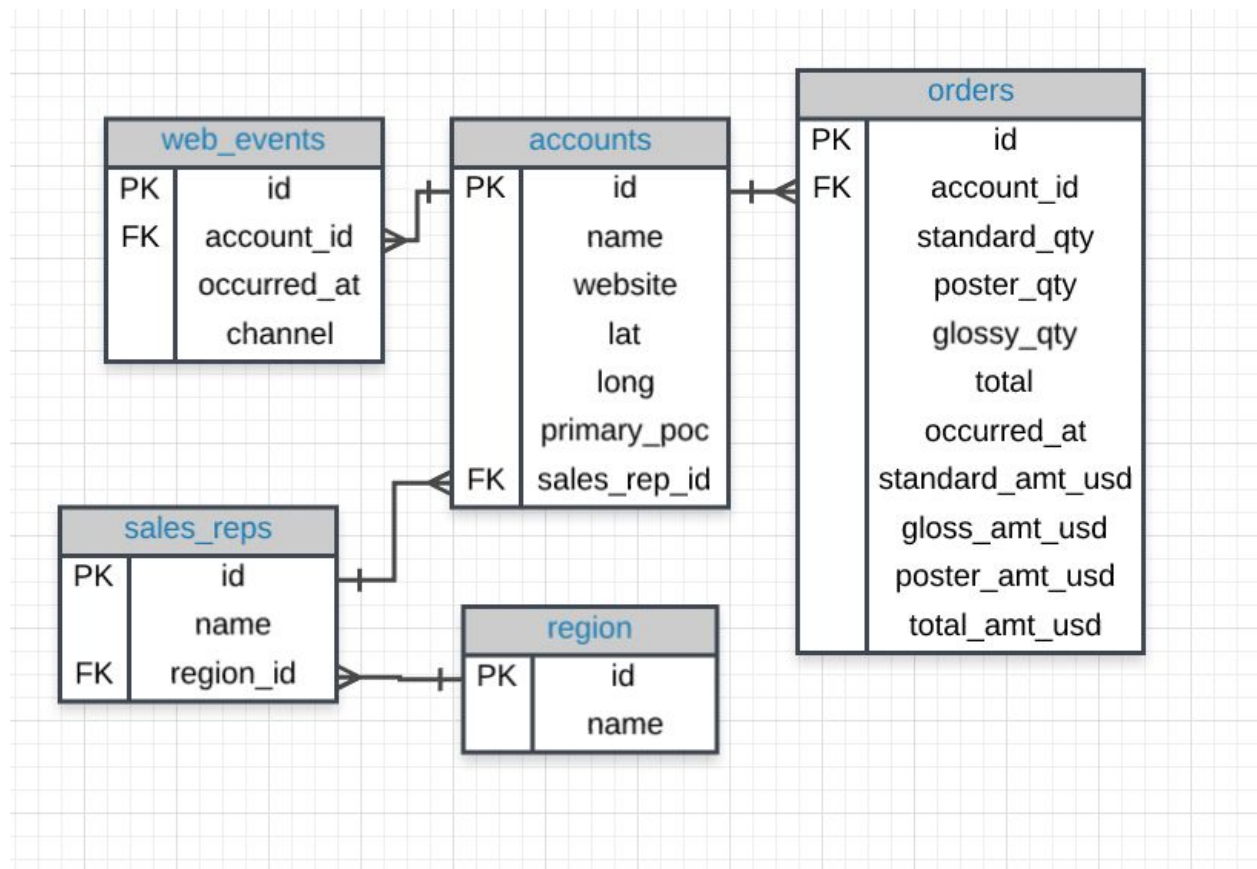
In the Parch & Posey database there are five tables (essentially 5 spreadsheets):

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.



The "crow's foot" that connects the tables together shows us how the columns in one table relate to the columns in another table. In this first lesson, you will be learning the basics of how to work with SQL to interact with a single table. In the next lesson, you will learn more about why these connections are so important for working with SQL and relational databases.



Note: glossy\_qty is incorrect, it is actually gloss\_qty in the database

## Introduction

Throughout the next three lessons, you will be learning how to write Structured Query Language (SQL) to interact with a database here in the classroom. You will not need to download any software, and you will still be able to test your skills!

SQL is an extremely in demand skill. [Tons of jobs use SQL](#), and in the next lessons you will be learning how to utilize SQL to analyze data and answer business questions.

