## AGGREGATE DATA FOR ANALYSIS

As part of your analysis, you’ll often have to combine data in order to gain insights and complete business objectives. In this part of the course, you’ll explore the functions, procedures, and syntax involved in combining, or aggregating, data. You’ll learn how to do this from multiple cells in spreadsheets and from multiple database tables using SQL queries.

### **Learning Objectives**

* Describe functions and procedures that may be used to combine data from multiple cells in a spreadsheets
* Demonstrate how to use functions and syntax to create SQL queries for combining data from multiple database tables
* Use VLOOKUP to query data, trim data, convert text data to numeric data, and create a summary table from a queried information

## VLOOKUP AND DATA AGGREGATION

### [AGGREGATE DATA FOR ANALYSIS](https://www.coursera.org/learn/analyze-data/lecture/UILlm/aggregate-data-for-analysis)

Up next we'll explore something called data aggregation.

**Aggregation means collecting or gathering many separate pieces into a whole**. For example, the Milky Way galaxy is an aggregation of stars, dust, and gasses. So data aggregation is the process of gathering data from multiple sources in order to combine it into a single summarized collection.

**In data analytics, a summarized collection, or summary, describes identifying the data you need and gathering it all together in one place**.

For example, let's say you have a cabinet full of different puzzles. One day, a shelf breaks, and all the boxes topple over, scattering the puzzle pieces everywhere.

To get each puzzle organized again, you need to identify the pieces that correspond to each particular puzzle, gather them together and put them back into their correct boxes. Only then can you work with these pieces and create a complete picture.

So in data, the **puzzle pieces represent the data that lives in different, separate datasets**.

Getting them organized is the aggregation process.

Then the piles of pieces that complete a single puzzle become your summary.

And finally, putting those pieces back together is like analyzing them to gain important insights.

**Data aggregation helps data analysts identify trends, make comparisons and gain insights that wouldn't be possible if each of the data elements were analyzed on its own**.

For instance, data on high school graduations for individual students can be aggregated into a single graduation rate for an entire class. Data can also be aggregated over a given time period to provide statistics, such as averages, minimums, maximums, and sums.

For example, that same yearly graduation rate data can be aggregated once again into a summary that shows us graduation rates for districts, states, and countries.

Here's another example. Let's say you had data on real estate sales in a particular neighborhood for each of the past 10 years. If you aggregated all of that data, you'd be able to discover the average price of a home in that area and how values have increased or decreased over time. **Functions are a big help in making data aggregation possible**. You'll learn how to use some of the most common ones to create your summaries soon.

In addition, we'll talk about aggregating data using something called a **subquery**. You've seen SQL in action, and you understand that ***a query is a request for information from a database***.

So ***a subquery, also called an inner or nested query, is a query within another query***. After the next several videos, you'll know how to aggregate data and understand the tools you'll be using along the way. Let's get started!

### 

### 

### 

### [PREPARE FOR VLOOKUP](https://www.coursera.org/learn/analyze-data/lecture/u5Utd/prepare-for-vlookup)

We'll prep our data for VLOOKUP, a data aggregation tool. As you learned before, **data aggregation is the process of gathering data from multiple sources in order to combine it into a single summarized collection.**

Data aggregation can give you all kinds of information about the data you are looking at. For example, in marketing, you can aggregate data from an ad campaign to see how it performed over time and for particular customers.

Travel companies use data aggregation to figure out how much their competitors charge for a certain flight, hotel room, or rental car type. Then, they can make sure they price their own products as competitively as possible. One thing these businesses all have in common is that they can use **VLOOKUP** to help them achieve these goals.

As a reminder, **VLOOKUP stands for vertical lookup. Basically, it's a function that searches for a certain value in a column to return a corresponding piece of information**. Earlier, we used VLOOKUP to take the value in one cell and search for a match in another. We were able to match a product code made up of numbers and letters that lived in one spreadsheet to the actual name of the product that lived in another. But before any of that can happen, we need to make sure our data is properly prepared. As you've heard many times, clean data is much more likely to give you accurate results.

***Let's start with the first common data-cleaning task: different data types.***

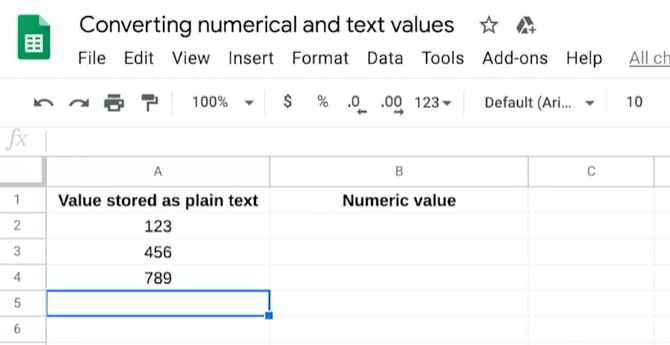
For example, a dataset might have dates formatted as numbers, or numbers represented as text strings instead of numeric values.

When data is not in a consistent format or a format that the spreadsheet application recognizes, VLOOKUP won't know what to do with that data, and it will return an error.

Earlier, you learned how to convert numbers to dates using the Format tool. **Now, let's focus on converting text to numeric values**.

To do this, you could **use the Format menu to select a type of number, but you could also use the VALUE function**. VALUE is a function that converts a text string that represents a number to a numerical value.

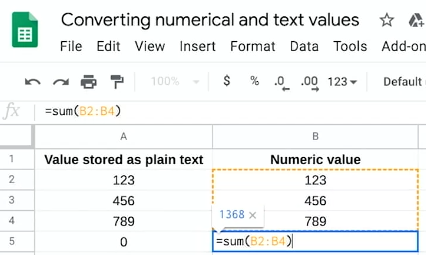
Here's an example. In this spreadsheet, the numbers in column A are currently text strings.



**We can confirm this by running a simple SUM function**. The syntax equals SUM, open parenthesis, and then the items you want to add together. Here, it's A2 to A4. The colon says we're including everything between these two references. Now you can add a closed parenthesis and press Enter, or you can click and drag on the cells you want inside the parentheses to save a little bit of time. **The result is zero. That's because the function doesn't work on text strings. But if we apply the VALUE function, it automatically converts that text to a numeric value**.

To do that, we'll type **equals VALUE, then an open parenthesis. Inside, we reference the cell whose value we want to convert, in this case A2. Now if we close the parentheses and press Enter**, you'll notice that the 1, 2, 3 is numeric. If we drag it down the column, the 4, 5, 6 and 7, 8, 9 also become numeric.

Now we can test it by **running another SUM function**. We'll type equals SUM and an open parenthesis, then B2, colon, and B4. B2, B3, and B4 are included in the sum. Close the parentheses and press Enter. **Now it shows that the total is 1,368**.

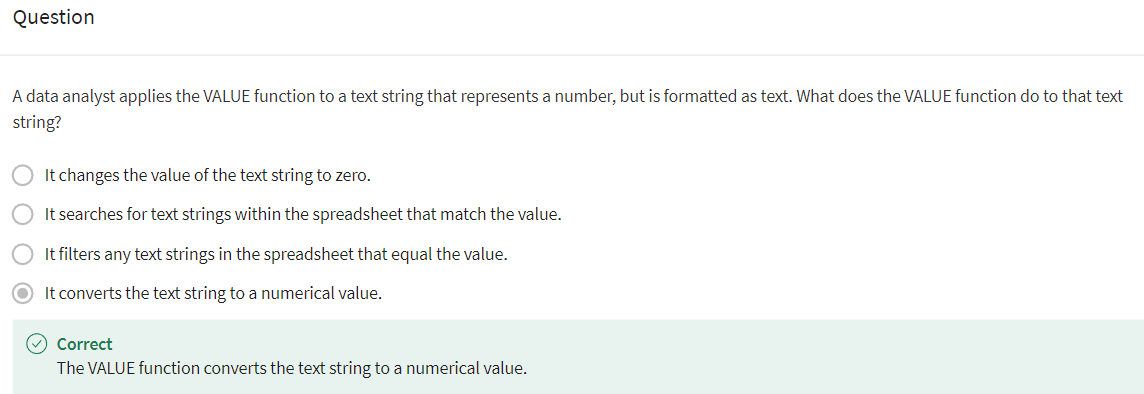


***The next common error comes from having extra spaces in your spreadsheet.***

As you've learned, when data is copied from one source to another, sometimes a few leading or trailing spaces tag along. These can cause problems when using VLOOKUP. We want to make sure to use TRIM during the data- cleaning process. **TRIM automatically deletes any extra spaces added to the cell**.

**Another typical mistake in VLOOKUP, which you can easily catch during data cleaning, are duplicates**. If there are duplicate rows in the search, it will return only the first match it finds.

**Remove duplicates is a tool that automatically searches for and eliminates duplicate entries from a spreadsheet**. Using Remove duplicates, as you saw in a video a little while ago, is a great way to get rid of duplicates and help make sure you find the right record during the lookup.

***It's always good to remember that clean data is the foundation that everything else is built on***. VLOOKUP can be a very useful data-cleaning tool. In the next video, we'll keep exploring more ways you can use VLOOKUP. See you there.

### 

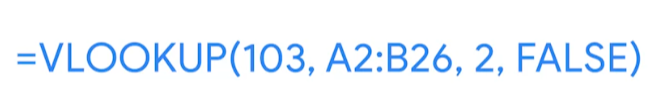
### [VLOOKUP IN ACTION](https://www.coursera.org/learn/analyze-data/lecture/kKAKe/vlookup-in-action)

We talked about VLOOKUP for data cleaning. We also discussed the importance of preparing our spreadsheet before putting VLOOKUP to use.

Now we're going to experience it in action. As a quick reminder, **VLOOKUP is a spreadsheet function that vertically searches for a certain value in a column to return a corresponding piece of information**.

Let's start with **VLOOKUP syntax**.

For example, 103 is a value to search for. A2:B26 is the range that will be searched. As you may remember, VLOOKUP will not recognize column names such as A, B, or C. **We use a number to indicate the column**.



Lastly, **FALSE tells VLOOKUP to find an exact match**.

**If this said TRUE, the function will return only a close match, which might not be what we want.**

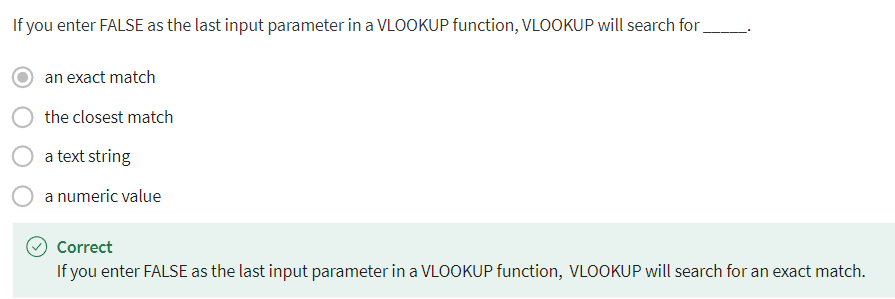
Now let's put VLOOKUP to use. **One of the most common things data analysts do with VLOOKUP is populating data in one spreadsheet from another.**

Here's an example. Let's say we're working with data that exists in two different spreadsheets, but we need information from both in order to answer our business question. **VLOOKUP can connect two sheets together on a matching column to populate one single sheet**. [VLOOKUP in Action Example](https://docs.google.com/spreadsheets/d/1f39uwSgf7N-bgc5GMAtaSefUPw0EIaAB4eKau7t0SXU/template/preview?resourcekey=0-fTi9xv6oG-88MRHHfVkHGQ)

In this spreadsheet, we have employee ID numbers and their rates of pay. In this spreadsheet we have the same employee ID numbers and how many hours each person worked. We can use VLOOKUP to search for the rate of pay from the employee rates spreadsheet and add it to the employee hours spreadsheet.

The formula equals VLOOKUP open parentheses, then A2, which is the first employee ID number and the employee hours spreadsheet. Next, we add a comma, the name of the spreadsheet we want to search in, employee rates. Be sure to put single quotation marks around the spreadsheet name and add an exclamation point after it. This is the way to reference the other spreadsheet. Next, we add the range, which is A2 through B5. As you saw in a previous video, we can also choose to add dollar signs to lock the range with absolute cell references. **This prevents them from changing when copying the formula to other cells**. Add another comma, then a two. **The two indicate that we want to search for a match in the second column**, column B for rate of pay. Finally, one more comma and we add false to look up an exact match.

Drag the formula down the column and now we can use a simple multiplication formula to calculate each person's paycheck by multiplying hours worked by our newly created pay rate column. Great work. In an upcoming reading, you'll learn even more about VLOOKUP and access some helpful VLOOKUP reminders and resources. VLOOKUP is one of the more complicated functions , so keep practicing.



### [IDENTIFY AND FIX COMMON VLOOKUP ERRORS](https://www.coursera.org/learn/analyze-data/lecture/D8tQg/identify-and-fix-common-vlookup-errors)

When people start out in data analytics, they often think that those of us who've been in the field for a while know everything. But trust me, we're all still figuring things out. **And a lot of the time that means troubleshooting.**

***Troubleshooting has to do with asking the right questions***, and that's what we'll focus on in this video. We'll learn how you can use troubleshooting to solve all kinds of problems. To do this, we'll need to **talk about some of the limitations of VLOOKUP and then practice fixing some of the most common problems that data analysts face**.

***Some of the troubleshooting questions I like to ask myself:***

**How should I prioritize these issues?** Trying to solve lots of problems all at once can feel overwhelming. I find it helps when you take things one at a time.

Next I ask, **In a single sentence, what's the issue I'm facing?** This helps to clarify what's really going on, so I don't get bogged down with extra details. After all, if you don't have a clear objective before looking at the data, you can find just about anything. It's always best to start with your own clear understanding of the situation. Then let the data tell you if you're on the right track or not.

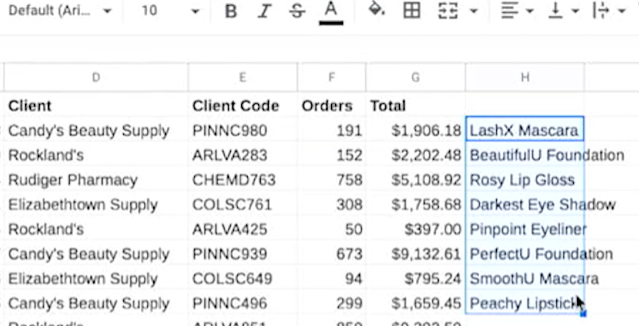
The next question I ask myself is, **What resources can help me solve the problem?** The internet is one of the best resources out there. If you have a question, chances are thousands of others run into exactly the same thing. So a quick search can be really helpful. And it's good to remember that people are resources, too. Don't be afraid to ask questions. Not only is it a great way to learn, it can also help you build strong relationships with your colleagues.

And a final important question I think about: **How can I stop this problem from happening in the future?** If a new procedure or guideline can stop the same issue from popping up again, that's a great time-saver. All right. Let's start by noting that VLOOKUP only returns the first match it finds, even if there are lots of possible matches.

**Something else to keep in mind is that VLOOKUP can only return a value from the data to the right. It can't look left.**

Good news. **There's a simple solution**. Data analysts usually get around the problem by copying and pasting a column to the left of the data they want to look at. This way, the lookup value is in the leftmost column and the data they want is to the right of it.

