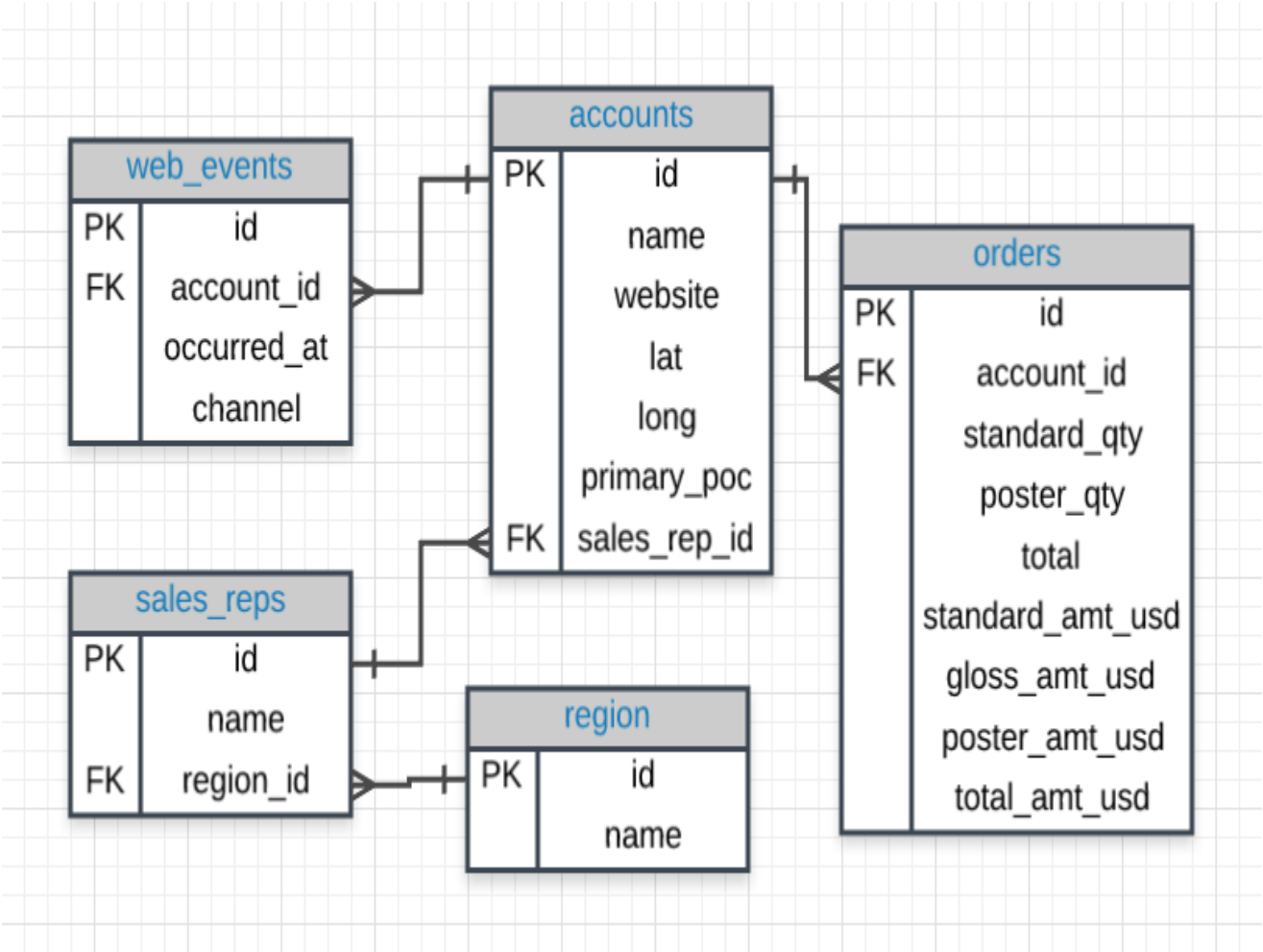
Entity Relationship Diagrams

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.