## Project

The skills you learn in the classroom are directly extendable to writing SQL in other environments outside this classroom. For the project at the end of these lessons, you will download a program that will allow you to write code on your local machine. You will then analyze and answer business questions using data associated with a music store by querying their database.

## Lesson Outline

There are three lessons in this Nanodegree aimed at helping you understand how to write SQL queries. If you choose to take the [Predictive Analytics Nanodegree](#) or the [Data Analyst Nanodegree](#) programs, these three lessons will also be a part of these programs. However, there is also an additional lesson on Advanced SQL also taught by Derek!

The three lessons in this course aim at the following components of SQL:

- **SQL Basics** - Here you will get your first taste at how SQL works, and learn the basics of the SQL language. You will learn how to write code to interact with tables similar to the ones we analyzed in Excel earlier. Specifically, you will learn a little about databases, the basic syntax of SQL, and you will write your first queries!
- **SQL Joins** - In this lesson, you will learn the real power of SQL. You will learn about Entity Relationship Diagrams (ERDs), and how to join multiple tables together from a relational database. The power to join tables is what really moved companies to adopt this approach to holding data.

- **SQL Aggregations** - In this final lesson, you will learn some more advanced features of SQL. You will gain the ability to summarize data from multiple tables in a database.

At the end of these three lessons, you will be ready to tackle the project. The project aims to assure you have mastered these three topics, but you will also see some of the more advanced queries that were not covered in this course. These are just meant to introduce you to the advanced material, but don't feel discouraged if you didn't get these - they were beyond the scope of the class, and they are not required to pass the project!

## Introduction

Before we dive into writing Structured Query Language (SQL) queries, let's take a look at what makes SQL and the databases that utilize SQL so popular.

I think it is an important distinction to say that SQL is a language. Hence, the last word of SQL 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. For this class, you can think of a database as a bunch of excel spreadsheets all sitting in one place. Not all databases are a bunch of excel spreadsheets sitting in one place, but it is a reasonable idea for this class.

# Why Do Data Analysts Use SQL?

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.

You will experience these advantages first hand, as we learn to write SQL to interact with data.

I realize you might be getting a little nervous or anxious to start writing code. This might even be the first time you have written in any sort of programming language. I assure you, we will work through examples to help assure you feel supported the whole time to take on this new challenge!

## 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](#). Udacity has a full course on MongoDB that you can take for free [here](#), but these will not be a focus of this program.

NoSQL is not a focus of analyzing data in this Nanodegree program, but you might see it referenced outside this course!

## Why Do Businesses Choose SQL?

### 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. In this course we will use [Postgres](#) within the classroom, which 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.

## Small Differences

Each of these SQL databases may have subtle differences in syntax and available functions -- for example, MySQL doesn't have some of the functions for modifying dates as Postgres. Most of what you see with Postgres will be directly applicable to

using SQL in other frameworks and database environments. For the differences that do exist, you should check the documentation. Most SQL environments have great documentation online that you can easily access with a quick Google search.

The article [here](#) compares three of the most common types of SQL: SQLite, PostgreSQL, and MySQL.

You will use PostgreSQL for the lessons in this course, but you are not required to download it to your machine. We provide SQL workspaces in the classroom throughout the lessons. You may download PostgreSQL [here](#) if you'd like though.

So with that, let's jump in!

#### QUIZ QUESTION

Check all of the below that are true about your learning experience here in the classroom.

- The code you write in the classroom is exactly what you would write to analyze data in PostgreSQL.
- The code you write in the classroom is exactly what you would write to analyze data in MySQL.
- The code you write in the classroom will allow you to easily pick up any SQL programming including Microsoft SQL Server, Oracle, or SQLite.
- When you leave the classroom, you will have to re-learn how to use SQL, as it works differently outside the classroom.

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.

You will use this SQL `SELECT` statement in every query in this course, but you will be learning a few additional statements and operators that can be used along with them to ask more advanced questions of your data.

## Your First SQL Statement

To get started, try using the right panel in the SQL workspace above to type in and evaluate the query (shown again below) that you saw in the previous video. You can begin typing right next to the number 1.

```
SELECT *  
FROM orders;
```

Once you have typed in your SQL code, you can click on the blue **EVALUATE** button to run the query. This may take a moment. An alternative to clicking EVALUATE is to use **control + Enter** to execute your query. If you get an error, it will sometimes cover the EVALUATE button, so this second option is very nice!