Here's another problem I see a lot. Let's say the first few rows of a VLOOKUP have returned the correct result. But when you drive the function down the column, problems start popping up.



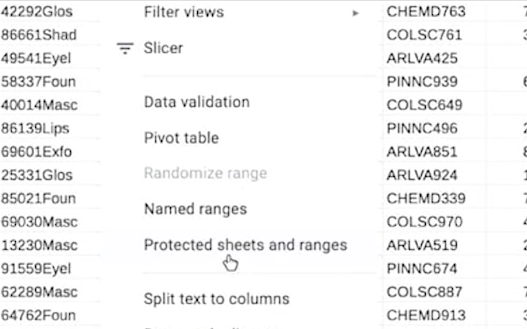
This is probably because the table array part of the function hasn't been locked or made absolute.

**An *absolute reference* is a reference that is locked so that rows and columns won't change when copied**. You can fix this issue by wrapping the table array in dollar signs.

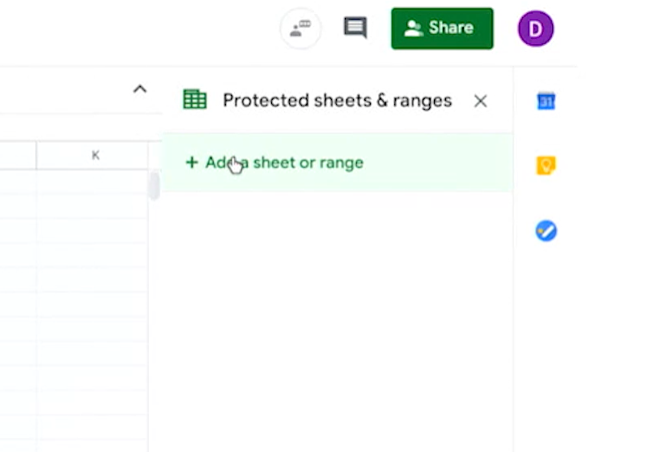


As you learned a while back, the dollar sign controls how the reference will be updated. They make sure that the corresponding part of the reference doesn't change. Something else that can throw off your VLOOKUP results are version control issues. In other words, a function worked perfectly at first, but then something in the spreadsheet it was referencing changed. For example, maybe a user inserted a column. So now the columns in your function no longer direct VLOOKUP to the right place. When something like this happens, it'll return an incorrect value. There are a few actions data analysts can take to ensure this doesn't happen. First, lock the spreadsheet. This stops other people from making changes.

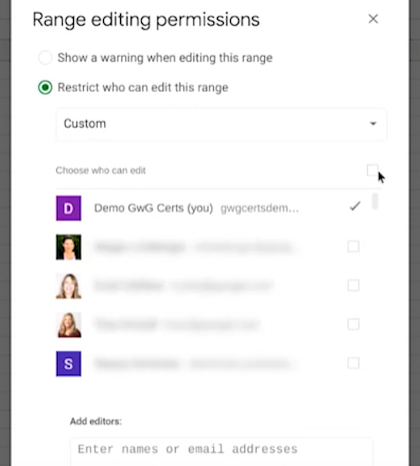
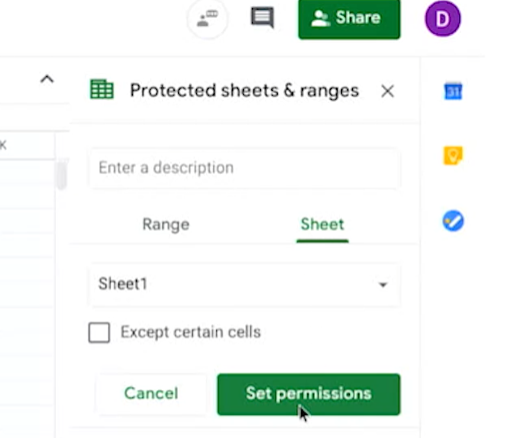
To do this in Sheets, select Data, then Protected sheets and ranges.



In other spreadsheet applications, there are other tools that do the same thing. Next, choose what you want to protect.



In this case, we want to protect the entire sheet. Then you can set permissions to either show a warning or restrict who can edit. Choose only you, then Done.



But keep in mind, **there will be times when other people need to work in the spreadshee**t, so locking them out might make you pretty unpopular with your coworkers. When that's the case, ***you can use MATCH, which is a function used to locate the position of a specific lookup value and can help you with version control***. We won't get into that right now, but just know that it's an option in case you ever need it.

**The final problem we'll talk about has to do with exact and approximate matching.**

When using VLOOKUP, you're likely to get different results, depending on whether you enter the word TRUE or FALSE within your function.

***TRUE tells VLOOKUP to look for approximate matches, and FALSE tells VLOOKUP to look for exact matches***. So if a function looks like this,

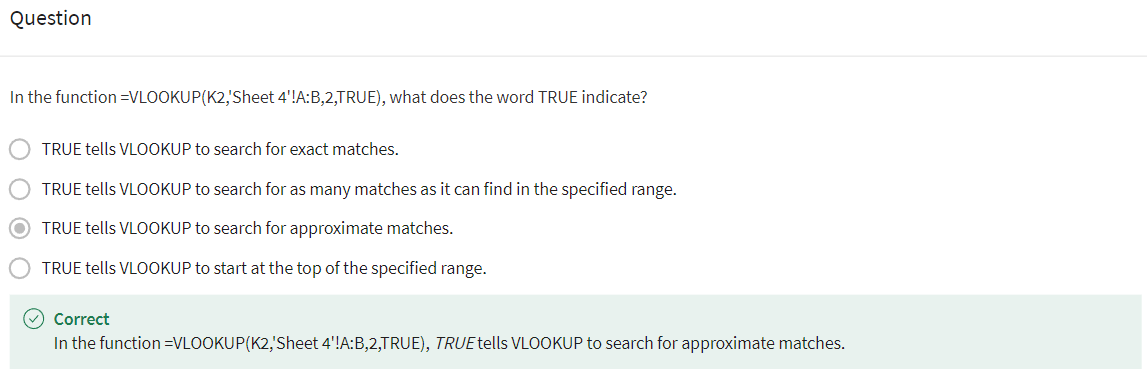
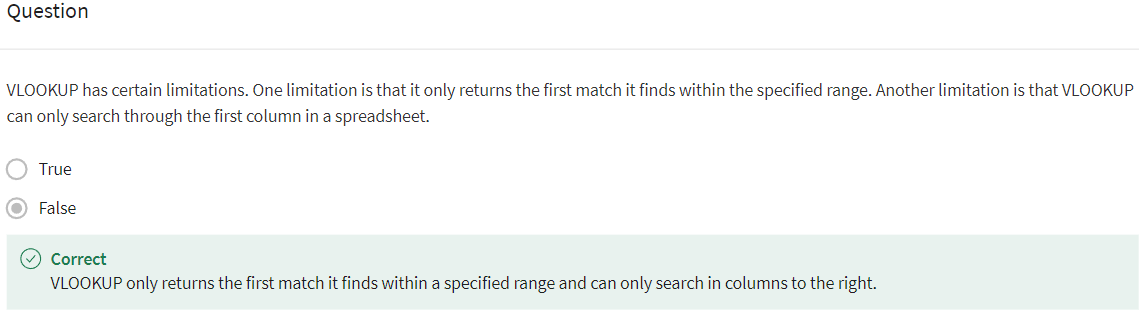


it's telling VLOOKUP to find the closest match to the text or number we're looking for. **It's important to know that VLOOKUP starts at the top of a specified range and searches downward vertically in each cell to find the right value. It stops searching when it finds any value that's greater than or equal to the lookup value**.

**That's why data analysts typically use FALSE,** like this. **That way VLOOKUP only returns the exact match to what you've entered in the lookup value**.

**VLOOKUP is one of the most popular lookup and reference functions in spreadsheets. It's also one of the trickiest**.

Coming up, you'll learn about more of these common challenges. Everything you learn will help you run into fewer problems when you start using VLOOKUP as a future data analyst.



### 

### [VLOOKUP CORE CONCEPTS](https://www.coursera.org/learn/analyze-data/supplement/SNmqP/vlookup-core-concepts)

Spreadsheet functions can be used to quickly find information and perform calculations using specific values. **VLOOKUP**, ***or Vertical Lookup, is one such function that vertically searches for a certain value in a column to return a corresponding piece of information***. In this reading, you’ll examine the intricacies of this extremely useful function so you understand how it works when you use it to analyze data.

## **VLOOKUP functionality**

**VLOOKUP** searches for a search term, called a **search\_key**, in one column of a spreadsheet. When the **search\_key** is found, the function returns the data from another column of the row from which it was located. **VLOOKUP** returns only the value that corresponds to the first item it matches. So, if there are multiple matching values, the spreadsheet will return only data about the first one.

## **VLOOKUP use cases**

Here are two common reasons why you might use **VLOOKUP**:

* **Populating data in a spreadsheet.** Perhaps a store manager is tracking incoming shipments before a busy holiday. They could use **VLOOKUP** to look up product ID codes in a product spreadsheet and retrieve the corresponding product information from another spreadsheet. This would help the manager know how many stock clerks they need to schedule to work when the shipments arrive.
* **Merging data from one spreadsheet with data in another.** If a teacher keeps one spreadsheet for student grades and another for attendance, they could use **VLOOKUP** to combine the spreadsheets. That way, they could search for a particular student in the attendance sheet, and **VLOOKUP** would pull the corresponding attendance record into the grades spreadsheet.

## **VLOOKUP syntax**

**VLOOKUP** is available in both Microsoft Excel and Google Sheets. Here, you’ll explore its syntax in Google Sheets. Refer to the resources at the end of this reading for more information about **VLOOKUP** in Microsoft Excel.

**VLOOKUP**’s syntax is:



The following sections explain each of the four parts of the syntax.

### **search\_key**

This is the value the function will search for. It can be a number, text string, or cell reference.

### **range**

This is the range of cells over which the function will search and return information. The first column in the **range** is searched. When the search key is found, the index from that row is returned.

For example, if you search for the **search\_key** in column B and return the data from column D, the range would need to include columns B through D, such as the range B2:D10. If you specified a range of A2:D10, the function would search for the search term in column A.

The **search\_key** must be to the left of the information you want the function to return. This may require you to move columns around before you use **VLOOKUP**. For example, if you plan to search for the **search\_key** column D, but the information you want the function to return is in column A, you must rearrange your columns before using **VLOOKUP**.

### **index**

This is the position of the column that contains the data to be returned. The first column in the range is column number 1, and each column is numbered sequentially to the right.

For example, if the range is B2:D10 and you want to return a value from column D, the index number would be 3. If the index is not between 1 and the number of columns in range, the error message **#VALUE!** will be returned.

### **is\_sorted**

This indicates whether to return an approximate or exact match. For example, if you’re searching for Google, then google would not count as a match.

* To return an exact match, set **is\_sorted** to **FALSE**. This is recommended.
* To return an approximate match, set **is\_sorted** to **TRUE**. The nearest match (less than or equal to the **search\_key**) is returned. To use this option to obtain accurate results, you must sort your data in ascending order. But, you could still find a value.
* If neither **TRUE** nor **FALSE** are selected, the function will default to **TRUE**.

## **The #N/A error**

**#N/A** indicates that a matching value can't be returned because no matches were found.

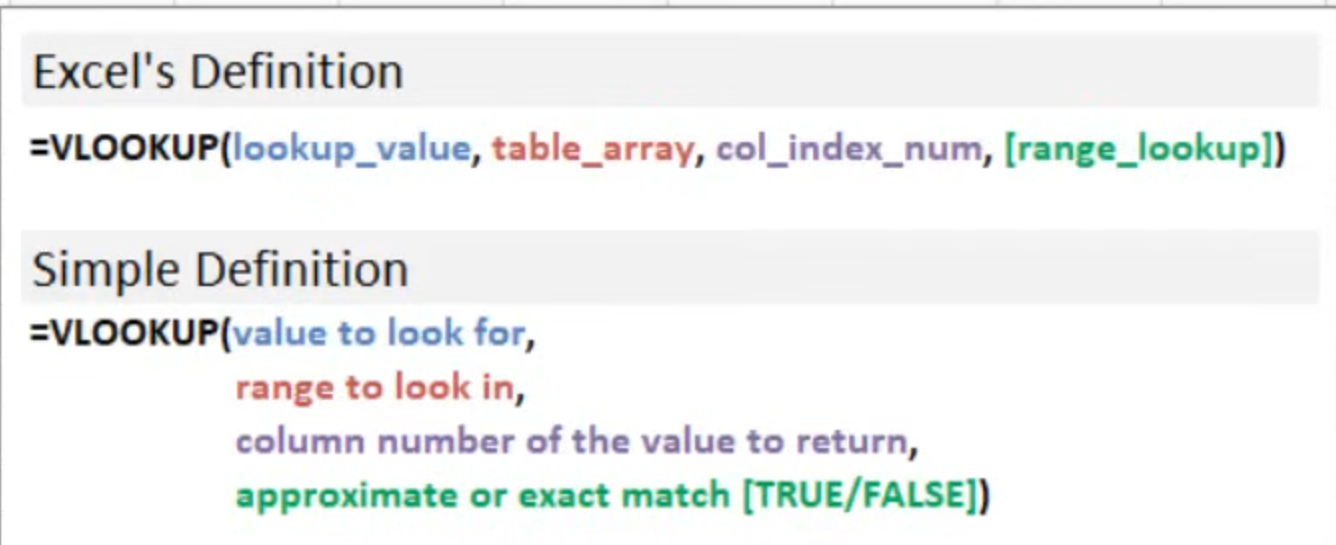
## **Key takeaways**

Use **VLOOKUP** to search for a value in a column and return a corresponding piece of information. It’s a very useful tool for data professionals, as it enables them to combine data from multiple sources and find information quickly. Keep in mind that the column that matches the **search\_key** in a **VLOOKUP** formula should be on the left side of the data. The range must include both the column being searched and the column that contains the information being returned. **TRUE** means an approximate match, and **FALSE** means an exact match on the **search\_key**.

## **VLOOKUP resources for Microsoft Excel**

**VLOOKUP** may slightly differ in Microsoft Excel, but the overall concepts can still be generally applied. Refer to the following resources if you’re working with Excel:

* [How to use VLOOKUP in Excel](https://support.microsoft.com/en-us/office/vlookup-function-0bbc8083-26fe-4963-8ab8-93a18ad188a1): This tutorial includes a video to help you get a general understanding of how the **VLOOKUP** function works in Excel, as well as practical examples to look through.
* [VLOOKUP in Excel tutorial](https://www.youtube.com/watch?v=d3BYVQ6xIE4): Follow along in this video lesson and learn how to write a **VLOOKUP** formula in Excel and master time-saving useful tips and tricks.
* [23 things you should know about VLOOKUP in Excel](https://exceljet.net/things-you-should-know-about-vlookup): Explore this list of **VLOOKUP** facts, common challenges and their solutions.
* [How to use Excel's VLOOKUP function](https://edu.gcfglobal.org/en/excel-tips/how-to-use-excels-vlookup-function/1/): This article shares a specific example about applying **VLOOKUP** in your searches.
* [VLOOKUP in Excel vs Google Sheets](https://infoinspired.com/sheets-vs-excel-formula/vlookup-formula-in-excel-and-google-sheets/): This guide offers a **VLOOKUP** comparison of Excel and Google Sheets.



