**INTERVIEWING YOUR DATA WITH SQL**

*A guide to data analysis with SQL, by Jill Castellano*

What the heck are we talking about today?

* **Relational database**: “A collection of data items organized as a set of **tables** from which data can be accessed or reassembled in many different ways without having to reorganize the database tables.”
* SQL (“Sequel”): The standard **language** used in relational databases
* Database applications: The programs/software tools you can use to communicate with your databases. Some are expensive, some are free. We are using MySQL Workbench.
* Query: When you ask your data a question using SQL by typing commands into your database program. This is how we **interview our data.**

Today’s tools: mysql workbench

* MySQL: One of many database management **servers** used to host your data. This is what we will use today. Another option is called “SQL Server”.
  + Note: different servers use slightly different variations of the SQL language, but they are similar enough that practicing in one will teach you the others
* MySQL Workbench: Free, open source database application that works on all sorts of computers.
* Standard workflow: Open MySQL Workbench, connect to the server, import your data, query your data, save the results, export the results

Why is this better than Excel?

* More powerful: process way more data
* Group the data any way you want to
* Keep track of your work by saving your steps
* Redo you’re analyses quickly if you get new data
* *If you want to do even more, start learning Python.*

Let’s get started!

* REMEMBER: This takes times to learn, and needs to be practiced like any other language. You WILL NOT recall everything you learned here.
* **You just need to know enough to know what to google.**

How to learn more

* Google it or ask me for help
* Stack Overflow: post your issue or see if others have
* Free tutorials online
  + W3 schools: <https://www.w3schools.com/SQL/deFault.asp>
  + SQLite: <https://github.com/tthibo/SQL-Tutorial>
* IRE and NICAR sessions and tipsheets
  + Advanced SQL from NICAR 2015: <https://github.com/eklucas/NICAR-Adv-SQL>
  + Advanced SQL from NICAR 2016: <https://github.com/taggartk/2016-NICAR-Adv-SQL>

Starting up MySQL Workbench, importing data and viewing it

Connecting to the MySQL server:

* Open MySQL Workbench
* Click on the gray square that says “Local instance MySQL Router”. We are connecting to servers hosted on our own computers, called “localhost.” Password is “**Password1**”.

How MySQL Workbench looks when you open it:

* On the left, under “schemas” you have your **databases**, each represented by little cylinder icons.
* Click the arrow to see your **tables**, which are the spreadsheets stored in your database.
* The box in the middle is where you type your **queries**.
* The box on the bottom, called “output”, shows you what actions you performed and whether or not they were successful. This is where you will see **error messages** if your query didn’t work.
* When you perform a query, another box will pop up with the **results**.

Create a new database:

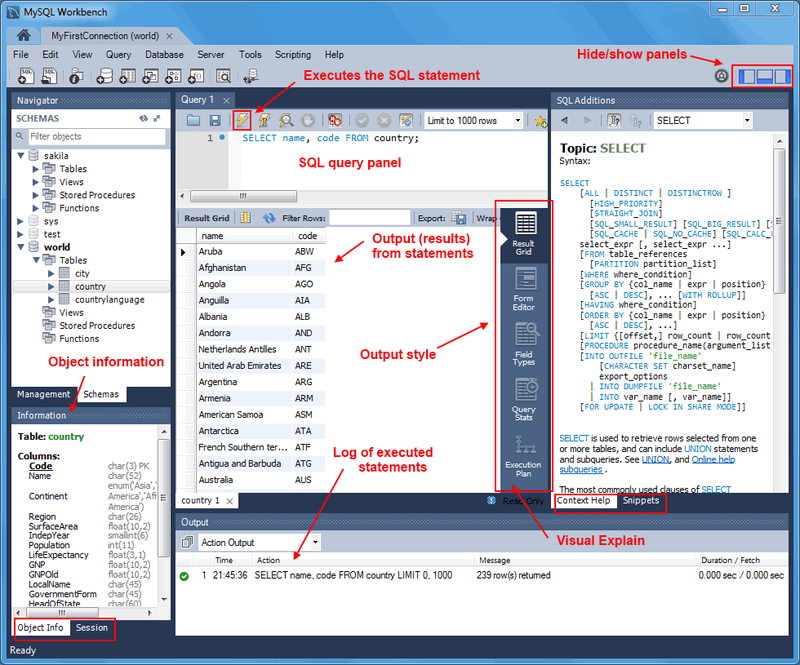
* Click the icon on the upper toolbar that looks like a cylinder with a plus sign. That’s a dabasase or “schema”. Right click in the box and press “create schema”. Name your new database “**training**”. Generally, you want to keep the names of your databases, tables, column headers and queries short and add an underscore instead of a space if it’s two words so they are easier to type later. Press “apply”, “apply” and finish.
* Now the database shows up in your “schemas” box. Double click it to see more.