You will notice that your instructor Derek uses "demo" tables, like `FROM demo.orders` (and he will continue to do this in future lessons), but you should write your queries using the table names exactly as shown in the Schema on the left, with "demo" removed, like `FROM orders`.

## **SELECT and FROM in Every SQL Query**

Every query will have at least a **SELECT** and **FROM** statement. The **SELECT** statement is where you put the **columns** for which you would like to show the data. The **FROM** statement is where you put the **tables** from which you would like to pull data.

## **Your Turn**

Try writing your own query to select only the `id`, `account_id`, and `occurred_at` columns for all orders in the **orders** table.

## **Checking the Output of your Query**

If you see an error message after executing your query, no problem! Programmers get errors all the time! Mistakes are opportunities to learn. :)

Just take a look at what the error says, check the syntax of your query compared to your notes or what was shown in the lesson video or text, make revisions, and try it again. See what you can learn from your mistake, and make a note about it.

If you see a message saying "Success!" this means you had no syntax errors and your query executed well. It does not necessarily mean that you got your desired results though! It is still up to you to examine your output and see if what you got was what you wanted.

You can also check on the next page after each quiz, to see how your query compares to the solution that we provide. We highly recommend that you try writing your query first, before looking at any solution. :)

### **Downloading the Schema Tables**

(In some versions of this lesson, you might be able to download the Parch & Posey database to your computer from the Resources tab in the left sidebar of your classroom, but this is not necessary for you to do at all.)

## **Solution to Previous Concept**

You may use the workspace below to try out this solution if you like.

```
SELECT id, account_id, occurred_at  
  
FROM orders;
```

In case you want to test any of the ideas below, I have embedded a SQL workspace environment at the bottom of this page.

# Formatting Your Queries

## Using Upper and Lower Case in SQL

SQL queries can be run successfully whether characters are written in upper- or lower-case. In other words, SQL queries are not case-sensitive. The following query:

```
SELECT account_id  
FROM orders
```

is the same as:

```
select account_id  
from orders
```

which is also the same as:

```
SeLeCt AcCoUnT_id  
FrOm oRdErS
```

**However**, you may have noticed that we have been capitalizing SELECT and FROM, while we leave table and column names in lower case. This is because even though SQL is case-insensitive, **it is common and best practice to capitalize all SQL commands, like SELECT and FROM, and keep everything else in your query lower case.**

Capitalizing command words makes queries easier to read, which will matter more as you write more complex queries. For now, it is just a good habit to start getting into, to make your SQL queries more readable.

One other note: The text data stored in SQL tables can be either upper or lower case, and SQL *is* case-sensitive in regard to this text data.

## Avoid Spaces in Table and Variable Names

It is common to use underscores and avoid spaces in column names. It is a bit annoying to work with spaces in SQL. In Postgres if you have spaces in column or table names, you need to refer to these columns/tables with double quotes around them (Ex: FROM "Table Name" as opposed to FROM table\_name). In other environments, you might see this as square brackets instead (Ex: FROM [Table Name]).

## Use White Space in Queries

SQL queries ignore spaces, so you can add as many spaces and blank lines between code as you want, and the queries are the same. This query

```
SELECT account_id FROM orders
```

is equivalent to this query:

```
SELECT account_id
FROM orders
```

and this query (but please don't ever write queries like this):

```
SELECT      account_id
```

```
FROM orders
```

## Semicolons

Depending on your SQL environment, your query may need a semicolon at the end to execute. Other environments are more flexible in terms of this being a "requirement." It is considered best practice to put a semicolon at the end of each statement, which also allows you to run multiple queries at once if your environment allows this.