OBS: Quando estiver utilizando VLOOKUP para lockar os intervalos você pode escrever o intervalo e depois apertar F4, isso gerará os $ lockando o intervalo.

### 

### [HANDS-ON ACTIVITY: USE VLOOKUP TO PERFORM A TASK](https://www.coursera.org/learn/analyze-data/quiz/kCl1b/hands-on-activity-use-vlookup-to-perform-a-task)



## **Activity Overview**

****

You’ve been learning about **VLOOKUP**, a spreadsheet function that vertically searches for a certain value in a column to return a corresponding piece of information. In this activity, you’ll practice cleaning data and using **VLOOKUP** to consolidate information between two spreadsheet tabs. You’ll complete your analysis by creating a pivot table.

By the time you complete this activity, you will be able to use **VLOOKUP** to find information in one sheet and add it to the correct row in another spreadsheet. This is an important skill that can help you work with large datasets in your career as a data analyst.



### 

### **Scenario**

Review the following scenario. Then complete the step-by-step instructions.

You’re the payroll manager at an accounting firm. To calculate payroll, you need to know how many hours each of your employees worked and their hourly rate of pay. This is easy to do manually in a small spreadsheet, but it becomes more difficult as the amount of information grows or is spread across multiple spreadsheets. You’d like to use the **VLOOKUP** function as a way to automate the information-gathering. So, follow the steps below to calculate the total weekly pay for your employees.

### 

### **Step-By-Step Instructions**

Follow the instructions to complete each step of the activity. Then answer the questions at the end of the activity before going to the next course item.

### 

### **Step 1: Access the template**

To get started, access the **VLOOKUP** Practice Worksheet. Select the link to the worksheet below and select USE TEMPLATE to create a copy. If you don’t have a Google account, you can download the **VLOOKUP** Practice Sheet directly from the attachment below.

Link to the worksheet: [VLOOKUP Practice Sheet](https://docs.google.com/spreadsheets/d/1Js6kRVYy6Nx6VENibaX9dmNOAP_7CXWuOqUjUz9cBtA/template/preview)

### **Step 2: Prepare the data**

Sheet1 of the **VLOOKUP** Practice Sheet contains a timesheet of hours worked by several employees. However, this data has not been cleaned. You'll create a clean version of the table in Sheet1 so you can manipulate the data without changing the data from the original table. Then, you’ll combine data from two sheets in the **VLOOKUP** Practice Sheet spreadsheet (Sheet1 and Sheet2) using the **VLOOKUP** function.

### 

### **Step 3: Label the columns**

Working with data gets messy quickly, and it’s important to keep track of what your columns mean. First, add labels to the columns in your new table to help keep your data organized.

Add the following labels to the Sheet1:

1. In cell B14, enter: Names.

2. In cells C14 to H14, enter: 1/1/2020, 1/2/2020, 1/3/2020, 1/4/2020, 1/5/2020, and 1/6/2020.

3. In cell I14, enter: Hours.

4. In cell J14, enter: Pay Rate.

5. In cell K14, enter: Total Pay.

### 

### **Step 4: Clean the data**

Some of the employee names in column B have extra spaces. Use the following steps to remove the extra spaces and clean your data.

1. In cell B15, enter **=TRIM(B2)**.

2. Select and drag the fill handle to cell B19, then release it. This populates the names, with extra spaces removed, in these cells.

### 

### 

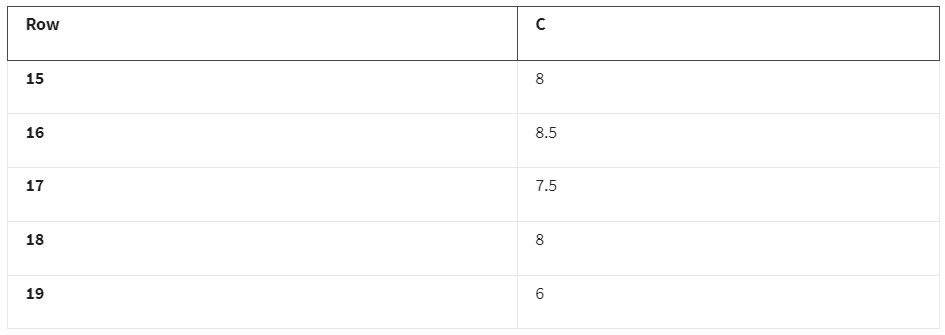
### **Step 5: Populate the daily employee hours**

Next, move employee hours to your new table with cleaned employee names. Perform the following steps to populate the daily hours for the employees:

1. In cell C15, type **=value(C2)**.

2. Select and drag the fill handle to cell C19. This populates the hours for the other employees.

Your new table should contain the following data in rows 15-19 of column C:



Your table should have the values 8, 8.5, 7.5, 8, and 6 in the cells C15 through C19 respectively.

Use the populated cells from C15 through C19 to populate the remaining hours needed for each employee. To do this, perform the following steps:

3. Select and drag the fill handle for cell C15 to cell H15. This populates the remaining hours for Daniel Chan. You should see the values 8, 8, 8.5, 7, 5, and 2.5 in cells C15 through H15.

4. Select and drag the fill handle for cell C16 to cell H16. This populates the remaining hours for Dana Ali. You should see the values 8.5, 7, 8, 8, 9, and 5.5 in cells C16 through H16.

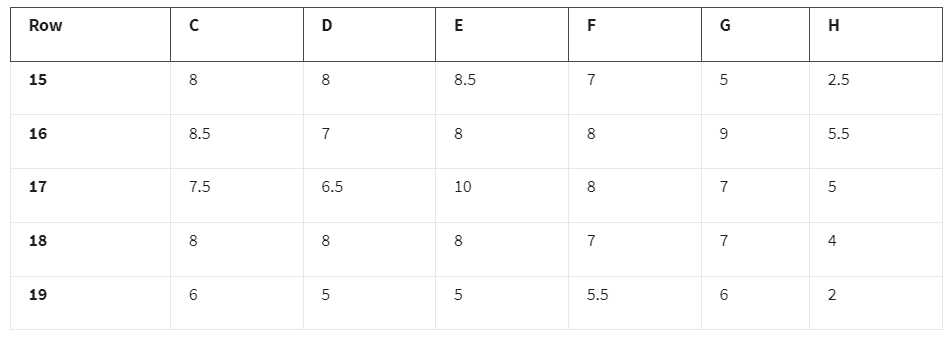
5. Repeat this process in rows 17, 18, and 19 for the remaining employees.

a. In row 17, you should see the values 7.5, 6.5, 10, 8, 7, and 5 in cells C17 through H17.

b. In row 18, you should see the values 8, 8, 8, 7, 7, and 4 in cells C18 through H18.

c. In row 19, you should see the values 6, 5, 5, 5.5, 6, and 2 in cells C19 through H19.

Verify that your spreadsheet contains the following data:



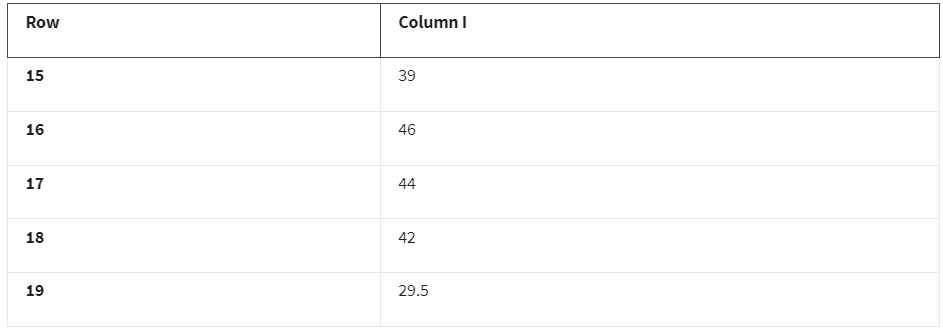
**Step 6: Sum the total hours for each employee**

Fill in the Hours column for the employees.

1. In cell I15, enter **=sum(C15:H15)**.

2. Select and drag the fill handle for cell I15 to cell H15. This populates the sums for the remaining employees.

Column I in rows 15–19 should contain the following data:



Your table should have the values 39, 46, 44, 42, and 29.5 in the cells I15 through I19 respectively.

### 

### **Step 7: Import pay rate data**

You keep track of your employee’s hourly pay rate in Sheet2 of the **VLOOKUP** Practice Sheet. This sheet also includes employee ID, date of hire (DOH), and employee status.

Use **VLOOKUP** to import pay rate data from Sheet2 into Sheet1.

1. In cell J15 on Sheet1, enter: **=VLOOKUP(A2, Sheet2!$A$2:$D$6, 4, false)**.

Consider the syntax for this **VLOOKUP** function:

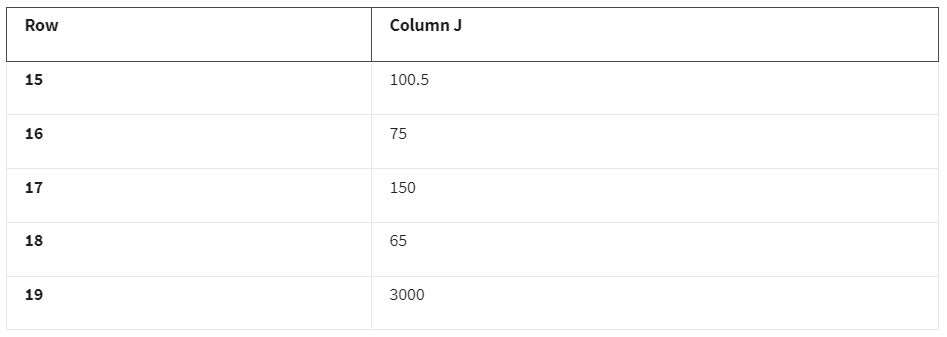
"A2" refers to cell A2 in Sheet1.

Note: In Sheet2, the rate of pay and related fields are referenced by ID instead of employee name. You need to use employee ID to import the pay rate from Sheet2.

* "**Sheet2!**" refers to the sheet from which you want to access the data.
* "**$A$2:$D$6**" to the range of cells that make up the table array. The "**$**" placed in front of the column tabs and cell numbers locks the formula so that it can be copied by dragging cell J15 down to cell J19 to import the pay rate for the other employees.
* The "**4**" refers to the column from which the returned value will come. The "**4**" means that the returned value will come from the 4th column in the selected range.
* "**false**" signifies that you want an exact, character-for-character match to the lookup value. Using "**true**" will return an approximate match (or the closest match available) for the lookup value.

2. Select and drag the fill handle for cell J15 to cell J19. This populates the pay rate for the remaining employees.

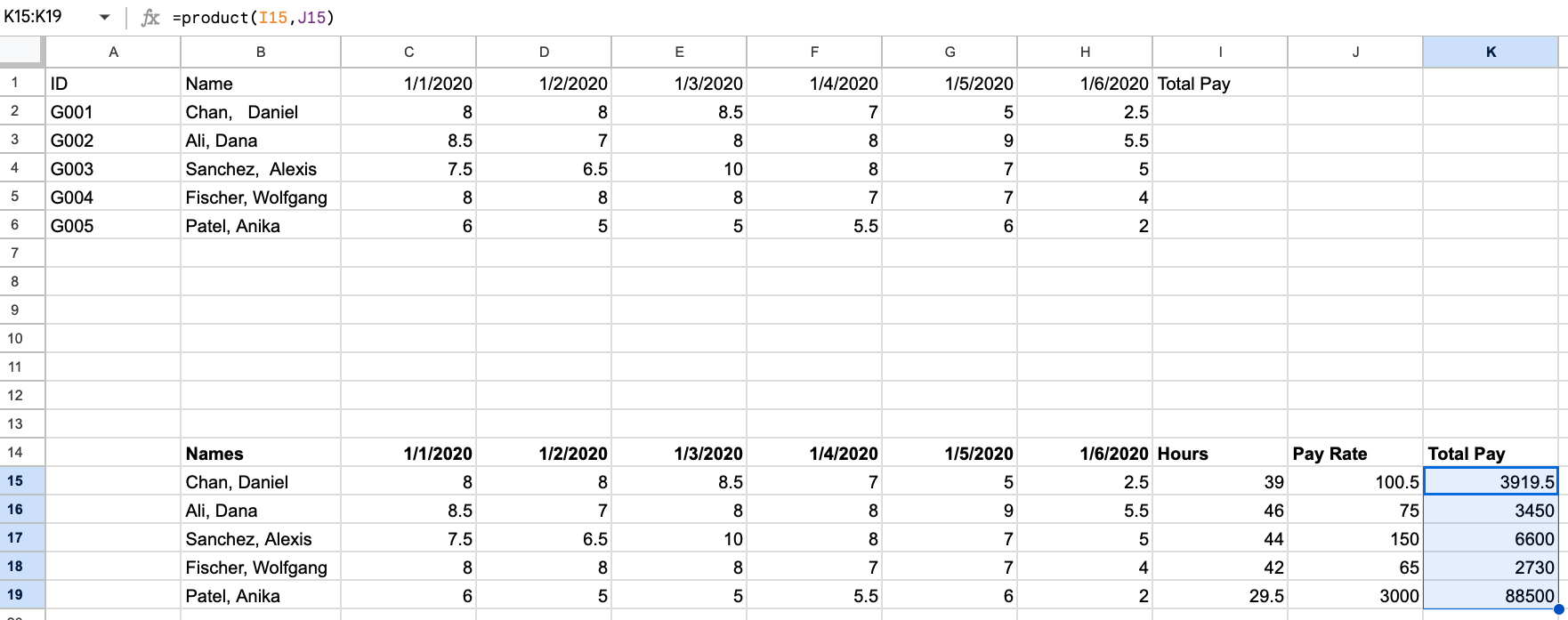
Rows 15–19 in column J should contain the following data:



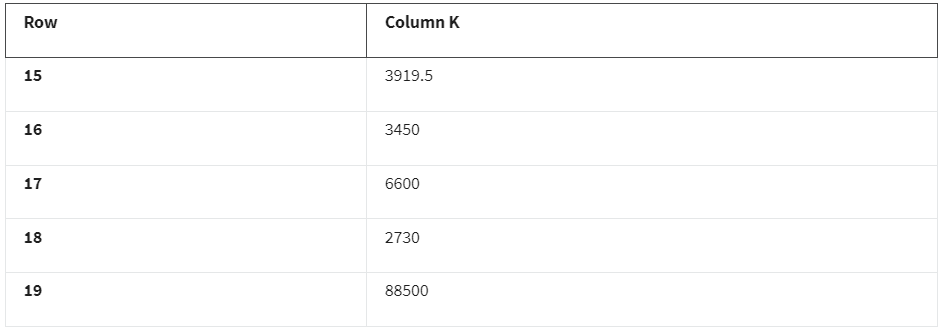
Your table should have the values 100.5, 75, 150, 65, and 3,000 in the cells J15 through J19 respectively.

3. In cell K15, enter: **=PRODUCT(I15, J15)**to calculate the total pay.

4. Select and drag the fill handle of cell K15 to cell K19 to populate the total pay for the remaining employees.



Rows 15–19 in column K should contain the following data for Total Pay:



Your table should have the values 3919.5, 3450, 6600, 2730, and 88500 in the cells K15 through K19 respectively.

