**Highbrow:Introduction To Structured Query Language by Mike Schmitt**

**Episode #1 Intro To SQL**

Data is all around us, used in increasingly creative ways to drive both personal choices and business decisions.

Your fitness tracking apps and wearable technology are gathering data on your nutrition, your activity and workouts, and even your sleep habits.

Your favourite online shopping sites are showing you all the items available to purchase and what you might like to buy based on past purchases or similar users’ purchases.

Then there is the dating app on your smartphone showing potential matches based on your profile.

How is this possible? Where does this information come from?

Many companies today store a wealth of data related to the various parts of their business in databases. A database is a collection of interrelated information stored as objects such as tables, reports, and views. It is organized in a way that facilitates processing of data to serve a specific business function. With the fitness tracking app, each day’s step count is stored so that a user can view their progress over time.

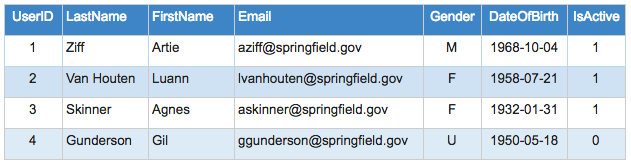
But how do we work with all this data? How can we view the data that is entered in a database? One option, which we will be looking at, is SQL. SQL, which stands for Structured Query Language, is a query language that allows a user or another application to view and manipulate data. Like any other programming language, syntax and rules must be adhered to in order to perform tasks. This syntax is run on what’s known as a relational database management system; some common ones include Microsoft SQL Server, MS Access, and Oracle.

In this course, we will be taking a high-level look at how data can be organized, displayed, and manipulated with SQL. We will cover the basic commands of SQL and allow you to practice on your own in a free online SQL environment. By the end of the course, you should be able to visualize how systems like online shopping searches work and how a list of available inventory matching your search criteria is viewable with a simple click of a button.

**Episode #2 Viewing Your Data**

Let's begin by simply viewing a dataset. But why might we want to do this? Well, perhaps you have a question you want to find the answer to. If you are an online retailer, you may need to gather info on who your users are or determine what inventory is currently in stock. If you are using a personal health tracking app, you might want to see how many steps you have taken each day or track how your weight loss goals are progressing. This data is likely stored in a database.

A database is made up of any number of tables composed of columns and rows similar to a spreadsheet. The column names contain a label for the values to be stored, and a row is a record in that dataset.



We can access this data by executing a query against a database with SQL.

SQL is a standard language for accessing and manipulating databases.

The following is the SQL syntax for viewing the Users table listed above:

SELECT \* FROM Users;

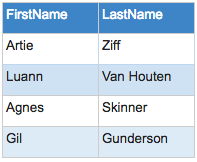
The SELECT is the most basic query in SQL—it allows you to view the records in the table specified. This statement can be run standalone in an editor such as SQL Developer.

The asterisk (\*) in this statement, read aloud as “star,” indicates we want to view all the columns. So this statement says select all the columns from the Users table and have it displayed to us.

The semicolon is a statement terminator. It signals the end of the statement. It is not required in all database systems, but it is general good practice for readability.

We can also limit the fields to view by listing out the column names specifically. Here’s an example if we only wanted the first and last names in our Users table returned:

SELECT FirstName, LastName FROM Users;

Note that the order of the fields in our query does not need to correspond to their order in the table, but the column name must match exactly.You can now query your database to see what it holds!

Explore on your own with an [SQL Fiddle](http://gohighbrow.us8.list-manage.com/track/click?u=51c6d69760513a5e5e4f75b29&id=36de9a5088&e=0e92abd919) of today’s lesson. SQL Fiddle provides a place to test your queries. You’ll notice three panels when you click the link. The left panel builds the database (which we have already done for you), the right panel is where you can write and test queries, and the results are displayed in the bottom panel.

**Episode #3 Filtering Your Data**

We now know how to view the data in our database tables, but what if we don’t need every record returned with a query? What if we only need a portion of the records on the table? With SQL, we can filter what is returned with a WHERE clause.

Let’s use our online retailer example and look at their inventory and look at their inventory stored on the following *Inventory* table:

A handy and common feature on many websites is a search bar. A search function allows you to find out whether an item you’re looking for is in inventory and available for purchase.