Best practice:

```
SELECT account_id  
FROM orders;
```

Since our environment here doesn't require it, you will see solutions written without the semicolon:

```
SELECT account_id  
FROM orders
```

**Phew!!! That was a lot of rules. Let's just write some queries. You will make mistakes, but that is part of the learning process!**

We have already seen the **SELECT** (to choose columns) and **FROM** (to choose tables) statements. The **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;
```

We could also change the number of rows by changing the 10 to any other number of rows.

## Can You Use LIMIT?

1. Try using LIMIT yourself below by writing a query that displays all the data in the `occurred_at`, `account_id`, and `channel` columns of the `web_events` table, and limits the output to only the first 15 rows.

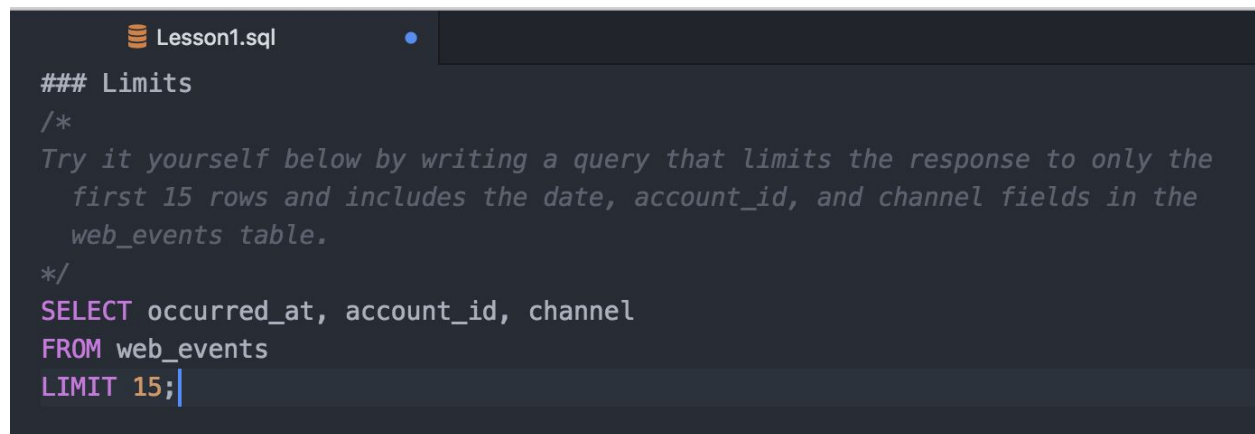
## Using a Separate Text Editor to Write SQL Queries and Save Your Notes

You might want to use a separate text editor to write SQL queries, and make notes on what they are used for. I copy and paste my SQL queries back and forth between the classroom and the Atom text editor, which you can download free [here](#) if you'd like.

You can use any method or text editor for writing your queries and keeping your own notes. Notepad and Word are other options. The screenshot below shows what my file looks like in Atom.

I save my SQL query files in Atom with a `.sql` extension to get highlighting support with SQL syntax.

All of this is optional for you though. You can do all of your work in the Udacity classroom if you like, but using a separate text editor is a way to save your notes and look back at them later.



```
Lesson1.sql
### Limits
/*
Try it yourself below by writing a query that limits the response to only the
first 15 rows and includes the date, account_id, and channel fields in the
web_events table.
*/
SELECT occurred_at, account_id, channel
FROM web_events
LIMIT 15;
```

## Course Flow

A few notes about the course flow from this point. As you are introduced to new SQL functionality, we will prompt you to practice writing queries on your own.

You will usually watch Derek answer a business question with a new SQL command. Then we'll ask you a few questions to practice use the same functionality. For each question, a solution is provided on the following page, so that you can check your answer.

You can usually see this structure in the concept names: **Video** shows Derek performing the query, **Quiz** provides questions and a workspace for you to practice, and **Solutions** provides the solutions for the quiz questions.

## Solution to Previous LIMIT Question

```
SELECT occurred_at, account_id, channel
```

```
FROM web_events
```

```
LIMIT 15;
```

The **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.**

In other words, when you use ORDER BY in a SQL query, your output will be sorted that way, but then the next query you run will encounter the unsorted data again. It's important to keep in mind that this is different than using common spreadsheet software, where

sorting the spreadsheet by column actually alters the data in that sheet until you undo or change that sorting. This highlights the meaning and function of a SQL "query."

The **ORDER BY** statement always comes in a query after the **SELECT** and **FROM** statements, but before the **LIMIT** statement. If you are using the **LIMIT** statement, it will always appear last. As you learn additional commands, the order of these statements will matter more.

## 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.

[NEXT](#)

## Practice

Let's get some practice using **ORDER BY**:

1. Write a query to return the 10 earliest orders in the **orders** table. Include the `id`, `occurred_at`, and `total_amt_usd`.
2. Write a query to return the top 5 **orders** in terms of largest `total_amt_usd`. Include the `id`, `account_id`, and `total_amt_usd`.

3. Write a query to return the lowest 20 **orders** in terms of smallest

`total_amt_usd`. Include the `id`, `account_id`, and `total_amt_usd`.

Remember, **just because a query successfully runs, does not mean you have the correct results**. To see if your query worked like you wanted it to, you need to examine your output to see if it satisfies the problem or the question. You can also compare your query and results with the solution on the following page (concept).

## Solutions to previous ORDER BY questions

Write a query to return the 10 earliest orders in the **orders** table. Include the `id`, `occurred_at`, and `total_amt_usd`.