### 

### 

### **Step 8: Create a pivot table**

Now that the data is clean and includes the pay rate information, you can create a pivot table. This makes it easier to quickly identify trends and patterns and generate reports without having to search the raw data. This section demonstrates how to create a pivot table in Google Sheets. If you are using Excel, follow the [documentation for how to manually create a pivot table in Excel](https://support.microsoft.com/en-us/office/create-a-pivottable-to-analyze-worksheet-data-a9a84538-bfe9-40a9-a8e9-f99134456576).

To create a table for data in cells B14:K19:

1. Select the data in cells B14:K19.

2. Select Insert from the menu, then select Pivot Table.

3. From the pop-up window, select New Sheet, then select Create.

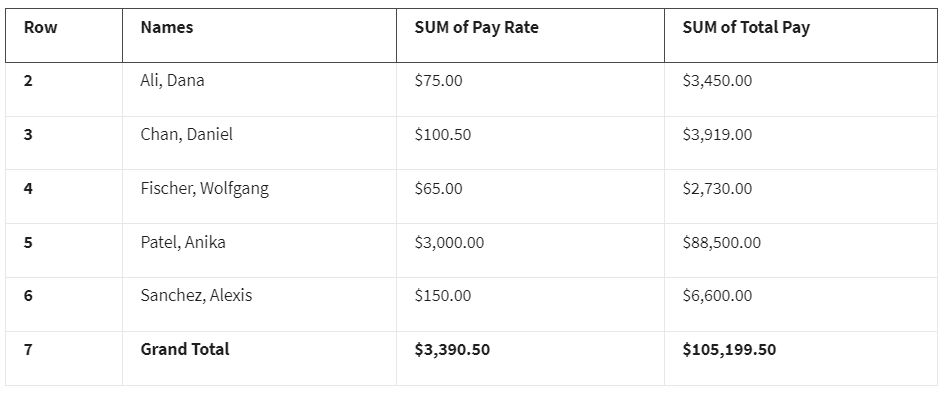
A new tab titled Pivot Table 1 will appear between Sheet1 and Sheet2. A Pivot table editor will pop up on the screen. Use this editor to create a pivot table that contains each employee’s name, pay rate, and total pay:

1. Select Add for Rows. Then, select Names from the dropdown options.

2. Select Add for Values. Then, select Pay Rate from the dropdown options.

3. Select Add for Values again. Then, select Total Pay from the dropdown options.

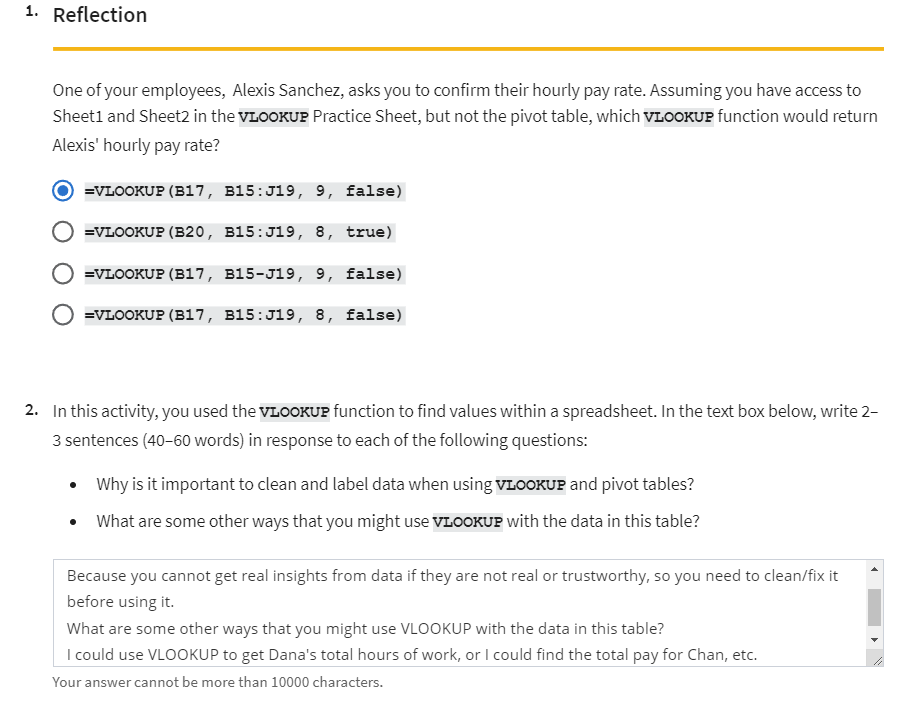
4. Select cells B2 through C6, then select the $ symbol from the toolbar to reformat these cells as currency.

Rows 1-7 in columns A, B, and C in the sheet Pivot Table 1 should contain the following data:

Your table should have the following values:

In row 2, you see the name Ali, Dana with a pay rate of $75.00 and a total pay of $3,450.00. In row 3, you see the name Chan, Daniel with a pay rate of $100.50 and a total pay of $3,919.00. In row 4, you see the name Fischer, Wolfgang with a pay rate of $65.00 and a total pay of $2,730.00. In row 5, you see the name Patel, Anika with a pay rate of $3,000.00 and a total pay of $88,500.00. In row 6, you see the name Sanchez, Alexis with a pay rate of $150.00 and a total pay of $6,600.00. In row 7 are the grand totals, which include the total combined pay rate of $3,390.50 and the total combined pay of $105,199.50.

Congratulations! You have now cleaned and labeled your data, used **VLOOKUP** to import data from another spreadsheet, and created a pivot table. Now you’ll be able to easily complete payroll for your employees.



### [TEST YOUR KNOWLEDGE ON VLOOKUP](https://www.coursera.org/learn/analyze-data/quiz/BcibG/test-your-knowledge-on-vlookup)

## USE JOINS TO AGGREGATE DATA IN SQL

### [UPLOAD THE EMPLOYEE DATASET TO BIGQUERY](https://www.coursera.org/learn/analyze-data/supplement/13KQO/upload-the-employee-dataset-to-bigquery)

This reading outlines the steps you need to perform before watching the video and following along in the step-by-step guide.

**What you will need**

Download the two .csv files from the attachments below:

[Employees Table - Understanding JOINS](https://d3c33hcgiwev3.cloudfront.net/TMwinKTQQ2aYTZGdcL0Fog_84586bd2265a4888af22e8060747c8e1_Employees-Table---Understanding-JOINS.csv?Expires=1712880000&Signature=BAUd1IHLnwUWk9SbHKmzzZ66-hCAC1Wtwa4N1viivsAGzLRzfjm5d7PdgjTOhy-CVqjmC2YPFtBhXNX0gxNgW-tx9KZ~nSXH0Z9sTbq5oHSBvtGDfe4t0p-duWjCzRrrFVyT0KbNmGIIKMaBL2qzWnWZk9I9WpOMHBzkkuTDws0_&Key-Pair-Id=APKAJLTNE6QMUY6HBC5A)

[Departments Table - Understanding JOINS](https://d3c33hcgiwev3.cloudfront.net/9vVSOuERRjusAUScVM-rOw_d98f88a17db84f198dee412210c13ae1_Departments-Table---Understanding-JOINS.csv?Expires=1712880000&Signature=JzSNTPx2lCwwqdM9AZ1JpFiSUmgvB1EmKPYMjLgXlQwHiWlSUMkUkFL~MSGTUhPyvROag5YM024SZItlyokuOR4FmmwcttiW06Xxljod73e9I5XvvHgV~1NXi6hu9sMflfQ~eVjsSPz9bfOfkMGssIsNsW462rc3eAWU7qcyUgQ_&Key-Pair-Id=APKAJLTNE6QMUY6HBC5A)

Then, log in to your BigQuery account and follow along with this reading to upload the employee data provided as two new tables.

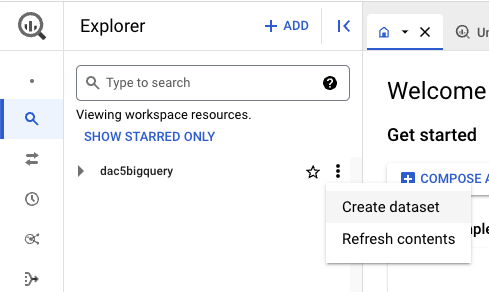
## 

## **Prepare for the next video**

### **Create a new dataset**

1. Open your BigQuery console and select the project to which you’ll upload data. For the purpose of this reading, the example project is named **dac5bigquery**. Your project will have a different name.

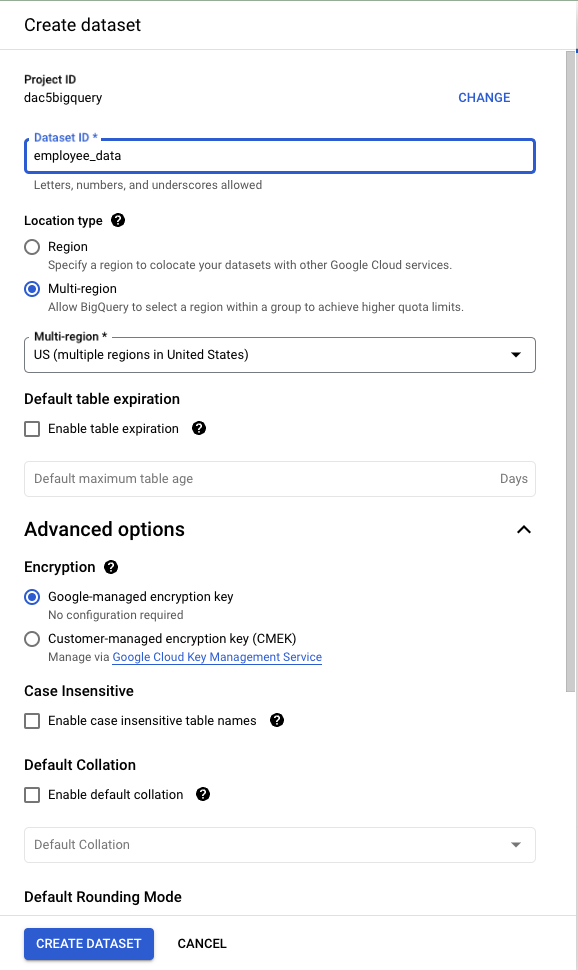
2. From the **Explorer** pane, select the **Actions** icon (the three vertical dots) next to your project name. Then select **Create dataset**.



3. In the **Create dataset** window, enter **employee\_data** for the **Dataset ID**.

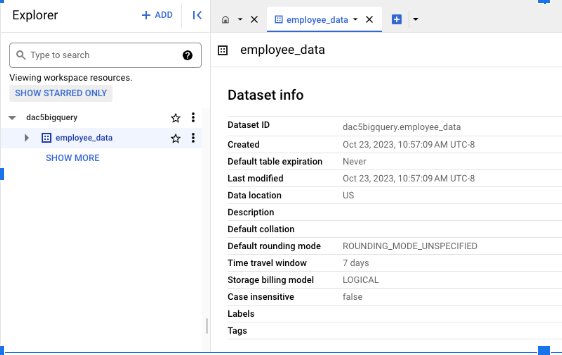
4. Make sure the **Location type** is set to **Multi-region** with **US(multiple regions in the United States)** selected.

5. Leave the **Advanced options** set to their default settings.



6. Select **CREATE DATASET** to add the dataset to your project. It will now appear under your project in the **Explorer** pane. If you don’t see the new dataset listed, select the arrow next to your project in the **Explorer** pane to expand its contents.

7. In the **Explorer** pane, select the **employee\_data** dataset you just created. The **Dataset info** window opens in the **Details** pane.



### **Create the employees table**

Create a table for the employees within your employee\_data dataset. To do this:

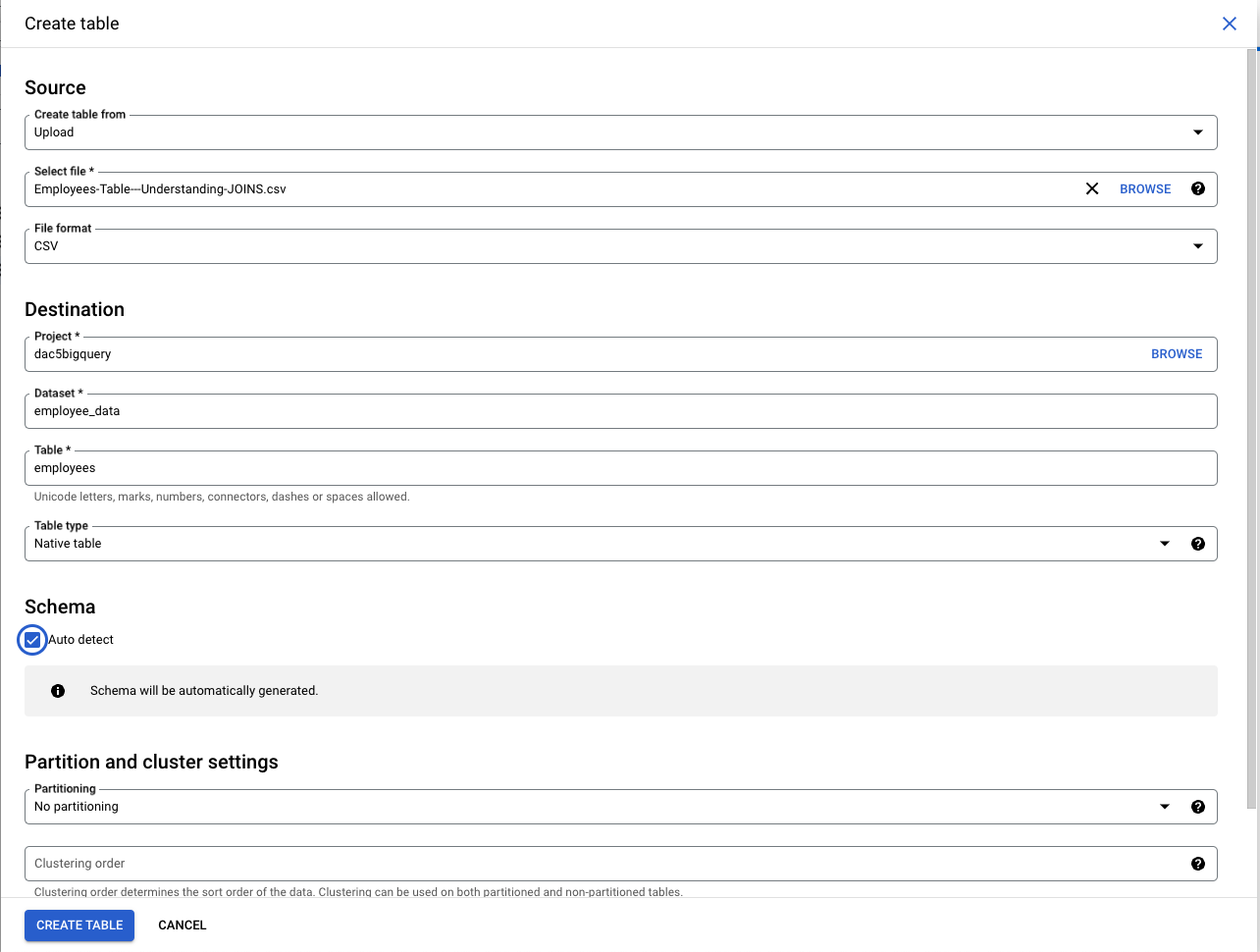
1. Select **+ CREATE TABLE** from the options in the **Details** pane. Under **Source**:

* From the dropdown in the field **Create table from**, select **Upload**.
* From the **Select file** field, select **BROWSE**. Then, find and select the [Employees Table - Understanding JOINS](https://d3c33hcgiwev3.cloudfront.net/TMwinKTQQ2aYTZGdcL0Fog_84586bd2265a4888af22e8060747c8e1_Employees-Table---Understanding-JOINS.csv?Expires=1700006400&Signature=XeqsCQhRu8D7cS7Ew8kaS3ySfbc2aYOuk~bTZZ8eBdF07KQtiMu4sAgRB4V1xboPLvCry4H61bXFNjqBT1PmbKFo8dgzPbmOUsS0msC6Fca6c-gj95HVM1PJq5pRctkQiC8OaBKml3T9KRs0lctwoqeJI-HKUo5e7hs2dF-q~EA_&Key-Pair-Id=APKAJLTNE6QMUY6HBC5A).csv File you downloaded previously.
* The file format should be automatically detected after selecting the Employee Table .csv file. If it’s not, select **CSV** from the **File format** drop-down.