E.G we are interested in purchasing pants, so we type ‘Pants’ in our search. Behind the scenes, an SQL query is run against the database that looks something like this:

SELECT \* FROM Inventory WHERE Type=’Pants’;

We are familiar with the first half of the syntax, which is our basic SELECT, but there is now an additional WHERE clause following the table name. This acts as a filter to limit the returned results to what matches the filtering criteria. The results are limited to those where the value in the Type column is equal to ‘Pants.’

Note that the value in our query is contained in single quotes.

This is required for any data type aside from a number, so any text or dates.

Now what if we need to further refine our search with additional criteria? Not a problem! You can append any number of additional filtering statements by adding the additional criteria separated by an AND. Here’s an SQL example that further limits our results to those with a colour of ‘Gray’:

SELECT \* FROM Inventory WHERE Type=’Pants’ AND Color=’Gray’;

Now we know how all those websites can bring up exactly what we’re looking for!

Explore on your own with an [SQL Fiddle](http://gohighbrow.us8.list-manage.com/track/click?u=51c6d69760513a5e5e4f75b29&id=d5f7df2f1f&e=0e92abd919) of today’s lesson! The database has already been built for you, so you can test your queries in the panel on the right-hand side!

**Episode #4 Wildcards**

Up until now, we’ve done searches where we’ve known the exact criteria we are looking for. But what if we are querying our database and don’t know exactly what we are looking for? Perhaps you remember the first part of the name brand but can’t quite recall it exactly. Good news! That’s not a problem! We can get close enough by returning similar or like results.

Let’s do another search in our online retailer database for a specific pair of pants:



For cases with uncertainty, we can use what’s known as a wildcard, which is represented with a percentage sign (%).

Let’s see what the following SQL query pulls up and then break down the syntax:

SELECT \* FROM Inventory WHERE Brand LIKE ’B%’;

Our result set:



Anything without a ‘B’ in the first position of the Brand has been excluded from our result set, and we are left with only brand names that begin with ‘B.’

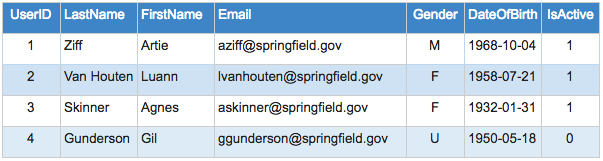
In the syntax, we use the SELECT and WHERE as we normally do, but note that where we used an equal sign before is now replaced with a LIKE. This signals that we’re not looking for an exact match but instead similar or like matches. The percentage sign, or wildcard, comes after the ‘B’ and will return any result where the value in the Brand field begins with a ‘B’ followed by any amount of text. In this case, both “Banana Brand” and “Boga” match.

Close enough for jazz! Bananas Brand was what we were looking for!

Explore on your own with an [SQL Fiddle](http://gohighbrow.us8.list-manage1.com/track/click?u=51c6d69760513a5e5e4f75b29&id=3ce4d66d36&e=0e92abd919) of today’s lesson!

**Episode #5 Adding And Removing Data**

We were able to view our data in the previous lessons, but how did those records get in the database in the first place? And what if we want to remove data? Let’s revisit our Users table from earlier lessons and simulate the lifecycle of a customer.



So a new customer comes along, loves the pants on your website, and decides to create an account to keep current on your offerings. Your site prompts for some basic personal information (hint: what is listed in our table above), which the user types in. The user clicks submit and triggers the following SQL to run:

INSERT INTO Users (Email, LastName, FirstName, Gender, DateOfBirth, IsActive)

VALUES ('milhouse@springfield.gov', 'Van Houten', 'Milhouse', 'M', '1979-04-01', 1);

Let’s break down the syntax.

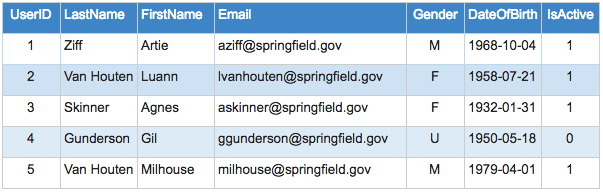
The first part is INSERT INTO, which signals we will create a new record in our table, followed by a listing of our column headers.