Importing data into MySQL Workbench:

* Right click on the “tables” icon below your new database and press “data table import wizard.”
* Click “browse” and select the file you want to import. This program works with CSVs, not Excel files, so you will have to convert Excel files to CSVs to import them. Press “next.”
* Select “create new table”. The drop down menu to the right lets you select which database you want to upload the table into. Make sure it’s uploading into “training”.
* Farther to the right, type in the name of your new table. Let’s call it “**PS\_salary**”. Click “next”.
* Now check off all the columns you want to import. Make sure each column is being read in the correct **data type** – numbers should be read as “int” for integer, strings (letters) are “text” or in some programs “varchar”. MySQL Workbench unfortunately only reads dates in a “datetime” format, YYYY-MM-DD HH:MM:SS, so you should convert dates to this format, adding 00:00:00 to the end before importing your data.
* Click through the rest of the options and check at the end that it says the import was successful.
* Right click anywhere in the “schema” box and press “Refresh All”. Now the new table you imported will appear under the “training” database.

Viewing our data:

* Hover over the table name on the left menu and click the small wrench icon. You’ll see a screen pop up with information about the table, including data types and column names. You can very easily change column and table names here.
* Right click on the table and press “select rows.” It might tell you “**limit 1000**”, which means it’s going to return a maximum of 1,000 rows, making the data load faster. You can change the limit in the dropdown menu above the query editor. A new box will pop up with all your data in it so you can check that the data imported properly.



Basic commands and querying using SQL

Click the icon on the top left of your MySQL Workbench, right below the menu bar. This will open the SQL editor where you can enter your queries.

Each query consists of a **statement** (command) that you give to your computer. Each statement is made up of **clauses** that tell the computer different kinds of information you need in order to return the data you want.

For starters, know that there are five basic clauses in a **SELECT statement,** the simplest SQL statement, which must be written in a **particular order**. We will discuss each one in detail. Notice that these clauses produce similar functions to what you will see in Excel, but here they’re much more powerful.

**SELECT:** what columns you want to see

**FROM:** which dataset you want to see them from

**WHERE:** filter out data you do or don’t want to include

**GROUP BY:** group data into categories, like a pivot table

**ORDER BY:** sort in ascending or descending order

**SELECT and FROM clauses: the building blocks of SQL**

The two most basic SQL clauses are SELECT and FROM. For every SQL query, you will always need to state which columns you want to (SELECT) and the table where those columns are located (FROM). The other clauses are optional, but these are not. In MySQL Workbench, the FROM statement includes the database your table is in, then a period, and then the name of the table. (Most programs don’t need you to specify the database, but MySQL Workbench does.)

To select all columns and data from the ps\_salary table, enter this:

SELECT \*  
 FROM training.ps\_salary;

The \* is a wildcard character, which means “give me everything” in this context. Click the **lightning bolt** button or press CTRL + Enter to “run” the query. The box that pops up should give you the same thing we saw when we viewed our data earlier. Note that the log at the bottom of the screen tells us how many rows of data were returned.

MySQL Workbench displays up to 1000 rows at once. You can get to more pages of results by clicking the “fetch rows” icons directly above the results.

Now let's select specific columns from the ps\_salary table.

SELECT EmployeeName, JobTitle, Year

FROM training.ps\_salary

If the column names aren't clear or are difficult to remember, you can rename the column headers in your output easier to make them easier to understand. (This doesn’t change the underlying data, just your results.) Use the AS operator to rename a column.