2. Under Destination:

* Enter **employees** in the **Table** field.
* **Project**, **Dataset**, and **Table type** will be automatically filled. You do not need to update these fields.

3. Under **Schema**, select the **Auto detect** checkbox.



4. Select **CREATE TABLE**. The employees table will be nested under **employee\_data** in the **Explorer** pane.



### **Create a departments table**

Next, create a table for the departments within the employee\_data dataset. To do this:

1. Select the **employee\_data** dataset from the **Explorer** panel.

2. Select **+ CREATE TABLE** from the options located in the main editor window.

3. Under **Source**:

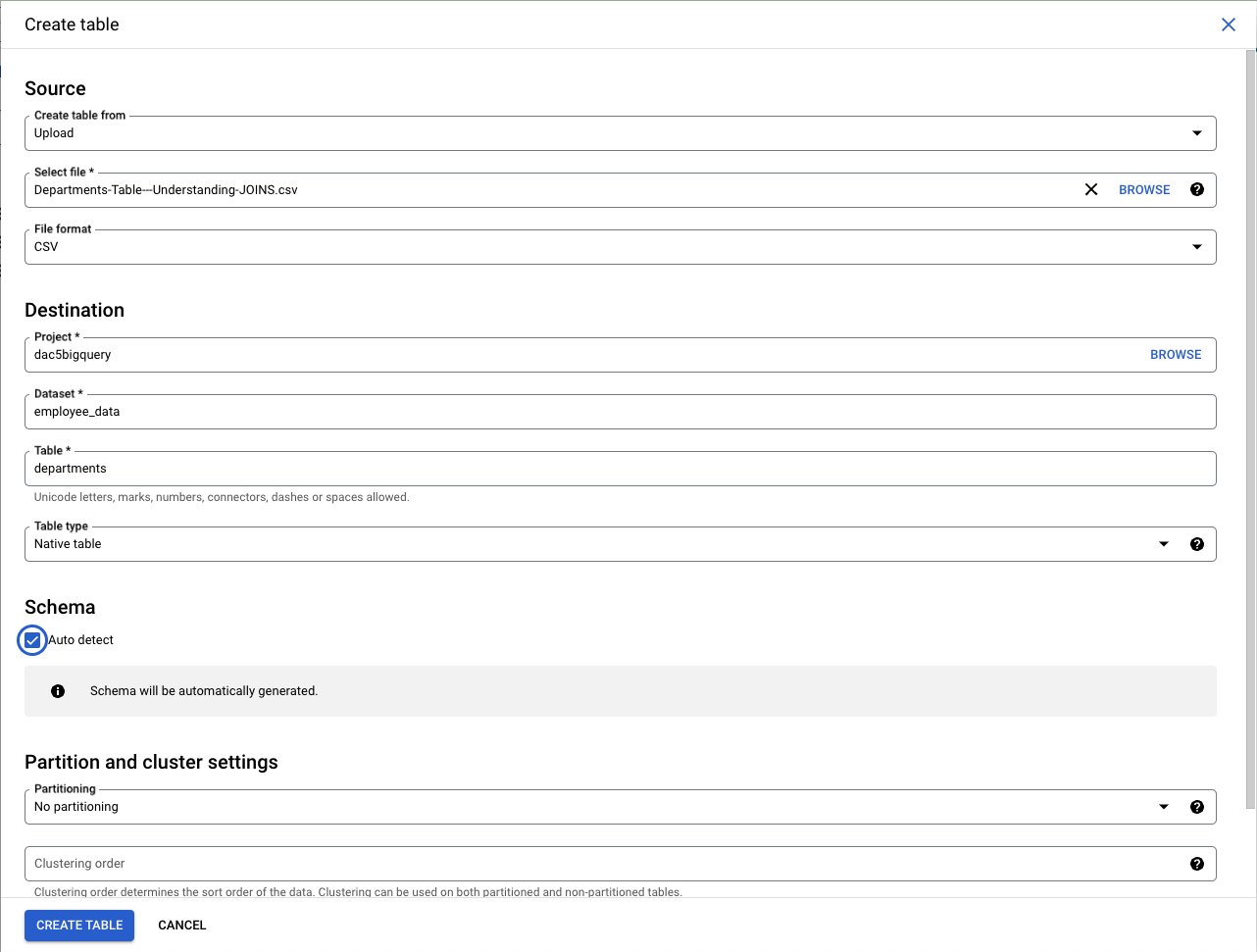
* From the dropdown in the field **Create table from**, select **Upload** from the list of options.
* From the **Select file** field, select **BROWSE**. Find and select the [Departments Table - Understanding JOINS](https://d3c33hcgiwev3.cloudfront.net/9vVSOuERRjusAUScVM-rOw_d98f88a17db84f198dee412210c13ae1_Departments-Table---Understanding-JOINS.csv?Expires=1700006400&Signature=BTJ2YLesayzmJfmfNXNbbpPk1iw5Gtn-S5VuVwvjSBrHHVcx43MzcGPWP6skOxWPQZATx2JVKw2x0aR9HKbxe80VLIK9enqXKETlIOcPYpxjDw52KFdA-Ugi4gpqegDFSl8E1bqumIk-tPxDwRe9SKnBDAeTxXGgroVc~ssKB88_&Key-Pair-Id=APKAJLTNE6QMUY6HBC5A).csv File you downloaded previously.

4. The file format may be automatically detected after selecting the Departments Table .csv file. If it’s not, select **CSV** from the **File format** drop-down.

5. Under the **Destination** section:

* Enter **departments** in the **Table** field.
* **Project**, **Dataset**, and **Table type** will be automatically filled out so you do not need to update these fields.

6. Under **Schema**, select the **Auto detect** checkbox.



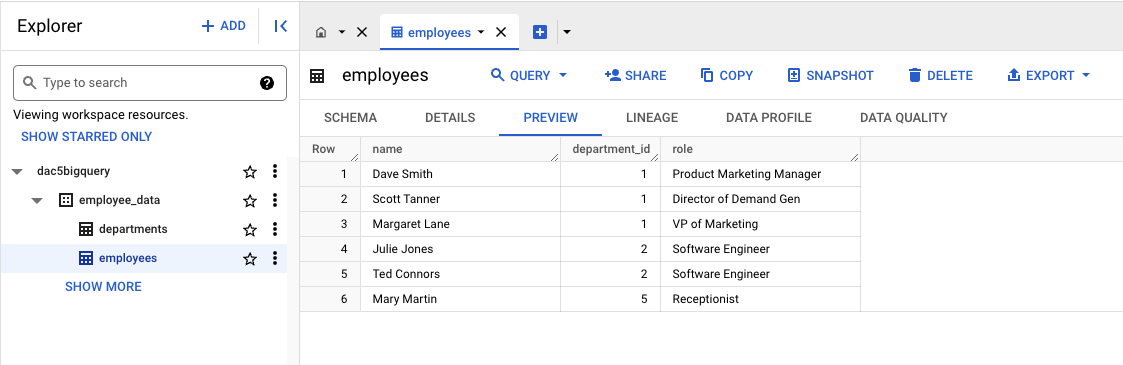
7. Select **CREATE TABLE** in the **Create Table** window. The **departments** table will appear under the **employee\_data** dataset listed within your project.

### **Verify the data**

Make sure that you’ve uploaded the tables correctly by previewing the tables you just created.

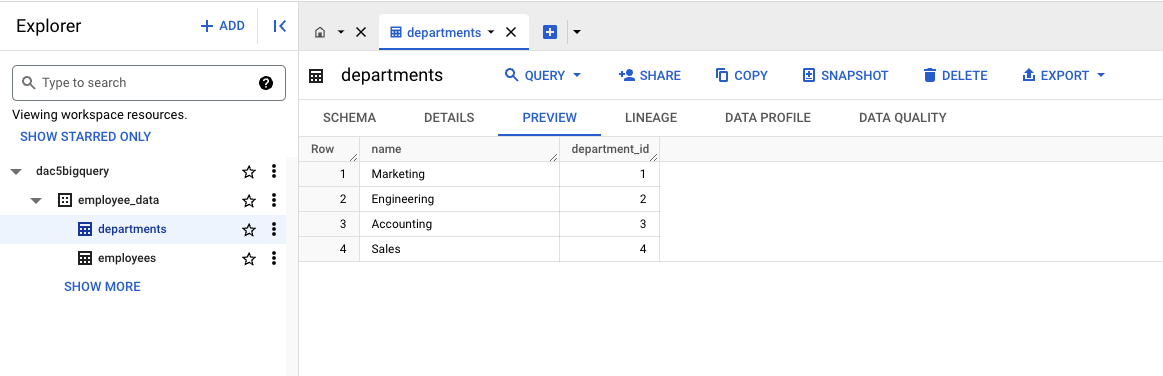
1. From the **Explorer** pane, select the **employees** table.

2. Select the **Preview** tab to verify that you have the following data:



3. Select the **departments** table.

4. Select the **Preview** tab to verify that you have the following data:



There are six rows in the table. In Row 1: name: Dave Smith; department\_id: 1; role: Product Marketing Manager. In Row 2: name: Scott Tanner; department\_id: 1; role: Director of Demand Gen.

When your data previews match these instructions, you are ready to follow along with the step-by-step guide and video on [Explore how JOINs work](https://www.coursera.org/learn/analyze-data/lecture/uLZJH/how-joins-work).

### [STEP-BY-STEP: EXPLORE HOW JOINS WORK](https://www.coursera.org/learn/analyze-data/supplement/PltQp/step-by-step-explore-how-joins-work)

This reading provides you with the steps the instructor performs in the following video, [Explore how JOINs work](https://www.coursera.org/learn/analyze-data/lecture/uLZJH/how-joins-work). The video teaches you how to use **JOIN** in SQL to aggregate data in databases.

Keep this step-by-step guide open as you watch the video. It can serve as a helpful reference tool if you need additional context or clarification while following the video steps. This is not a graded activity, but you can complete these steps to practice the skills demonstrated in the video.

**What you’ll need**

In order to follow along with the instructor, you will need the employee dataset uploaded into your project space. If you haven’t already uploaded this data, follow the instructions in the [Upload the employee dataset to BigQuery](https://www.coursera.org/learn/analyze-data/supplement/13KQO/optional-upload-the-employee-dataset-to-bigquery) reading.

## 

## **Common JOINs**

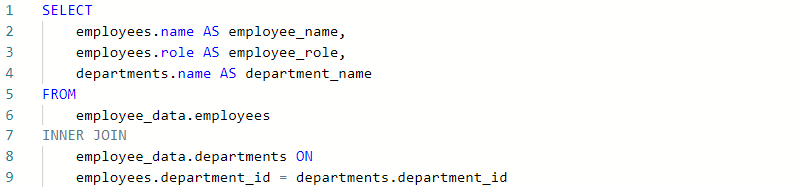
This video explores exactly how **JOIN**s work. A **JOIN** is a SQL clause that is used to combine rows from two or more tables based on a related column. The instructor discusses the different types of **JOIN**s in more detail in the video; here’s a quick reference you can review as you follow along:

* **INNER JOIN:** a function that returns records with matching values in both tables
* **LEFT JOIN:** a function that returns all the records from the left table (first mentioned) and only the matching records from the right table (second mentioned)
* **RIGHT JOIN:** a function that returns all records from the right table (second mentioned) and only the matching records from the left table (first mentioned).
* **OUTER JOIN:** a function that combines the **RIGHT JOIN** and **LEFT JOIN** to return all matching records in both tables.

## **Example 1: INNER JOIN**

In the video, the instructor uses BigQuery to join the employees and departments tables. The following steps take you through typing the query into the query window. If you prefer, you can copy and paste the following query into the query window instead.

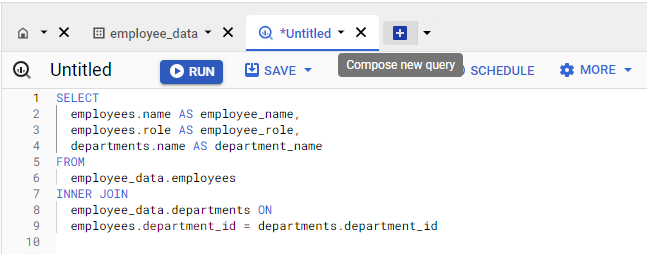
1. In BigQuery, select the **COMPOSE A NEW QUERY** button. BigQuery opens a query window where you can enter your query. The instructor has already executed some queries so their starting line number may be different from yours. If you’ve just opened BigQuery, your line number will be 1. It’s okay if your line numbers aren’t the same as what’s in the video.
2. In line 1 of the query window, enter **SELECT** and then press **Enter**.
3. Press Tab and then in line 2 enter **employees.name AS employee\_name**, then press **Enter**.
4. In line 3, enter **employees.role AS employee\_role**, then press **Enter**.
5. In line 4, enter **departments.name AS department\_name**, then press **Enter**.
6. In line 5, press **Backspace** to stop indenting, enter **FROM**, then press **Enter**.
7. In line 6, press **Tab**, then enter **employee\_data.employees**. Press **Enter**.
8. In line 7, press **Backspace** to stop indenting, enter **INNER JOIN**, then press **Enter**.
9. In line 8, press **Tab**, enter **employee\_data.departments ON**, then press **Enter**.
10. In line 9, enter **employees.deparment\_id = departments.department\_id**.
11. Run the query by selecting the **Run** button.



## 

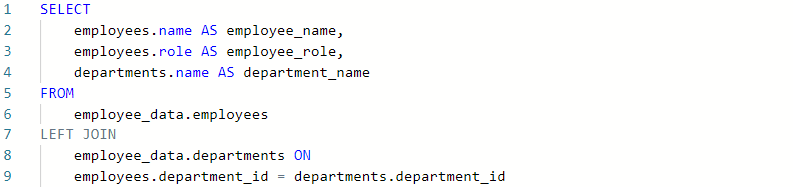
## **Example 2: LEFT JOIN**

You start a new query in BigQuery either by deleting your previous query in the current query window or by opening a new query window. To open a new query window, select the **+** button.



Now, add a new query that uses **LEFT JOIN**. If you prefer, you can copy and paste this query into the query window in BigQuery.