```
SELECT id, occurred_at, total_amt_usd
```

```
FROM orders
```

```
ORDER BY occurred_at
```

```
LIMIT 10;
```

1.

Write a query to return the top 5 **orders** in terms of largest `total_amt_usd`. Include the `id`, `account_id`, and `total_amt_usd`.

```
SELECT id, account_id, total_amt_usd
```

```
FROM orders
```

```
ORDER BY total_amt_usd DESC
```

```
LIMIT 5;
```

2.

Write a query to return the lowest 20 **orders** in terms of smallest `total_amt_usd`. Include the `id`, `account_id`, and `total_amt_usd`.

```
SELECT id, account_id, total_amt_usd
```

```
FROM orders
```

```
ORDER BY total_amt_usd
```

3. `LIMIT 20;`

Here, we saw that we can **ORDER BY** more than one column at a time. When you provide a list of columns in an **ORDER BY** command, the sorting occurs using the leftmost column in your list first, then the next column from the left, and so on. We still have the ability to flip the way we order using **DESC**.

## Questions

1. Write a query that displays the order ID, account ID, and total dollar amount for all the orders, sorted first by the account ID (in ascending order), and then by the total dollar amount (in descending order).
2. Now write a query that again displays order ID, account ID, and total dollar amount for each order, but this time sorted first by total dollar amount (in descending order), and then by account ID (in ascending order).

3. Compare the results of these two queries above. How are the results different when you switch the column you sort on first?

## Solutions to previous ORDER BY Questions

Write a query that displays the order ID, account ID, and total dollar amount for all the orders, sorted first by the account ID (in ascending order), and then by the total dollar amount (in descending order).

```
SELECT id, account_id, total_amt_usd  
  
FROM orders  
  
ORDER BY account_id, total_amt_usd DESC;
```

1.

Now write a query that again displays order ID, account ID, and total dollar amount for each order, but this time sorted first by total dollar amount (in descending order), and then by account ID (in ascending order).

```
SELECT id, account_id, total_amt_usd  
  
FROM orders  
  
ORDER BY total_amt_usd DESC, account_id;
```

2.

3. Compare the results of these two queries above. How are the results different when you switch the column you sort on first?

**In query #1, all of the orders for each account ID are grouped together, and then within each of those groupings, the orders appear from the greatest order amount to the least. In query #2, since you sorted by the total dollar**

amount first, the orders appear from greatest to least regardless of which account ID they were from. Then they are sorted by account ID next. (The secondary sorting by account ID is difficult to see here, since only if there were two orders with equal total dollar amounts would there need to be any sorting by account ID.)

Using the **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.

This video above shows how this can be used, and in the upcoming concepts, you will learn some common operators that are useful with the **WHERE** statement.

Common symbols used in **WHERE** statements include:

1.  $>$  (greater than)
2.  $<$  (less than)
3.  $>=$  (greater than or equal to)
4.  $<=$  (less than or equal to)
5.  $=$  (equal to)
6.  $\neq$  (not equal to)

## Questions



Write a query that:

1. Pulls the first 5 rows and all columns from the **orders** table that have a dollar amount of `gross_amt_usd` greater than or equal to 1000.
2. Pulls the first 10 rows and all columns from the **orders** table that have a `total_amt_usd` less than 500.

## Solution from previous WHERE Questions

```
SELECT *
```

```
FROM orders
```

```
WHERE gross_amt_usd >= 1000
```

```
LIMIT 5;
```

1.