SELECT EmployeeName AS Employee, JobTitle AS Job, Year

FROM training.ps\_salary

You can rearrange the order of column names in the SELECT statement to tailor how data is presented in the output. (Again, this doesn’t affect the underlying data.)

SELECT Year, EmployeeName, BasePay

FROM training.ps\_salary

**Formatting**

The capitalization of SQL syntax words is not necessary, but helps to differentiate between SQL commands and other information. It also makes the information easier to scan. This is also why you can include new lines for each successive SQL command. They make reading easier but are not necessary.

You can use a semicolon at the end of each query to indicate the end, but it is not required by all SQL software. It’s a good habit to get into.

**Saving your work**

Saving your queries helps ensure that you don’t forget the work that you did, and also allows you to go back to your code later in case you want to see if you made any mistakes. If your data is updated later on, you can always return to your saved queries and run them again, quickly getting a new output.

After running your query, press the star with the plus sign icon to save your “snippet”, meaning your query. The bar on the right side of the screen will pop open, where you can name your query.

You can export your results (output) as a CSV file by pressing the “Export” button right above your results. Name your CSV file and hold onto it for later use.

**WHERE clause: Filtering the data**

Now we're going to look at one of the most useful parts of SQL. The WHERE clause lets you filter your data based on any number of criteria. If a row matches the given criteria, that row is returned.

For example, we can limit the rows to only just the 2014 salary data using the following query. The output will include all rows where the statement “year = 2014” is true.

SELECT \*   
 FROM training.ps\_salary  
 WHERE year = 2014

In addition to equals (=), other common comparison operators include does not equal (!=), greater than (>), less than (<), greater than or equal (>=) and less than or equal (<=). You can use the equals sign if you want to return an exact match for certain text in your cells.

SELECT \*   
 FROM training.ps\_salary  
 WHERE year >= 2014

SELECT EmployeeName, BasePay, Year

FROM training.ps\_salary

WHERE TotalPay > 250000

SELECT Year, Agency, EmployeeName, BasePay

FROM training.ps\_salary

WHERE Agency = “Palm Springs”

Of course, your data must be numeric in order to use mathematical operators such as > or <. That wouldn't make much sense when comparing two words or phrases.

Note that when we queried for numbers, we could enter the number by itself (*TotalPay = 250000*). But when we query the computer for words and other values, we have put single or double quotes around the text (*Agency = “Palm Springs”*). The only significant reason to prefer double quotes is in case you have an apostrophe in your data that the computer would read as a quotation mark.

**LIKE operator**

**Operators** are parts of clauses that help explain to the computer what information you are looking for.

Perhaps you only know a part of the text that you are seeking. For example, you want to know the salaries of all the people working in the fire department, but their job titles might be different. SQL offers a useful command that lets you search by pieces of text. The % acts as a wildcard character, meaning it can represent zero or more characters on either side of a word, phrase or letter.

For the query below, any cells in the JobTitle column that contain ‘Fire’ will be returned:

SELECT \*

FROM training.ps\_salary

WHERE JobTitle LIKE '%Fire%'

You could also use it to only show features that start or end in a certain phrase. With the wildcard % before Fire, it will search for anything that ends with that phrase.

SELECT \*

FROM training.ps\_salary

WHERE EmployeeName LIKE '%Carranza'

**AND and OR operators**

AND and OR are two more useful filtering tools that can help us answer some questions about the data. Let’s say we want to know which employees were making over $150,000 in base pay but only in the year 2016. To capture both concepts, we can use the AND filter:

SELECT \*

FROM training.ps\_salary

WHERE BasePay > 150000 AND Year = 2016

SELECT \*

FROM training.ps\_salary

WHERE BasePay > 150000 AND Year = 2016 AND Agency = “Palm Springs”

Notice how the more AND operators we add, the narrower and narrower our data becomes.

With OR we can add more options to our query.

SELECT \*

FROM training.ps\_salary

WHERE BasePay > 150000

AND (Agency = “Palm Springs” OR Agency = “Cathedral City”)

With OR, rows are returned as long as they match at least one of the filters. Notice how this broadens out our data with the more OR operators we add.

Notice the use of parentheses around the OR operator in the above query. This groups the result of that comparison, which is then used in the AND comparison. The parentheses help to stay organized.