1. In line 1, enter **SELECT** and then press **Enter**.
2. In line 2, press **Tab** to indent the row, enter **employees.name AS employee\_name**, then press **Enter**.
3. In line 3, enter **employees.role AS employee\_role**, then press **Enter**.
4. In line 4, enter **departments.name AS department\_name**, then press **Enter**.
5. In line 5, press **Backspace** to stop indenting, enter **FROM**, then press **Enter**.
6. In line 6, press **Tab**, then enter **employee\_data.employees**, then press **Enter**.
7. In line 7, press **Backspace** to stop indenting, enter **LEFT JOIN**, then press **Enter**.
8. In line 8, press **Tab**, enter **employee\_data.departments ON**, then press **Enter**.
9. In line 9, enter **employees.deparment\_id = departments.department\_id**.
10. Run the query by selecting the **Run** button.



## **Example 3: RIGHT JOIN**

Use the following steps to write a **RIGHT JOIN** query. If you prefer, you can copy and paste the query below into the query window.

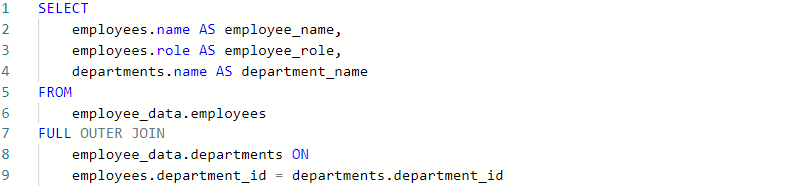
1. Delete the previous query or open a new query window.
2. In line 1, enter **SELECT** and then press **Enter**.
3. In line 2, press **Tab** to indent the row, enter **employees.name AS employee\_name**, then press **Enter**.
4. In line 3, enter **employees.role AS employee\_role**, then press **Enter**.
5. In line 4, enter **departments.name AS department\_name**, then press **Enter**.
6. In line 5, press **Backspace** to stop indenting, enter **FROM**, then press **Enter**.
7. In line 6, press **Tab**, then enter **employee\_data.employees**, then press **Enter**.
8. In line 7, press **Backspace** to stop indenting, enter **RIGHT JOIN**, then press **Enter**.
9. In line 8, press **Tab**, enter **employee\_data.departments ON**, then press **Enter**.
10. In line 9, enter **employees.deparment\_id = departments.department\_id**.
11. Run the query by selecting the **Run** button.



## **Example 4: OUTER JOIN**

Use the following steps to write an **OUTER JOIN** query. If you prefer, you can copy and paste the query below into the query window.

1. Delete the previous query or open a new query window.
2. In line 1, enter **SELECT** and then press **Enter**.
3. In line 2, press **Tab** to indent the row, enter **employees.name AS employee\_name**, then press **Enter**.
4. In line 3, enter **employees.role AS employee\_role**, then press **Enter**.
5. In line 4, enter **departments.name AS department\_name**, then press **Enter**.
6. In line 5, press **Backspace** to stop indenting, enter **FROM**, then press **Enter**.
7. In line 6, press **Tab**, then enter **employee\_data.employees**, then press **Enter**.
8. In line 7, press **Backspace** to stop indenting, enter **FULL OUTER JOIN**, then press **Enter**.
9. In line 8, press **Tab**, enter **employee\_data.departments ON**, then press **Enter**.
10. In line 9, enter **employees.deparment\_id = departments.department\_id**.
11. Run the query by selecting the **Run** button.



### [EXPLORE HOW JOINS WORK](https://www.coursera.org/learn/analyze-data/lecture/uLZJH/explore-how-joins-work)

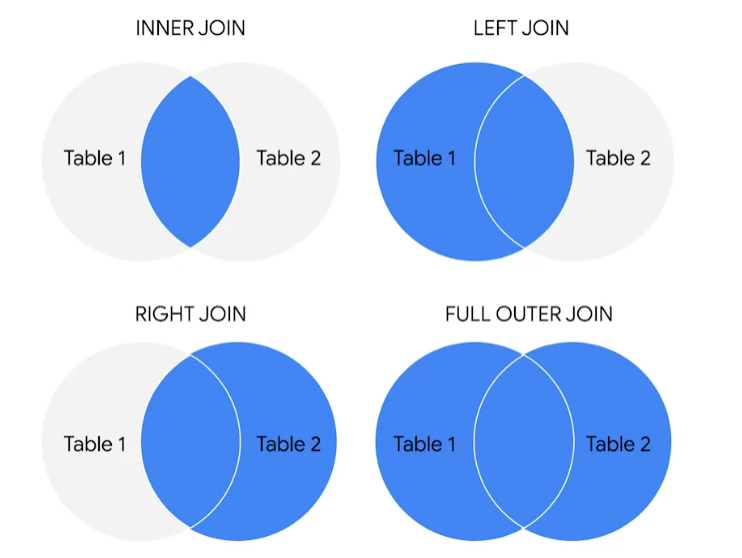
So far we've checked out a few different tools you can use to aggregate data within spreadsheets. In this video, we'll cover how to **use** **JOIN in SQL to aggregate data in databases**.

First, I'll tell you a little bit about what a JOIN actually is, and then we'll explore some of the most common JOINs in action.

Let's get started. **JOIN is a SQL clause that's used to combine rows from two or more tables based on a related column**. Basically, you can **think of a JOIN as a SQL version of VLOOKUP** which we just covered.

*There are* ***four common JOINs*** *data analysts use,* ***inner, left, right, and outer****.*

Here's a handy visualization of what each JOIN actually does. We'll use these to help us understand these functions.



***JOINs help you combine matching or related columns from different tables.***

When we learn about relational databases, we refer to these values as ***primary*** and ***foreign keys***.

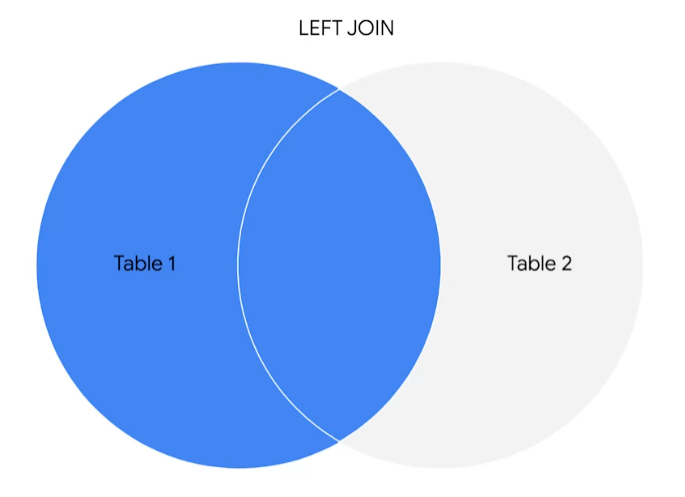
***Primary keys* reference columns in which each value is unique to that table**.

**But that table can have multiple *foreign keys* which are primary keys in other tables.**

For example, in a table about employees, the employee ID is a primary key and the office ID is a foreign key. JOIN uses these keys to identify relationships and corresponding values.

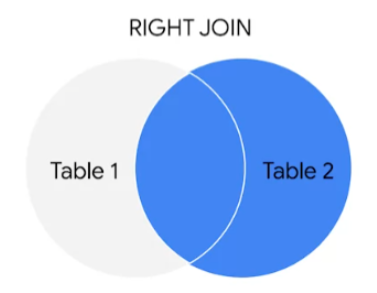
**An inner JOIN is a function that returns records with matching values in both tables**. If we think about our tables as a circle of this **Venn diagram**, then an inner JOIN would return the records that exist where the tables are overlapping. For the records to appear in the results table, they'll have to be key values in both tables. **The records will only merge if there are matches in both tables**. ***When we input JOIN into SQL, it usually defaults to inner JOIN***. A lot of analysts will use JOIN as shorthand instead of typing the whole query.

A **LEFT JOIN is a function that will return all the records from the left table and only the matching records from the right table**. Here's how you can figure out which table is left or right. In English and SQL we read from left to right. **The table mentioned first is left and the table mentioned second is right**. You can also think of left as a table name to the left of the JOIN statement and right as a table name to the right of the JOIN statement. In this diagram, you'll notice that the entire left table is colored in, and that's the overlap with the right table which shows us that the left table and the records it shares with the right table are being selected.

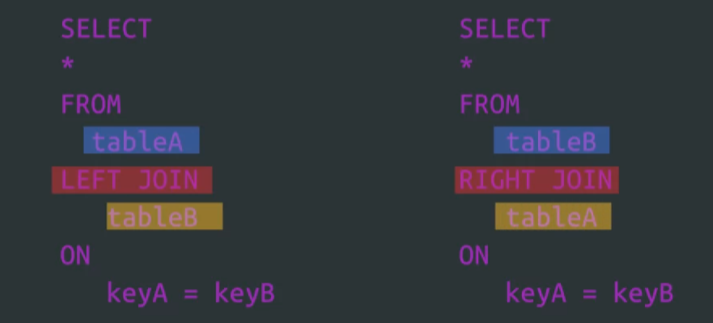


Each row in the left table appears in the results even if there are no matches in the right table.

**RIGHT JOIN does the opposite. It will return all records from the right table and only the matching records from the left**. You can get the same results if you flip the order of the tables and use a left JOIN.



For example, SELECT from table A, LEFT JOIN table B is the same as SELECT from table B, RIGHT JOIN table A.



Finally, there's OUTER JOIN. **OUTER join combines RIGHT and LEFT JOIN to return all matching records in both tables. This means it will return all records in both tables**. If there are records in one table without a match, it'll create a record with no values for the other table.

**Using JOINs can make working with multiple data sources a lot easier and it can make relationships between tables more clear.**

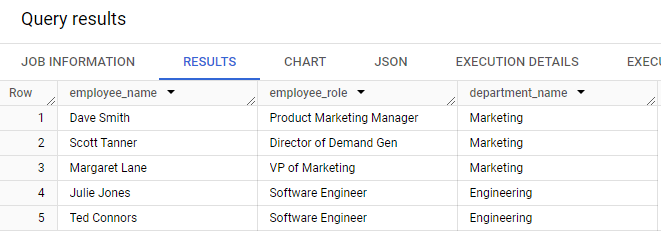
Here's an example. Let's say we're working with employee data across multiple departments. We have an employees table and a departments table which both have some columns like department ID. We can use different JOIN clauses to help us put different data from our tables and aggregate it. Maybe we want to get a list of employees with their department name, excluding any employee without a department ID. Because the department ID record is used in both tables, we can use an INNER JOIN to return a list with only those employees. As a quick reminder, analysts will sometimes just input JOIN for an INNER JOIN but for this example, we'll write it out.

To build this query, we'll start with SELECT and AS to tell SQL how we want the columns titled.

Then we'll use FROM to tell it where we're getting this data, in this case the employees table.   
Then we'll input INNER JOIN and the other table we're using, which is departments.



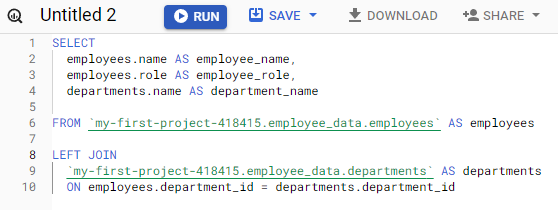
**We can specify which column and each table will contain the matching JOIN key by writing ON employees.department\_id equals departments.departments\_id**. Now, let's run it, and there.



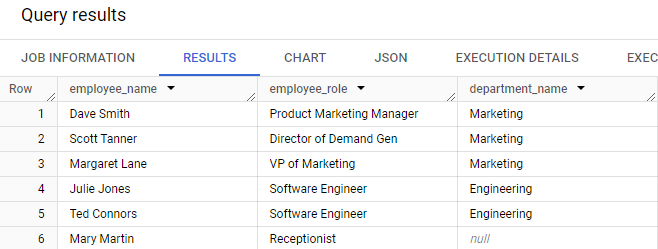
Now we've got a list of employee names and department IDs for the employees that have those IDs.

But we could use LEFT or RIGHT join to return a list of all employee names and their departments when available. Let's try both really quickly.

This will start similar to the last query, we'll put in SELECT AS and FROM again. But this time we'll say LEFT JOIN and use ON like we did with the last query.

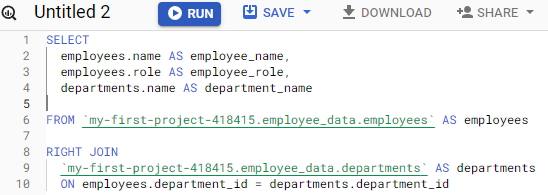


When we execute the query, we get back this new list with the employee names and departments.

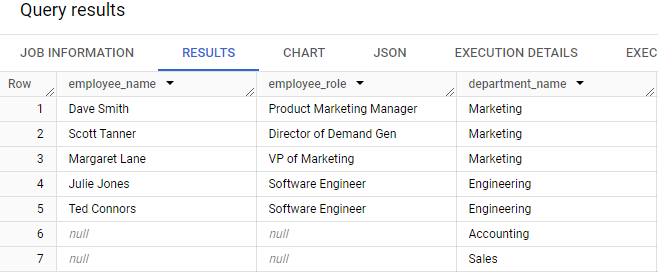


But **you'll notice there's null values**. These are places where the right table which is departments in this case didn't have corresponding values.

Let's try RIGHT JOIN just to test it out. This query will be almost the same. Only difference is that we'll use the RIGHT JOIN clause to return all the rows from the right table, whether they have matching values in the table to the left of the JOIN statement or not.

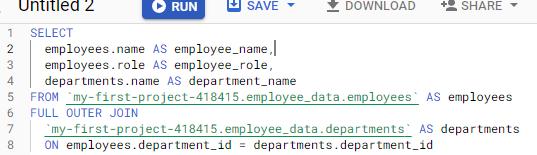


In this case, the right table is departments.

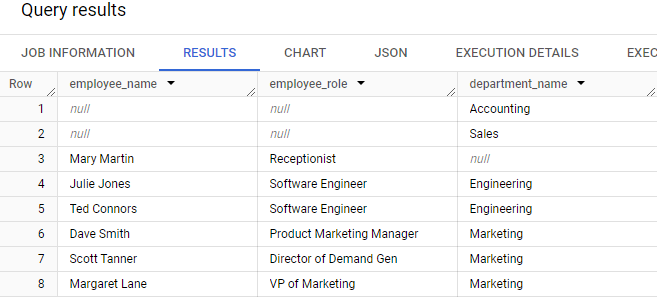


Now, let's try out one last JOIN: OUTER.

**OUTER JOIN will fetch all of the employee names and departments**. Again, this query will start a lot like the other ones we've done, we'll use SELECT AS and FROM to choose what data we want and how. We'll grab this from the employees table, and put FULL OUTER JOIN with the departments table to get all of the records from both. We'll also use ON again here.



Now we can run this,



and we'll get all of the employee names and departments from these tables. There will be nulls in the department.name column, the employee.name column and role column because we've joined columns that don't have matching values, and there. Now you know how JOINs work. JOINs are super useful when you need to work with data from multiple related tables. They give you a lot of flexibility with how you combine and view that data. If you ever have trouble remembering what INNER, RIGHT, LEFT, or OUTER JOIN do, just think back to our Venn diagram. We'll keep learning about aggregating data in SQL next time. See you soon.



### [SECRET IDENTITIES: THE IMPORTANCE OF ALIASES](https://www.coursera.org/learn/analyze-data/supplement/qURXP/secret-identities-the-importance-of-aliases)

**Now you will learn about using aliasing to simplify your SQL queries**.

**Aliases** are used in SQL queries to create temporary names for a column or table.

Aliases make referencing tables and columns in your SQL queries much simpler when you have table or column names that are too long or complex to make use of in queries.