```
SELECT *
```

```
FROM orders
```

```
WHERE total_amt_usd < 500
```

```
LIMIT 10;
```

2.

You will notice when using these **WHERE** statements, we do not need to **ORDER BY** unless we want to actually order our data. Our condition will work without having to do any sorting of the data.

The **WHERE** statement can also be used with non-numeric data. We can use the `=` and `!=` operators here. You need to be sure to use single quotes (just be careful if you have quotes in the original text) with the text data, not double quotes.

Commonly when we are using **WHERE** with non-numeric data fields, we use the **LIKE**, **NOT**, or **IN** operators. We will see those before the end of this lesson!

[NEXT](#)

## Practice Question Using WHERE with Non-Numeric Data

1. Filter the accounts table to include the company `name`, `website`, and the primary point of contact (`primary_poc`) just for the Exxon Mobil company in the **accounts** table.

## Solution from WHERE with Non-Numeric Data

```
SELECT name, website, primary_poc  
  
FROM accounts  
  
WHERE name = 'Exxon Mobil';
```

1.

**Note:** If you received an error message when executing your query, remember that SQL requires single-quotes, not double-quotes, around text values like 'Exxon Mobil.'

## 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.

If you are deriving the new column from existing columns using a mathematical expression, then these familiar mathematical operators will be useful:

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

Consider this example:

```
SELECT id, (standard_amt_usd/total_amt_usd)*100 AS std_percent, total_amt_usd
FROM orders

LIMIT 10;
```

Here we divide the standard paper dollar amount by the total order amount to find the standard paper percent for the order, and use the **AS** keyword to name this new column "std\_percent." You can run this query on the next page if you'd like, to see the output.

## Order of Operations

Remember PEMDAS from math class to help remember the order of operations? If not, check out this [link](#) as a reminder. The same order of operations applies when using arithmetic operators in SQL.

The following two statements have very different end results:

1. **Standard\_qty / standard\_qty + gloss\_qty + poster\_qty**
2. **standard\_qty / (standard\_qty + gloss\_qty + poster\_qty)**

It is likely that you mean to do the calculation as written in statement number 2!

## Questions using Arithmetic Operations

Using the **orders** table:

1. Create a column that divides the `standard_amt_usd` by the `standard_qty` to find the unit price for standard paper for each order. Limit the results to the first 10 orders, and include the `id` and `account_id` fields.
2. Write a query that finds the percentage of revenue that comes from poster paper for each order. You will need to use only the columns that end with `_usd`. (Try to do this without using the `total` column.) Display the `id` and `account_id` fields also. **NOTE - you will receive an error with the correct solution to this question. This occurs because at least one of the values in the data creates a division by zero in your formula. You will learn later in the course how to fully handle this issue. For now, you can just limit your calculations to the first 10 orders, as we did in question #1, and you'll avoid that set of data that causes the problem.**

Notice, the above operators combine information across columns for the same row. If you want to combine values of a particular column, across multiple rows, we will do this with aggregations. We will get to that before the end of the course!

## Solutions to Arithmetic Operator Questions

```
SELECT id, account_id, standard_amt_usd/standard_qty AS unit_price  
  
FROM orders  
  
LIMIT 10;
```

1.

```
SELECT id, account_id,  
  
       poster_amt_usd / (standard_amt_usd + gloss_amt_usd + poster_amt_usd) AS  
post_per  
  
FROM orders
```

2. `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 allow you to combine operations where at least one of the combined conditions must be true.

---

Don't worry if this doesn't make total sense right now. You will get practice with each in the next sections.

[NEXT](#)

The **LIKE** operator 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. Remember you will need to use single quotes for the text you pass to the **LIKE** operator, because of this lower and uppercase letters are not the same within the string. Searching for **'T'** is not the same as searching for **'t'**. In other SQL environments (outside the classroom), you can use either single or double quotes.

Hopefully you are starting to get more comfortable with SQL, as we are starting to move toward operations that have more applications, but this also means we can't show you every use case. Hopefully, you can start to think about how you might use these types

of applications to identify phone numbers from a certain region, or an individual where you can't quite remember the full name.

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. In the upcoming concepts, you will see the **OR** operator that would also allow us to perform these tasks, but the **IN** operator is a cleaner way to write these queries.

### Expert Tip

In most SQL environments, although not in our Udacity's classroom, you can use single or double quotation marks - and you may NEED to use double quotation marks if you have an apostrophe within the text you are attempting to pull.

In our Udacity SQL workspaces, note you can include an apostrophe by putting two single quotes together. For example, Macy's in our workspace would be 'Macy's'.

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.

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. 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**, 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. In the previous video, we probably should have used **BETWEEN**.

Instead of writing :