**ORDER BY clause: Sorting your data**

Since we're looking at salaries, it might help to rank the tracts by largest to smallest. This is where we’ll use the ORDER BY command. You can select the column which will determine the order of the rows.

This query returns the employees in Coachella ranked by highest to lowest paid.

SELECT EmployeeName, JobTitle, Year, TotalPay

FROM training.ps\_salary

WHERE Agency = “Coachella”

ORDER BY TotalPay DESC

The default setting for ORDER BY is ascending order. In the above query, ASC order could be achieved by writing “ORDER BY BasePay ASC” or “ORDER BY BasePay.”

**LIMIT operator**

The LIMIT command forces your query to only return the specified number of rows. This is commonly used in conjunction with ORDER BY to show a small set of ranked rows ("The 10 highest paid police department officials").

This query returns results in descending order, based on total pay, and only returns the first 10 values:

SELECT EmployeeName, JobTitle, Year, TotalPay

FROM training.ps\_salary

WHERE JobTitle LIKE "%Police%" AND Year = 2016

ORDER BY TotalPay DESC

LIMIT 10

This query shows the 10 highest paid police department officials in the Coachella Valley in 2016.

Basic calculations using SQL

**Aggregate functions**

SQL offers built-in **aggregate functions** to perform calculations on your numeric data. COUNT, MAX, MIN, and AVG are some common ones.

**COUNT function**

Return the number of rows matching your query. This is especially useful when combined with WHERE statements to understand how many rows match your filters. This shows us how people were paid by the city of Palm Springs in 2016.

SELECT Count(EmployeeName)

FROM training.ps\_salary

WHERE Year = 2016

Notice that the column header for the output is the aggregate function you entered into your SELECT statement. Let’s rename it so we can remember what it is.

SELECT Count(EmployeeName) AS PS\_Emp\_2016

FROM training.ps\_salary

WHERE Year = 2016

**AVG function**

This command is especially helpful with salary data. The below query tells us the average yearly overtime pay of each employee:

SELECT AVG(OvertimePay)

FROM training.ps\_salary

Or the average overtime pay in 2016 specifically:

SELECT AVG(OvertimePay)

FROM training.ps\_salary

WHERE Year = 2016

**MAX function**

Return the greatest value for the column specified.

SELECT MAX(OvertimePay)  
 FROM training.ps\_salary

**MIN function**

Return the smallest value for the column specified.

SELECT MIN(OvertimePay)  
 FROM training.ps\_salary

Or we can view all of these aggregate functions together to get a more complete understanding of the data we have on overtime pay:

SELECT MAX(OvertimePay), MIN(OvertimePay), AVG(OvertimePay)

FROM training.ps\_salary

Now let’s say we want to know how much people are earning in terms of their base pay combined with their overtime pay. We can add up the values of columns easily in the SELECT clause.

SELECT EmployeeName, Year, (BasePay + OvertimePay) AS Base\_OT\_Pay

FROM training.ps\_salary

Notice how we could use this method to fact check our data. We have a column called TotalPay, which should be the sum of the base pay people are earning, plus their overtime pay and their “other” pay. It doesn’t include benefits. Let’s see if we can add up those different payments and they will match up with the TotalPay column.

SELECT EmployeeName, Year, (BasePay + OvertimePay + OtherPay) AS TotalPayAdded, TotalPay

FROM training.ps\_salary

**Error Messages**

Dealing with errors is a normal, albeit frustrating, part of using a SQL program. A misplaced comma or letter can throw off your whole command. If you typed something incorrectly, MySQL Workbench will return an **error message** in the bottom part of the screen. Try to carefully read these messages to get an understanding of what you did wrong.

For example, let’s say I typed a previous function incorrectly:

SELECT MAX(OvertimePa)

FROM training.ps\_salary

On the bottom of the screen, I see a message that says, “Error code: 1054. Unknown column ‘OvertimePa’ in ‘field list’.” This message is telling me that I typed a column name called “OvertimePa” which does not exist. I can quickly recognize that I meant to type “OvertimePay” and correct the error.

**Staying organized with notes**

As your queries grow more and more complex, it helps to write comments within your SQL code to note what a particular command shows or explain why you’re using that query in the first place.