Imagine a table name like **special\_projects\_customer\_negotiation\_mileages**. That would be difficult to re-enter every time you use that table. With an alias, you can create a meaningful nickname that you can use for your analysis. In this case **special\_projects\_customer\_negotiation\_mileages** can be aliased to simply **mileage**. Instead of having to write out the long table name, you can use a meaningful nickname that you decide.

## **Basic syntax for aliasing**

**Aliasing** is the process of using aliases. In SQL queries, aliases are implemented by making use of the **AS** command. The basic syntax for the **AS** command can be seen in the following query for aliasing a table:

***NOTICE*** that **AS** is preceded by the table name and followed by the new nickname. It is a similar approach to aliasing a column:In both cases, you now have a new name that you can use to refer to the column or table that was aliased.

### **Alternate syntax for aliases**

If using **AS** results in an error when running a query because the SQL database you are working with doesn't support it, you can leave it out. In the previous examples, the alternate syntax for aliasing a table or column would be:

* **FROM table\_name alias\_name**
* **SELECT column\_name alias\_name**

The key takeaway is that queries can run with or without using **AS** for aliasing, but using **AS** has the benefit of making queries more readable. It helps to make aliases stand out more clearly.

## **Aliasing in action**

Let’s check out an example of a SQL query that uses aliasing. Let’s say that you are working with two tables: one of them has employee data and the other one has department data. The **FROM** statement to alias those tables could be:



These aliases still let you know exactly what is in these tables, but now you don’t have to manually input those long table names. **Aliases can be really helpful for long, complicated queries.** It is easier to read and write your queries when you have aliases that tell you what is included within your tables.

## **For more information**

If you are interested in learning more about aliasing, here are some resources to help you get started:

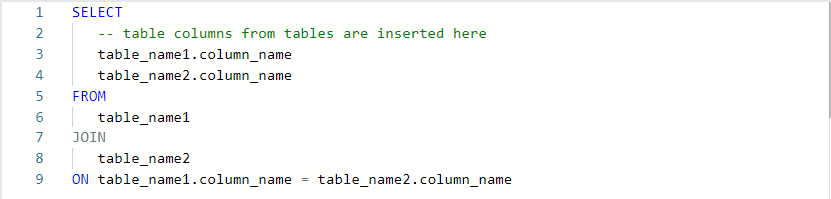
* [**SQL Aliases**](https://www.w3schools.com/sql/sql_alias.asp)**:** This tutorial on aliasing is a really useful resource to have when you start practicing writing queries and aliasing tables on your own. It also demonstrates how aliasing works with real tables.
* [**SQL Alias**](https://www.sqltutorial.org/sql-alias/)**:** This detailed introduction to aliasing includes multiple examples. This is another great resource to reference if you need more examples.
* [**Using Column Aliasing**](https://documentation.sas.com/?cdcId=pgmsascdc&cdcVersion=9.4_3.5&docsetId=sqlproc&docsetTarget=p0aymxwsvbt5wcn1lncugwjtf758.htm&locale=en)**:** This is a guide that focuses on column aliasing specifically. Generally, you will be aliasing entire tables, but if you find yourself needing to alias just a column, this is a great resource to have bookmarked.

### [USE JOINS EFFECTIVELY](https://www.coursera.org/learn/analyze-data/supplement/DBOi7/use-joins-effectively)

In this reading, you will review how **JOIN**s are used and will be introduced to some resources that you can use to learn more about them. A **JOIN** combines tables by using a primary or foreign key to align the information coming from both tables in the combination process. **JOIN**s use these keys to identify relationships and corresponding values across tables.

If you need a refresher on primary and foreign keys, refer to the [glossary](https://www.coursera.org/learn/analyze-data/supplement/0p8b6/glossary-terms-and-definitions) for this course, or go back to [Databases in data analytics](https://www.coursera.org/learn/data-preparation/supplement/uXqEX/databases-in-data-analytics).

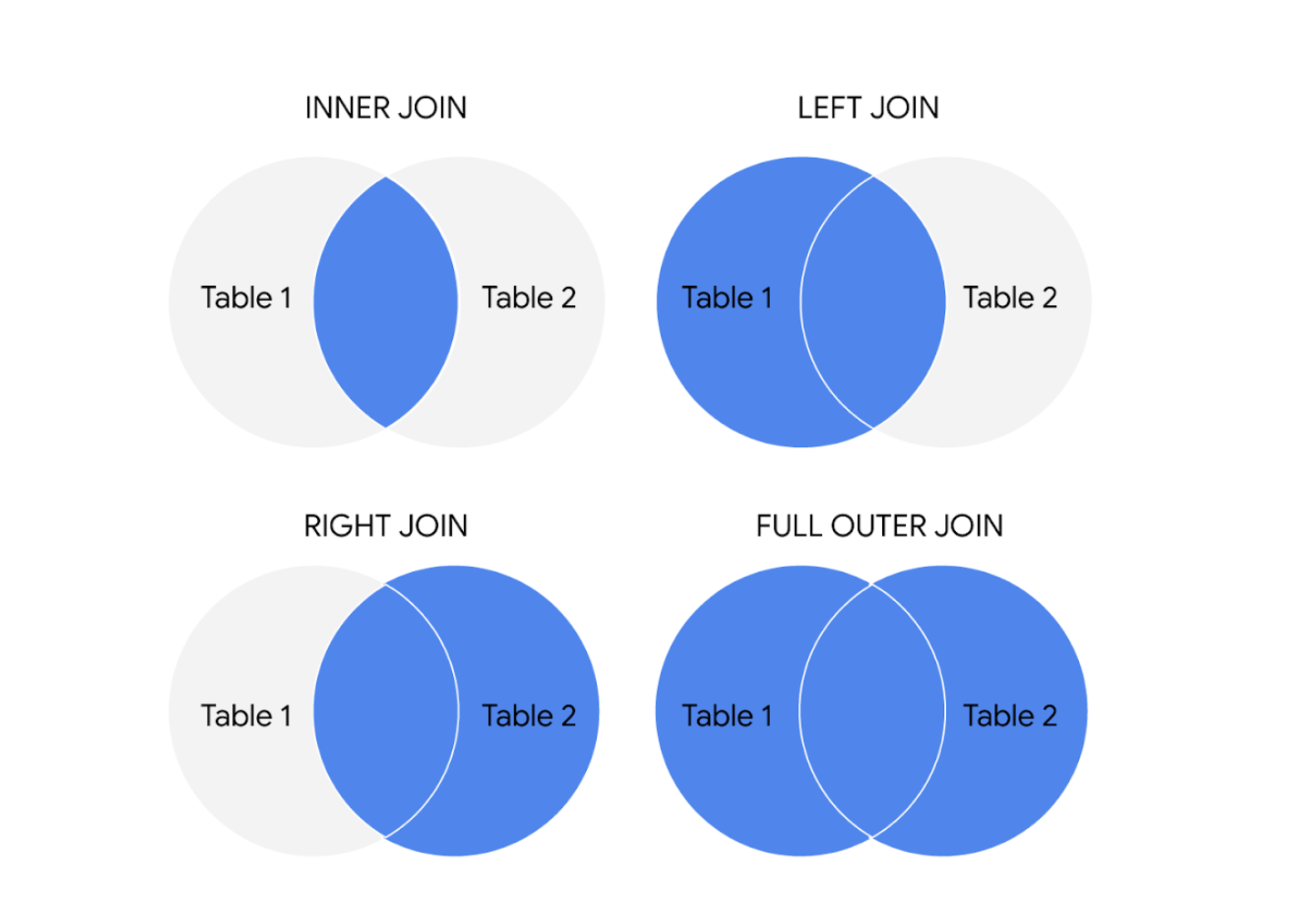
## **The general JOIN syntax**



As you can see from the syntax, the **JOIN** statement is part of the FROM clause of the query. **JOIN** in SQL indicates that you are going to combine data from two tables. **ON** in SQL identifies how the tables are to be matched for the correct information to be combined from both.

## **Type of JOINs**

There are four general ways in which to conduct **JOIN**s in SQL queries: **INNER**, **LEFT**, **RIGHT**, and **FULL OUTER**.



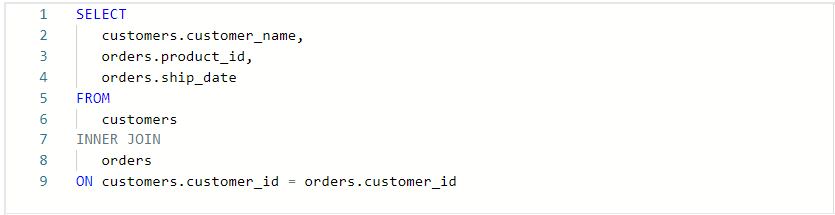
The circles represent left and right tables, and where they are joined is highlighted in blue

Here is what these different **JOIN** queries do.

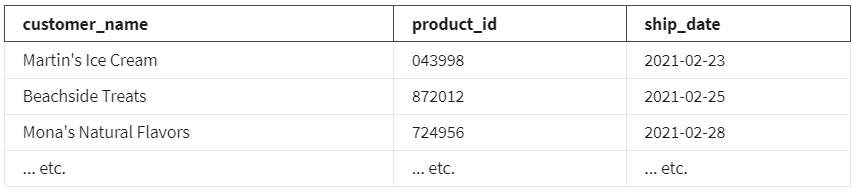
### 

### **INNER JOIN**

INNER is *optional* in this SQL query because it is the default as well as the most commonly used **JOIN** operation. You may see this as **JOIN** only. **INNER JOIN** returns records if the data lives in both tables. For example, if you use **INNER JOIN** for the **customers** and **orders** tables and match the data using the **customer\_id** key, you would combine the data for each **customer\_id** that exists in both tables. If a **customer\_id** exists in the **customers** table but not the **orders** table, data for that **customer\_id** isn’t joined or returned by the query.

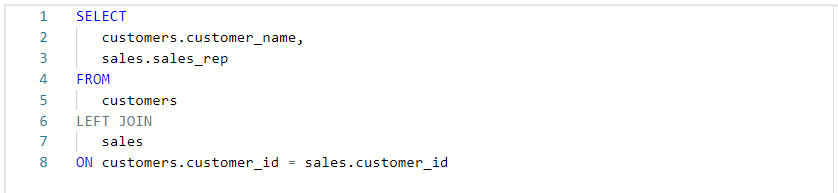


The results from the query might look like the following, where customer\_name is from the customers table and product\_id and ship\_date are from the orders table:

The data from both tables was joined together by matching the **customer\_id** common to both tables. Notice that **customer\_id** doesn’t show up in the query results. It is simply used to establish the relationship between the data in the two tables so the data can be joined and returned.

### **LEFT JOIN**

You may see this as **LEFT OUTER JOIN**, but most users prefer **LEFT JOIN**. Both are correct syntax. **LEFT JOIN** returns all the records from the left table and only the matching records from the right table. Use **LEFT JOIN** whenever you need the data from the entire first table and values from the second table, if they exist. For example, in the query below, **LEFT JOIN** will return **customer\_name** with the corresponding **sales\_rep**, if it is available. If there is a customer who did not interact with a sales representative, that customer would still show up in the query results but with a **NULL** value for **sales\_rep**.

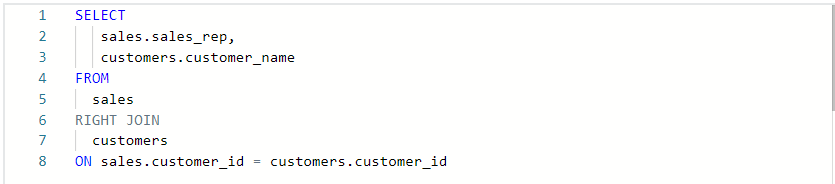


The results from the query might look like the following where **customer\_name** is from the **customers** table and **sales\_rep** is from the **sales** table. Again, the data from both tables was joined together by matching the **customer\_id** common to both tables even though **customer\_id** wasn't returned in the query results.

### 

### **RIGHT JOIN**

You may see this as **RIGHT OUTER JOIN** or **RIGHT JOIN**. **RIGHT JOIN** returns all records from the right table and the corresponding records from the left table. Practically speaking, **RIGHT JOIN** is rarely used. Most people simply switch the tables and stick with **LEFT JOIN**. But using the previous example for **LEFT JOIN**, the query using **RIGHT JOIN** would look like the following:

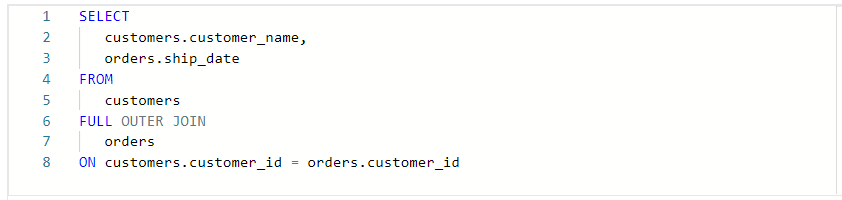


The query results are the same as the previous **LEFT JOIN** example.

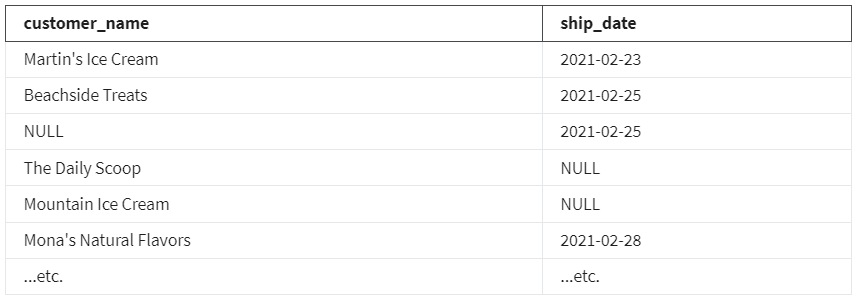
### 

**FULL OUTER JOIN**

You may sometimes see this as **FULL JOIN**. **FULL OUTER JOIN** returns all records from the specified tables. You can combine tables this way, but remember that this can potentially be a large data pull as a result. **FULL OUTER JOIN** returns all records from *both* tables even if data isn’t populated in one of the tables. For example, in the query below, you will get all customers and their products’ shipping dates. Because you are using a **FULL OUTER JOIN**, you may get customers returned without corresponding shipping dates or shipping dates without corresponding customers. A **NULL** value is returned if corresponding data doesn’t exist in either table.