The column header listing signals which fields we will be passing information for. In this case,

it is the columns Email, LastName, FirstName, Gender, DateOfBirth, and IsActive.

The second part of the statement is simply the related information. This is signalled with VALUES followed by the column data that needs to be listed in the same order as the column specification following the INSERT INTO. For instance, the first data value is ‘milhouse@springfield.gov,’ which corresponds to our first listed column header of Email. Both the column headers and the data values need to be enclosed in parentheses.

We can now view our new record:



Perhaps you want to use the same table for an insert statement but do not want to include data for every column. For instance, to exclude gender, the SQL statement would alter slightly in the fields indicated to be included in the record insert.

INSERT INTO Users (Email, LastName, FirstName, DateOfBirth, IsActive)

VALUES ('milhouse@springfield.gov', 'Van Houten', 'Milhouse', '1979-04-01', 1);

This will simply insert a null value into any field on the table that is not listed.

Some time passes, and now we come to the end of the customer lifecycle and our customer wishes to unsubscribe. Sigh—can’t win them all.

Our user clicks unsubscribe, and the following SQL runs:

DELETE FROM Users WHERE UserID=5;

The syntax is similar to a SELECT statement, however the SELECT \* is replaced with DELETE. The statement says DELETE from table Users the record that matches UserID=5. The WHERE clause is very important here! If we didn’t specify which record to delete and instead ran this SQL statement,

DELETE FROM Users;

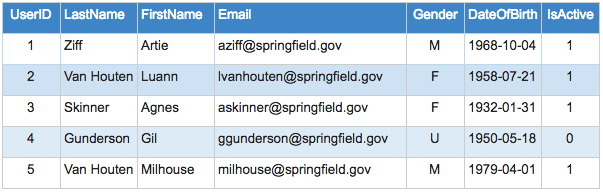
all records would have been deleted from the Users table, since we didn’t include a WHERE to limit which records are deleted!

And that’s how you add and remove a record from a table!

Explore on your own with an [SQL Fiddle](http://sqlfiddle.com/#!9/8a107/1) of today’s lesson! Since this is not a SELECT statement, your syntax must be written in the panel on the left-hand side.

**Episode #6 Updating Your Data**

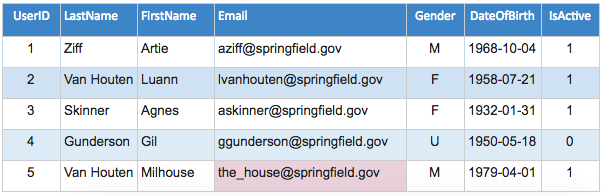
There are instances when one or more fields need to be updated on a record. For instance, what if a user’s mailing address or email changes and needs to be updated? Or if a value was entered incorrectly and needs to be corrected? We can use an UPDATE statement to modify the values of records in a table. Here is the current state of our Users table:



Let’s update the email address to ‘the\_house@springfield.gov’ for the user in our last example where the UserID was 5. Here is the SQL:

UPDATE Users SET Email='the\_house@springfield.gov' WHERE UserID=5;

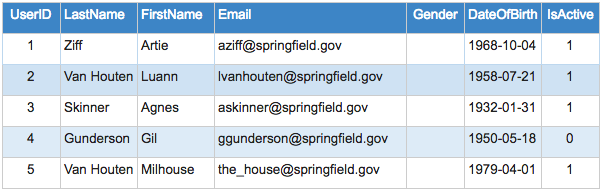
Our command is UPDATE, which is running against the Users table, we are setting the value of the Email field to ‘the\_house@springfield.gov,’ and we are only doing this for the record with a UserID of 5. We do not want to update all the records, so be sure to include the WHERE clause! Once the statement is run, the table will look like this:



There could be instances where you want to update every record on a table, and in those cases, you would not need a WHERE clause. If we wanted to remove the Gender information by nulling the field, which is essentially blanking out the field, we could run the following SQL statement:

UPDATE Users SET Gender=NULL;

This would run against every record on the Users table and result in the dataset showing like so, where the Gender information is blank:



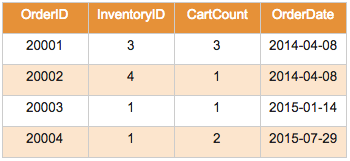
And now the data is gone for every record!