If you are familiar with other programming languages, then you are probably familiar with the idea of comments in your code. These are lines that are not executed and only exist for people reading the code.

SELECT MAX(OvertimePay), MIN(OvertimePay), AVG(OvertimePay)

FROM training.ps\_salary

/\*

Everything inside here is a comment and won't be executed in the SQL query.

This query shows me the maximum, minimum and average yearly overtime pay.

\*/

**Data types**

Stay aware of the different types of data in your tables. Common types include integers (whole numbers), floats (numbers with decimals), booleans (True or False), text and dates.

This is very important when you have data containing a leading zero (e.g. zip code 07712). If you were to convert that to an integer (7,712), it would lose its meaning. You should also make sure numeric data is stored as numbers and not text so that you can make use of mathematical operators (=, < and >).

**Differences in SQL syntaxes**

The various flavors of SQL (SQLite, MySQL, PostgreSQL, SQL Server, etc.) all have slightly different syntaxes, but they are mostly the same when it comes to basic usage. This can be annoying when switching between the SQL languages, but the good news is that they all have been around for decades. That means most syntax fixes are well-documented and only a quick Google search away.

**NULL**

One confusing point with SQL and programming languages is the idea of NULL. In databases, you can declare whether or not a column allows NULL entries, meaning whether or not they can lack any values. This is a subtle but significant difference between an empty value. An empty value means the emptiness is reported, whereas a NULL value means nothing is reported at all. It is the lack of anything. So, you can count the number of empty cells by searching for cells using “”, but you need to count the number of null cells using something else.

You can filter based on whether a cell is NULL using IS or IS NOT as the comparison operators, instead of = or !=. Again, this is because NULL is not really equal to anything; it's the absence of any value.

SELECT \*  
FROM training.ps\_salary  
WHERE EmployeeName IS NULL

Heavy analysis using SQL

**GROUP BY clause:** **the pivot table of SQL**

To do more advanced analysis in SQL, we’re going to introduce one of SQL’s more powerful clauses: the GROUP BY clause. This works like a pivot table in Excel, where you can slice your data any which way by lumping it into categories.

Let’s take another look at our salaries data. What are some ways we might want to group it? Looks like we could group it by employee name, by year, by agency or by job title. Let’s start with employee names so we can see a list of all the unique names of people employed by cities in our area. (Most people are included in the data more than once because they were paid multiple years.)

SELECT EmployeeName

FROM training.ps\_salary

GROUP BY EmployeeName

GROUP BY is often used in combination with an aggregate function like COUNT, SUM or AVG after the SELECT statement. Let’s say we want to see how much each city has doled out in overtime pay over the three years of data we have. In this case, we want to tell MySQL Workbench to GROUP BY all the cities, and then SUM the amount of overtime pay.

SELECT Agency, SUM(BasePay)

FROM training.ps\_salary

GROUP BY Agency

It’d be helpful to see the cities ranked by overtime pay. We can do that with our ORDER BY clause.

SELECT Agency, SUM(BasePay)

FROM training.ps\_salary

GROUP BY Agency

ORDER BY SUM(BasePay) DESC

Now let’s say we want to see which type of employee is making the most in average overtime pay.

SELECT JobTitle, AVG(OvertimePay)

FROM training.ps\_salary

GROUP BY JobTitle

ORDER BY 2 DESC

Notice that I can type a 2 instead of typing out “AVG(OvertimePay)” all over again. The AVG function is the second piece of information in our SELECT clause, so we can just tell our computers to give us back the second thing in the clause to save some typing (and potential typos.)

Try playing around and seeing what other patterns arise in the data. There’s so much to explore!

**Next commands to learn: ALTER TABLE and JOIN**

What’s next? Totally different kinds of statements in SQL. So far, we’ve only been working with SELECT statements that have those five basic clauses. Turns out there are other kinds of commands!

The **ALTER TABLE** command allows us to modify the data table we are working with rather than return data to us. You can add columns (ADD COLUMN), delete columns (DROP COLUMN), rename a table (RENAME TABLE), change the data type (MODIFY COLUMN) and more.

The **JOIN** command lets you combine or match up two or more different datasets based on a column they have in common. Then you can do a whole host of more analyses on them!

*Partially adapted from an IRE guide from 2017*