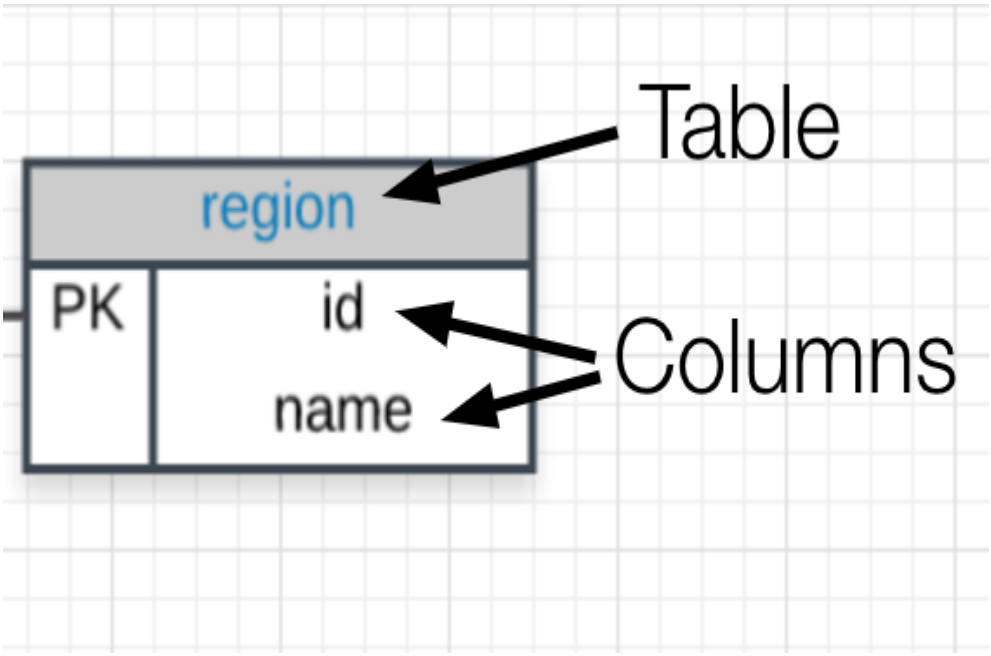


What to Notice

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.

Map of SQL Content

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 **Business Analyst 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 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.

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 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.

How Databases Store Data

A few key points about data stored in SQL databases:

1. **Data in databases is stored in tables that can be thought of just like Excel spreadsheets.** For the most part, you can think of a database as a bunch of Excel spreadsheets. Each spreadsheet has rows and columns. Where each row holds data on a transaction, a person, a company, etc., while each column holds data pertaining to a particular aspect of one of the rows you care about like a name, location, a unique id, etc.

2. **All the data in the same column must match in terms of data type.**
   An entire column is considered quantitative, discrete, or as some sort of string. This means if you have one row with a string in a particular column, the entire column might change to a text data type. **This can be very bad if you want to do math with this column!**

3. **Consistent column types are one of the main reasons working with databases is fast.**
   Often databases hold **a LOT** of data. So, knowing that the columns are all of the same type of data means that obtaining data from a database can still be fast.

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. Though you will use PostgreSQL in the classroom, you will utilize SQLite for the project. Again, once you have learned how to write SQL in one environment, the skills are mostly transferable.

PostgreSQL

Types Of Statements

The key to SQL is understanding **statements**. A few statements include:

1. **CREATE TABLE** is a statement that creates a new table in a database.
2. **DROP TABLE** is a statement that removes a table in a database.
3. **SELECT** allows you to read data and display it. This is called a **query**.
   The **SELECT** statement is the common statement used by analysts, and you will be learning all about them throughout this course!

SQL statements are code that can read and manipulate data. Basic syntax reminders: SQL isn't case sensitive - meaning you can write upper and lower case anywhere in the code. Additionally, you can end SQL statements with a semicolon, but some SQL environments don't require a semicolon at the end.

SELECT statement, which is called a query. The DROP and CREATE statements actually change the data in the database. In most companies, analysts are not given permission to use these types of statements. This a good thing - ACTUALLY changing the data in the database is a powerful thing. This is generally reserved for database administrators exclusively.

SELECT & FROM Statements

Here you were introduced to two statements that will be used in every query you write

1. **SELECT** is where you tell the query what columns you want back.

2. **FROM** is where you tell the query what table you are querying from. Notice the columns need to exist in this table.
   You will use these two statements 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.

