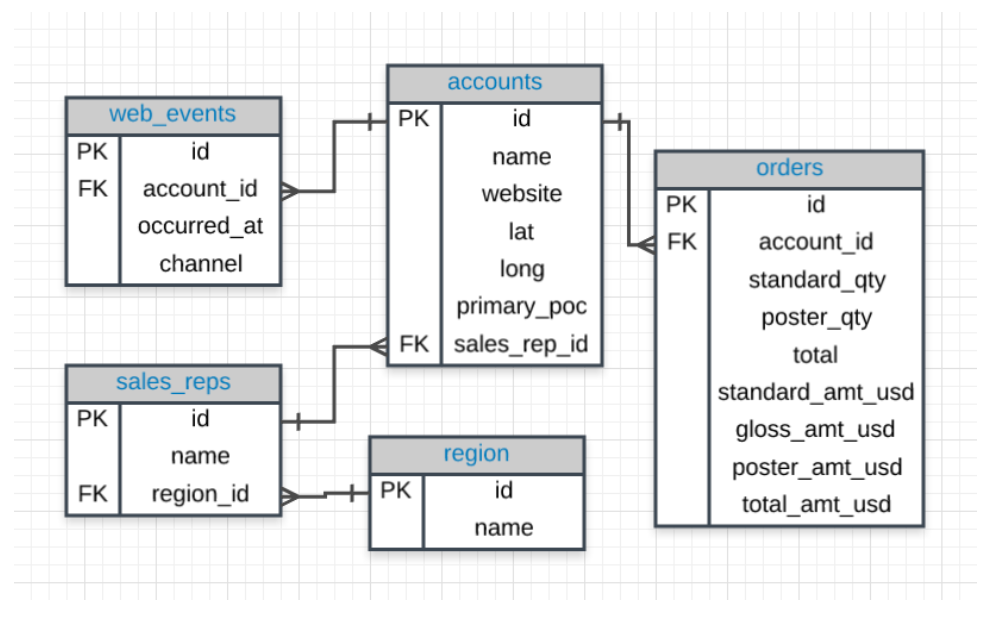
**Entity Relationship Diagrams**

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:

* The names of the tables.
* The columns in each table.
* The way the tables work together.



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.

**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**

* Data integrity is ensured - only the data you want entered is entered, and only certain users are able to enter data into the database.
* 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.
* 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.

**A few key points about data stored in SQL databases:**

* 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.
* 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!
* 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. (Note: You do not need to install PostgreSQL on your computer, unless you really want to. We provide SQL environments in the classroom for you to work in.)

Some of the most popular databases include:

* MySQL
* Access
* Oracle
* Microsoft SQL Server
* 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. Again, once you have learned how to write SQL in one environment, the skills are mostly transferable.

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

* CREATE TABLE is a statement that creates a new table in a database.
* DROP TABLE is a statement that removes a table in a database.
* 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!

**SELECT & FROM**

SELECT indicates which column(s) you want to be given the data for.

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

**SELECT** **id**, account\_id, occurred\_at **FROM** orders;

**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.

**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.**

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

**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;

**LIMIT**

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;

**SELECT** occurred\_at, account\_id, channel

**FROM** web\_events

**LIMIT** 15;

**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.

**ORDER BY**

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.

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.

1. 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;

1. 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

**LIMIT** 20;

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.

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).

**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;

1. 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.)*

**WHERE**

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:

* > (greater than)
* < (less than)
* >= (greater than or equal to)
* <= (less than or equal to)
* = (equal to)
* != (not equal to)

**SELECT** \* **FROM** orders

**WHERE** gloss\_amt\_usd >= 1000

**LIMIT** 5;

**WHERE with Non-Numeric 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!

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.

**SELECT** **name**, website, primary\_poc

**FROM** accounts

**WHERE** **name** = 'Exxon Mobil';

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.'

**Arithmetic 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.

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

\* (Multiplication)

+ (Addition)

- (Subtraction)

/ (Division)

Consider this example:

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

**FROM** orders

**LIMIT** 10;

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.

**SELECT** **id**, account\_id, standard\_amt\_usd/standard\_qty **AS** unit\_price

**FROM** orders

**LIMIT** 10;

1. 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*.

**SELECT** **id**, account\_id,

poster\_amt\_usd/(standard\_amt\_usd + gloss\_amt\_usd + poster\_amt\_usd) **AS** post\_per

**FROM** orders

**LIMIT** 10;

**Introduction to Logical Operators**

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

* 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.
* IN This allows you to perform operations similar to using WHERE and =, but for more than one condition.
* NOT This is used with IN and LIKE to select all of the rows NOT LIKE or NOT IN a certain condition.
* AND & BETWEEN These allow you to combine operations where all combined conditions must be true.
* OR This allows you to combine operations where at least one of the combined conditions must be true.

**LIKE**

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.

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';

**IN**

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.

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');

**NOT**

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.

1. 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');

1. Use the web\_events table to find all information regarding individuals who were contacted via any method except using organic or adwords methods.

**SELECT** \* **FROM** web\_events

**WHERE** channel **NOT** **IN** ('organic', 'adwords');

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

**SELECT** **name**

**FROM** accounts

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

1. All companies whose names do not contain the string 'one' somewhere in the name.

**SELECT** **name**

**FROM** accounts

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

1. All companies whose names do not end with 's'.

**SELECT** **name**

**FROM** accounts

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

**AND 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. 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

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.

**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';

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."

1. 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.

**SELECT** \***FROM** web\_events

**WHERE** channel **IN** ('organic', 'adwords') **AND** occurred\_at **BETWEEN** '2016-01-01' **AND** '2017-01-01'

**ORDER** **BY** occurred\_at **DESC**;

**OR**

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.

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

**SELECT** **id**

**FROM** orders

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

1. Write a query that returns a list of orders where the standard\_qty is zero and either the gloss\_qty or poster\_qty is over 1000.

**SELECT** \* **FROM** orders

**WHERE** standard\_qty = 0 **AND** (gloss\_qty > 1000 **OR** poster\_qty > 1000);

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

**SELECT** \* **FROM** accounts

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

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

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

**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.