And that’s how you can modify the data of existing records!

Explore on your own with an [SQL Fiddle](http://sqlfiddle.com/#!9/ab00ca/1) of today’s lesson! Since this is not a SELECT statement, your syntax must be written in the panel on the left-hand side.

**Episode #7 Joins**

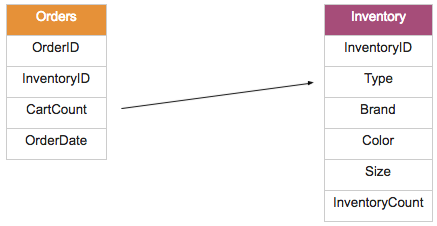
What if we need to combine or link data from two different tables? Say a small business makes a sale and needs to pull merchandise info from our Inventory table to create an order record. We can use a JOIN to pull data from two or more tables using a common field to link them. There are multiple types of JOINs available, and for our example, we will use an INNER JOIN. This type of JOIN returns all rows from the linked tables where the matching criteria is met. We will join the Orders table to our Inventory table, and the Orders table is structured like so:



Recall that this is what is contained in our Inventory table:



And here is the relationship model between the two tables showing the field in common that we can use to link:



The InventoryID field is shared by both tables, and we will use this to link. Let’s say we want to pull the merchandise Type, Brand, Color, and Size for all orders along with the CartCount and OrderDate. Here is the SQL syntax, which we will look at in further detail:

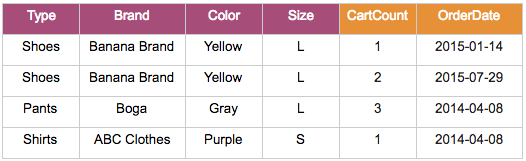
SELECT Inventory.Type, Inventory.Brand, Inventory.Color, Inventory.Size

, Orders.CartCount, Orders.OrderDate

FROM Orders INNER JOIN Inventory ON Orders.InventoryID=Inventory.InventoryID;

The SELECT portion looks somewhat familiar, but you’ll notice that each column name has the table name as a prefix. This indicates which table to pull the column from. For instance, Inventory.Type indicates we want to use the Type field from the Inventory table in our column selection. After the SELECT comes the JOIN syntax. We are saying take table Orders and INNER JOIN table Inventory. The column to match on, InventoryID in this case, follows the ON syntax. We are matching on InventoryID, so we write this as Orders.InventoryID=Inventory.InventoryID, again listing the table prefix to indicate which field to pull each column from.

Here is the result set once the query is executed:



This looks pretty similar to many online order confirmations!

Note that there are a few additional types of JOINs available to you that we won’t get to but are quite useful:

LEFT JOIN: Return all rows from left table plus any matched rows from right table.

RIGHT JOIN: Return all rows from right table plus any matched rows from left table.

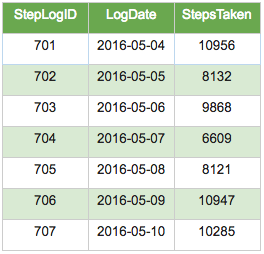
FULL JOIN: Return all rows when there is a match in either table.

And now you know how to link tables!

Explore on your own with an [SQL Fiddle](http://sqlfiddle.com/#!9/974325/1) of today’s lesson!

**Episode #8 Functions**

SQL provides a number of built-in functions that allow you to perform calculations on your data set. Let’s take a look at these and use our fitness tracker app as an example. Available to us is a table named StepLog, which holds the number of steps that we take each day:



What is displayed in your app, however, is much more helpful, as we are given averages and the total number of steps, among other data points. These values can be calculated rather easily with built-in SQL functions, and here are a few available for use:

COUNT(column\_name) – This returns the number of rows.

AVG(column\_name) – This returns the average value of the passed column.

MAX (column\_name) – This returns the largest value in the passed column.

MIN(column\_name) – This returns the smallest value in the passed column.

SUM(column\_name) – This returns the sum of the passed column.