```
WHERE column >= 6 AND column <= 10
```

we can instead write, equivalently:

```
WHERE column BETWEEN 6 AND 10
```

## Questions using AND and BETWEEN operators

1. Write a query that returns all the **orders** where the `standard_qty` is over 1000, the `poster_qty` is 0, and the `gloss_qty` is 0.
2. Using the **accounts** table, find all the companies whose names do not start with 'C' and end with 's'.
3. 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.

4. Use the **web\_events** table to find all information regarding individuals who were contacted via the **organic** or **adwords** channels, and started their account at any point in 2016, sorted from newest to oldest.

## Solutions to AND and BETWEEN Questions

Write a query that returns all the orders where the **standard\_qty** is over 1000, the **poster\_qty** is 0, and the **gloss\_qty** is 0.

```
SELECT *  
FROM orders  
WHERE standard_qty > 1000 AND poster_qty = 0 AND gloss_qty = 0;
```

1.

Using the **accounts** table, find all the companies whose names do not start with 'C' and end with 's'.

```
SELECT name  
FROM accounts  
WHERE name NOT LIKE 'C%' AND name LIKE '%s';
```

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;
```

3.

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."

You will notice that using **BETWEEN** is tricky for dates! While **BETWEEN** is generally inclusive of endpoints, it assumes the time is at 00:00:00 (i.e. midnight) for dates. This is the reason why we set the right-side endpoint of the period at '2017-01-01'.

```
SELECT *  
FROM web_events  
WHERE channel IN ('organic', 'adwords') AND occurred_at BETWEEN '2016-01-01'  
AND '2017-01-01'  
4. ORDER BY occurred_at DESC;
```

Similar to the **AND** operator, 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.

When combining multiple of these operations, we frequently might need to use parentheses to assure that logic we want to perform is being executed correctly. The video below shows an example of one of these situations.

## Recap

### Commands

You have already learned a lot about writing code in SQL! Let's take a moment to recap all that we have covered before moving on:

Statement	How to Use It	Other Details
SELECT	SELECT <b>Col1</b> , <b>Col2</b> , ...	Provide the columns you want
FROM	FROM <b>Table</b>	Provide the table where the columns exist
LIMIT	LIMIT <b>10</b>	Limits based number of rows returned
ORDER BY	ORDER BY <b>Col</b>	Orders table based on the column. Used with <b>DESC</b> .
WHERE	WHERE <b>Col &gt; 5</b>	A conditional statement to filter your results
LIKE	WHERE <b>Col LIKE '%me%'</b>	Only pulls rows where column has 'me' within the text
IN	WHERE <b>Col IN ('Y', 'N')</b>	A filter for only rows with column of 'Y' or 'N'

NOT	WHERE Col NOT IN ('Y', 'N')	NOT is frequently used with LIKE and IN
AND	WHERE Col1 > 5 AND Col2 < 3	Filter rows where two or more conditions must be true
OR	WHERE Col1 > 5 OR Col2 < 3	Filter rows where at least one condition must be true
BETWEEN	WHERE Col BETWEEN 3 AND 5	Often easier syntax than using an AND

## Other Tips

Though SQL is **not case sensitive** (it doesn't care if you write your statements as all uppercase or lowercase), we discussed some best practices. **The order of the key words does matter!** Using what you know so far, you will want to write your statements as:

```
SELECT col1, col2
FROM table1
WHERE col3 > 5 AND col4 LIKE '%OS%'
ORDER BY col5
LIMIT 10;
```

Notice, you can retrieve different columns than those being used in the **ORDER BY** and **WHERE** statements. Assuming all of these column names existed in this way (col1, col2, col3, col4, col5) within a table called table1, this query would run just fine.

## Looking Ahead

In the next lesson, you will be learning about **JOINS**. This is the real secret (well not really a secret) behind the success of SQL as a language. **JOINS** allow us to combine multiple tables together. All of the operations we learned here will still be important moving forward, but we will be able to answer much more complex questions by combining information from multiple tables! You have already mastered so much - potentially writing your first code ever, but it is about to get so much better!