First SQL Statement

The tables from the Parch & Posey database are stored in the backend of the box below. You will notice on the left margin under **SCHEMA** is a list of tables that were shown in the ERD earlier. We'll write queries and run them to see the results using the table below in the same way you would in most other database environments. In this lesson, we'll access just one table at a time, and in later lessons we'll be joining and doing aggregations across tables.

The Udacity SQL Environment

In order to get started, try running the query you saw in the previous video! In the environment at the bottom of the page, you have the ability to test your SQL code. In the left panel, you will find

the tables that we saw earlier in the ERD. In the right panel you can write your SQL code, and we can click the **EVALUATE** button to run the query. This may take a moment to run.
The **HISTORY** menu will show your previously run queries. The **MENU** will allow you to remove the SCHEMA from the left panel, as well as reset the database.

```
SELECT *
FROM orders;
```

You will notice that Derek was using a `demo` table (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 left margin. These tables are identical to those that you see from Derek in the future lessons too (with `demo` removed).
Every query you write will have at least these two parts: SELECT and FROM. In order to evaluate your query, you can either click **EVALUATE** OR hit **control + Enter**. If you get an error, it will sometimes cover the evaluate button, so this second option is very nice!
Again, **control + Enter** will run your query!

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.

Formatting Your Queries

Capitalization

You may have noticed that we have been capitalizing SELECT and FROM, while we leave table and column names lowercase. This is a common formatting convention. It is common practice to capitalize commands (SELECT, FROM), and keep everything else in your query lowercase. This 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.

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
```

SQL isn't Case Sensitive

If you have programmed in other languages, you might be familiar with programming languages that get very upset if you do not type the correct characters in terms of lower and uppercase. SQL is 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, I would again urge you to follow the conventions outlined earlier in terms of fully capitalizing the commands, while leaving other pieces of your code in lowercase.

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 practices to put a semicolon at the end of each statement, which also allows you to run multiple commands at once if your environment is able to show multiple results at once.

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
```

LIMIT Statement

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.

ORDER BY Statement

The **ORDER BY** statement allows us to order our table by any row. If you are familiar with Excel, this is similar to the sorting you can do with filters.
The **ORDER BY** statement is always after the **SELECT** and **FROM** statements, but it is before the **LIMIT** statement. As you learn additional commands, the order of these statements will matter more. If we are using the **LIMIT** statement, it will always appear last.

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.

Order By Part II

Here, we saw that we can **ORDER BY** more than one column at a time. The statement sorts according to columns listed from left first and those listed on the right after that. We still have the ability to flip the way we order using **DESC**.

WHERE Statements

Using the **WHERE** statement, we can subset out tables based on conditions that must be met. The above video shows how this can be used, and in the next sections, you will learn some of the common operators that are used with the **WHERE**statement.
Common symbols used within **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. `!=` (not equal to)

WHERE With Non-Numeric Data

The **WHERE** statement can also be used with non-numerical data. We can use the `=` and `!=` operators here. You also need to be sure to use single quotes (just be careful if you have quotes in the original text) with the text data.
Commonly when we are using **WHERE** with non-numeric data fields, we use the **LIKE**, **NOT**, or **IN** operators.

Arithmetic Operators

Derived Columns

Creating a new column that is a combination of existing columns is known as a **derived** column. Common operators include:

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

Order of Operations

Remember **PEMDAS** from math class? If not, check out this **link** as a reminder. The same order of operations apply 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 the case you mean to calculate the statement in part 2.

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.

LIKE Operator

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.

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. 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, 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 the work spaces in the classroom, note you can include an apostrophe by putting two single quotes together. Example Macy's in our work space would be 'Macy''s'.

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.

AND & BETWEEN Operators

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 all 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
```

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 **Col1**, **Col2**, ...	Provide the columns you want
FROM	FROM **Table**	Provide the table where the columns exist
LIMIT	LIMIT **10**	Limits based number of rows returned
ORDER BY	ORDER BY **Col**	Orders table based on the column. Used with **DESC**.
WHERE	WHERE **Col > 5**	A conditional statement to filter your results
LIKE	WHERE **Col LIKE '%me%'**	Only pulls rows where column has 'me' within the text
IN	WHERE **Col IN ('Y', 'N')**	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 **JOIN**s. This is the real secret (well not really a secret) behind the success of SQL as a language. **JOIN**s 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!