Let’s calculate the sum and average of our steps taken:

SELECT SUM(StepsTaken), AVG(StepsTaken) FROM StepLog;

Our SQL statement is similar to a basic SELECT, but we now have the function calls on the columns we wish to perform the calculations on (the sum of column StepsTaken and the average of column StepsTaken). Our result set when we execute the query shows as follows:



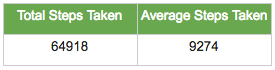
We are shown that the sum of steps taken is 64,918 and the average is roughly 9,274, but the column headers aren’t too pretty. We can rename those with what is called an alias to make them more descriptive.

SELECT SUM(StepsTaken) AS “Total Steps Taken”

, AVG(StepsTaken) AS “Average Steps Taken”

FROM StepLog;

The additions to the SQL syntax are the AS, which signals we will be renaming the column header, followed by our renaming in quotes. The quotes allow us to use spaces in the header name. Our result set will now appear like so:



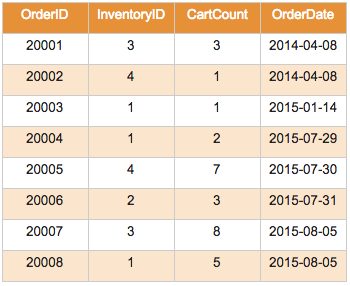
This looks much cleaner!

And now we have the ability to gather some additional insight into our data with basic calculations by way of functions!

Explore on your own with an [SQL Fiddle](http://sqlfiddle.com/#!9/97a71/2) of today’s lesson!

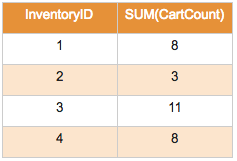
**Episode #9 Grouping**

With the previous lesson, we began to see how aggregate functions may be useful. But what if we wanted those values broken down further? What if our online business wanted a breakdown of the customer base by address? Or wouldn’t it be helpful to get a breakdown of sales by item? We can do this! Let’s take a look at our Orders table, which includes the InventoryID of the item sold along with the quantity sold (CartCount):

We want to get a sum of the CartCount values using the SUM() function, but we also want the sum broken down per item. There are InventoryIDs 1, 2, 3, and 4, so we want the total for each of these. Here is the SQL syntax:

SELECT InventoryID, SUM(CartCount) FROM Orders GROUP BY InventoryID;

Notice we have added InventoryID to our selected columns and added GROUP BY InventoryID to the end of our statement. GROUP BY indicates which columns to break down the aggregated portion by. Our result set would then look like:

Look! From the result set, we just discovered what our hottest-selling item is!

Also note that you can include additional aggregates like AVG() or MAX() in a group by query or list additional fields from your table to group by.

And now you know how to group your result sets!

Explore on your own with an [SQL Fiddl](http://sqlfiddle.com/#!9/d2dde/1)e of today’s lesson!

**Episode #10 Next Steps**

Hopefully you are beginning to understand how to use SQL to access and manipulate data and how that data can be used in different situations. The next time you open a smartphone app or visit a website, you can begin to brainstorm what data they might be collecting and using and how it is being accessed and displayed. We have only scratched the surface of what SQL has to offer. Many additional advanced queries and techniques exist that can offer further ways to manipulate your data to draw additional insights. There are many places to pick up where we left off if you would like to continue your exploration of the query language.

Where to Now?

[SQL Fiddle](http://sqlfiddle.com/) - Continue to use this site to test queries on your own user-created tables. You can create datasets and author your own queries and scripts to view and manipulate this data. Practice makes perfect!

[W3Schools](http://www.w3schools.com/sql/) - A handy reference for SQL syntax that also provides detailed examples to learn from. Bookmark this site as a spot to quickly look up how to properly use a function or command. Can’t recall how to properly perform a JOIN? No problem—W3Schools lists the different JOIN types, syntax, and use cases for each. There are even “try it yourself” sections similar to our practice problems.

[Oracle](https://livesql.oracle.com/) - You can browse scripts and tutorials created by other programmers and even practice your own chops in their databases of pre-existing tables. A great way to learn is by example, and you also get an idea of real-world problems being solved with SQL.

[Stack Overflow](http://stackoverflow.com/questions/tagged/sql) - If you get stuck with a syntax error or need to find a solution to a problem you just can’t seem to get your head around, there is an entire community of fellow SQL gurus to help you in a Q&A setting. Search to see if someone else has already run into your error or issue (it’s likely someone has) or ask your question to get input from programming professionals and enthusiasts from around the world.