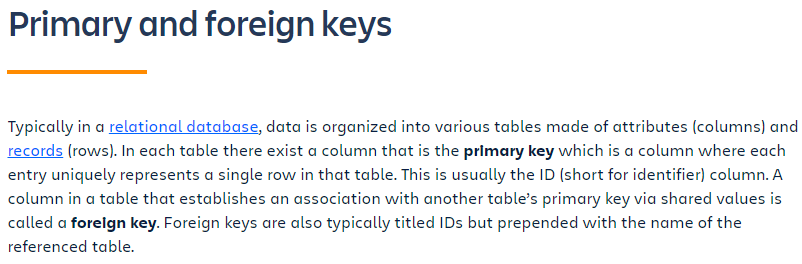
The results from the query might look like the following

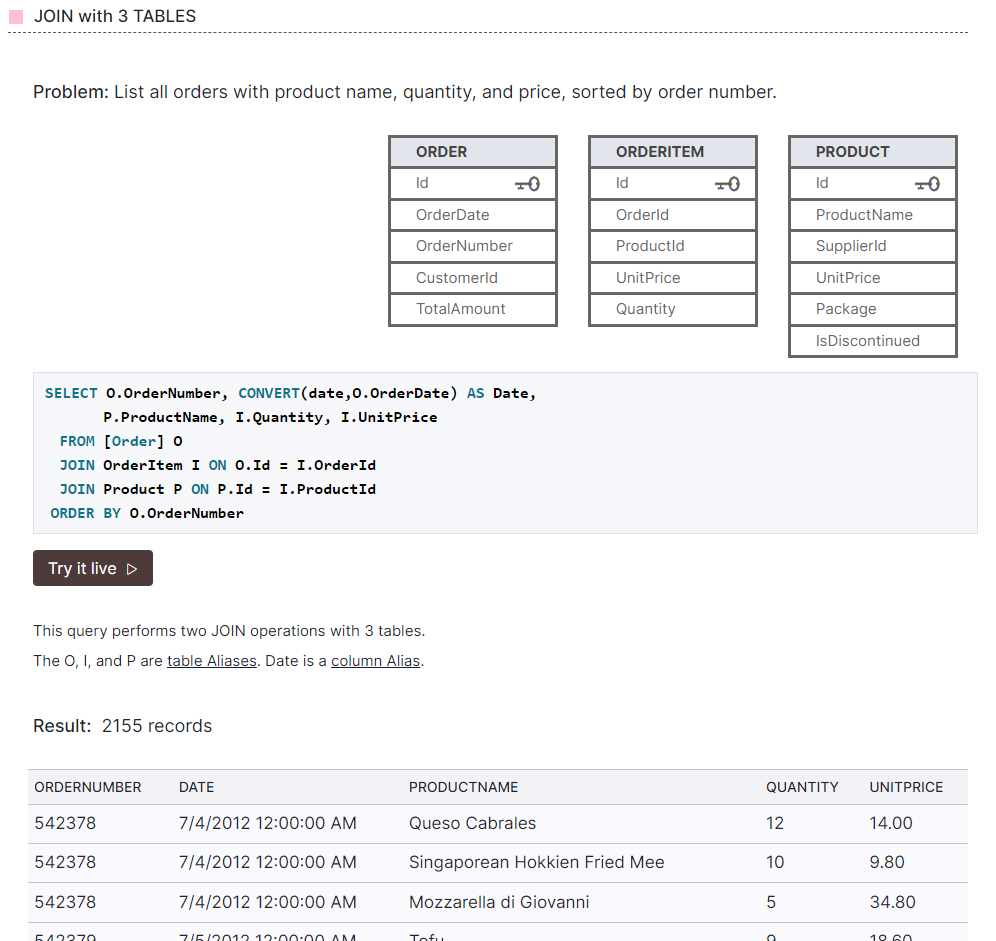
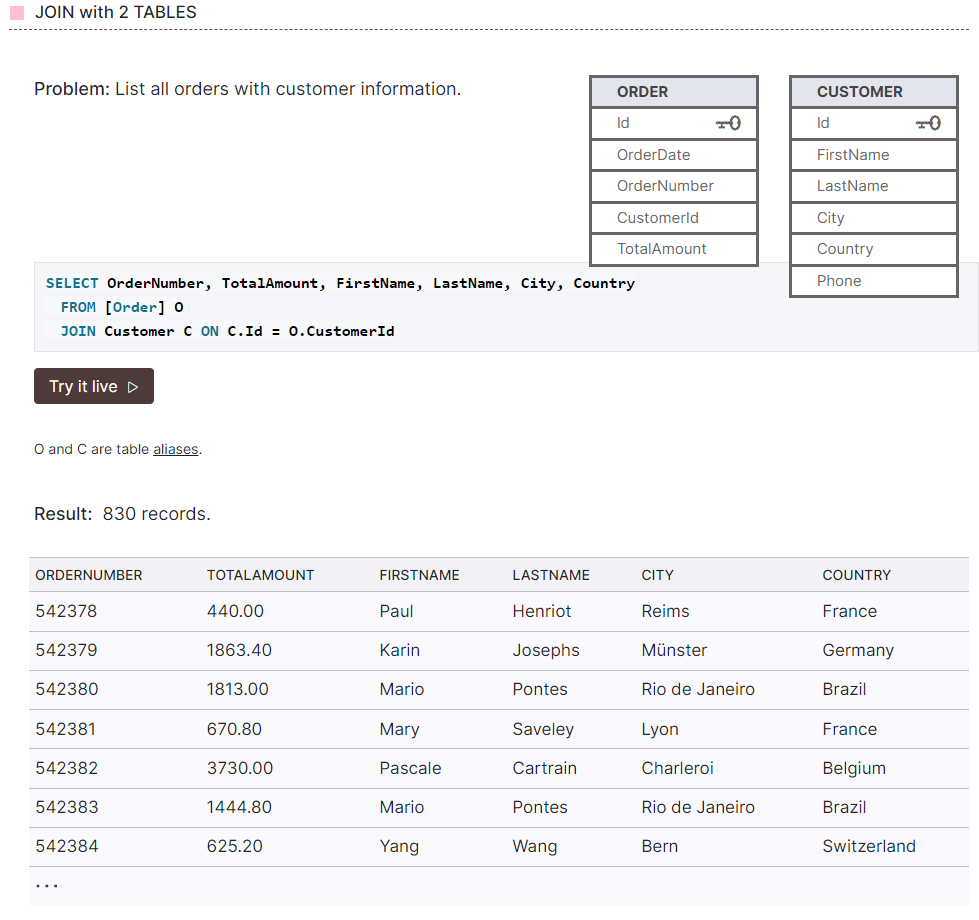
.

**FOR MORE INFORMATION**

**JOIN**s are going to be useful for working with relational databases and SQL—and you will have plenty of opportunities to practice them on your own. Here are a few other resources that can give you more information about **JOIN**s and how to use them:

* [**SQL JOINs**](https://www.w3schools.com/sql/sql_join.asp)**:** This is a good basic explanation of **JOIN**s with examples. If you need a quick reminder of what the different **JOIN**s do, this is a great resource to bookmark and come back to later.
* [**Database JOINs - Introduction to JOIN Types and Concepts**](https://www.essentialsql.com/introduction-database-joins/)**:** This is a really thorough introduction to **JOIN**s. Not only does this article explain what **JOIN**s are and how to use them, but it also explains the various scenarios in more detail of when and why you would use the different **JOIN**s. This is a great resource if you are interested in learning more about the logic behind **JOIN**ing.
* [**SQL JOIN Types Explained in Visuals**](https://dataschool.com/how-to-teach-people-sql/sql-join-types-explained-visually/)**:** This resource has a visual representation of the different **JOIN**s. This is a really useful way to think about **JOIN**s if you are a visual learner, and it can be a really useful way to remember the different **JOIN**s.
* [**SQL JOINs: Bringing Data Together One Join at a Time**](https://towardsdatascience.com/sql-join-8212e3eb9fde)**:** Not only does this resource have a detailed explanation of **JOIN**s with examples, but it also provides example data that you can use to follow along with their step-by-step guide. This is a useful way to practice **JOIN**s with some real data.
* [**SQL JOIN:**](https://www.dofactory.com/sql/join) This is another resource that provides a clear explanation of **JOINs** and uses examples to demonstrate how they work. The examples also combine **JOIN**s with aliasing. This is a great opportunity to see how **JOIN**s can be combined with other SQL concepts that you have been learning about in this course.





### [HANDS-ON ACTIVITY: QUERIES FOR JOINS](https://www.coursera.org/learn/analyze-data/quiz/vnDR2/hands-on-activity-queries-for-joins)

## **Activity Overview**

You’ve just learned about **JOIN**s and aliasing. Now, you’ll practice writing queries that join multiple tables to build more robust datasets and use aliasing to make your SQL queries clearer. In this activity, you will load and examine two tables from the World Bank’s International Education dataset in order to identify and understand their keys. You will also review **JOIN**s and write your own query that includes **JOIN**s and aliases.   
Finally, you’ll use a **JOIN** to answer a specific question about the data.

By learning how to apply **JOIN** statements and aliasing, you’ll be able to fully harness the power of relational databases by combining data from tables linked by keys.

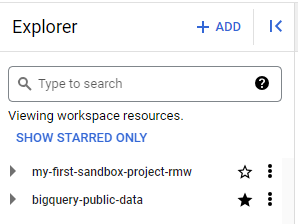
### **Step-By-Step Instructions**

### **Step 1: Load the dataset**

1. Log in to [BigQuery Sandbox](https://cloud.google.com/bigquery/docs/sandbox). If you have a free trial version of BigQuery, you can use that instead. On the BigQuery page, select the Go to BigQuery button.

Note: BigQuery Sandbox frequently updates its user interface. The latest changes may not be reflected in the screenshots presented in this activity, but the principles remain the same. Adapting to changes in software updates is an essential skill for data analysts, and it’s helpful for you to practice troubleshooting. You can also reach out to your community of learners on the discussion forum for help.

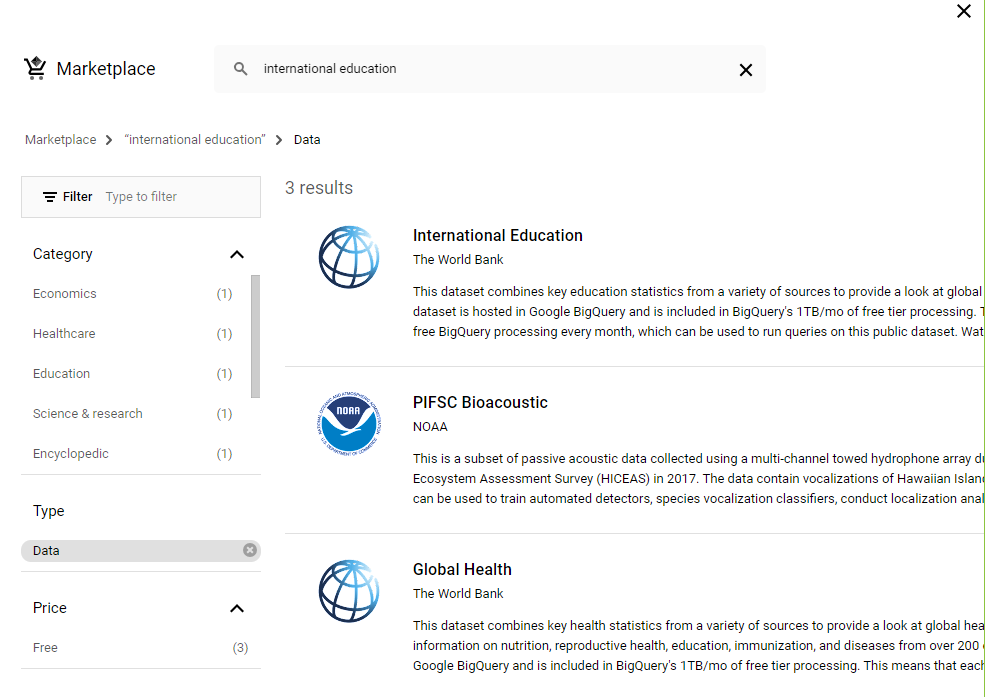
2. Now, you’ll find the Editor interface. There are three major menus: the BigQuery navigation menu, the Explorer pane where you can search for datasets, and the details pane.



3. Select the + ADD button in the Explorer pane, then scroll down and select the Public Datasets option.

Screenshot of the Public Datasets option in the drop-down list

4. In the Search Marketplace text box, enter international education and press Return to find the search results.

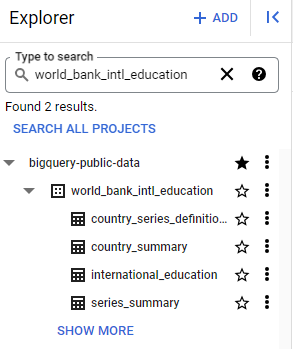


5. Select the World Bank's International Education dataset, which is the first result.

6. Select the View Dataset button to open the dataset in BigQuery in a new tab.

Note: You may want to star the bigquery-public-data resource in the Explorer pane. That way, you can access and browse the public datasets and tables more easily without having to navigate to the marketplace in the future.

7. In the Explorer pane, search for world\_bank\_intl\_education. Expand the dataset to explore the tables it contains. You may also need to click on the SHOW MORE button for the additional tables to display.



### 

### **Step 2: Identify and understand keys**

Before you begin joining tables together, **take a moment to consider how JOINs work:** Two tables must be connected by their primary and foreign keys in order to join them. **Keys are the most important elements of JOIN**s—**JOIN**s function by combining tables based on those shared fields.

When designing a **JOIN** statement, these keys are listed in the **ON** statements as references to specific columns or fields within each table from the join: primary and foreign keys.

Consider two tables in the world\_bank\_intl\_education dataset: international\_education and country\_summary. In order to understand how you might join these tables, take a moment to identify which columns you could use to combine them from each table. You can do that by examining the table schemas.

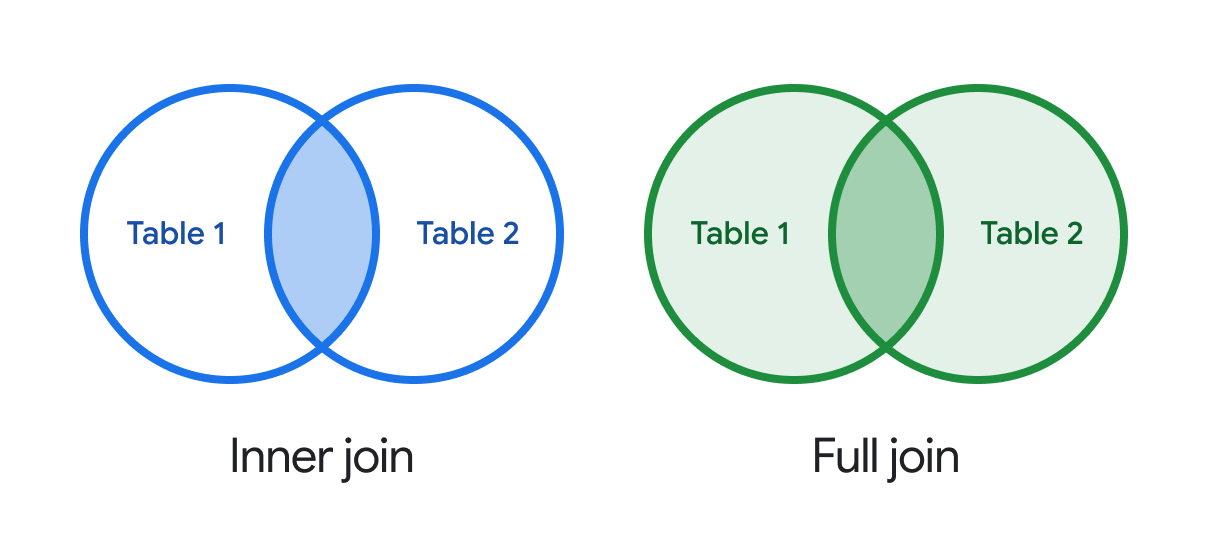
1. In the Explorer pane, select the international\_education table. This will bring up the table’s schema in the details pane. If the schema doesn’t appear, select the Schema tab in the details pane. You might also check out the preview to view the data in the table.
2. Next, select the country\_summary table and examine its schema. You’ll find that the country\_code column appears in both table schemas. You might also check out the preview.

What do you notice about the columns from each table?

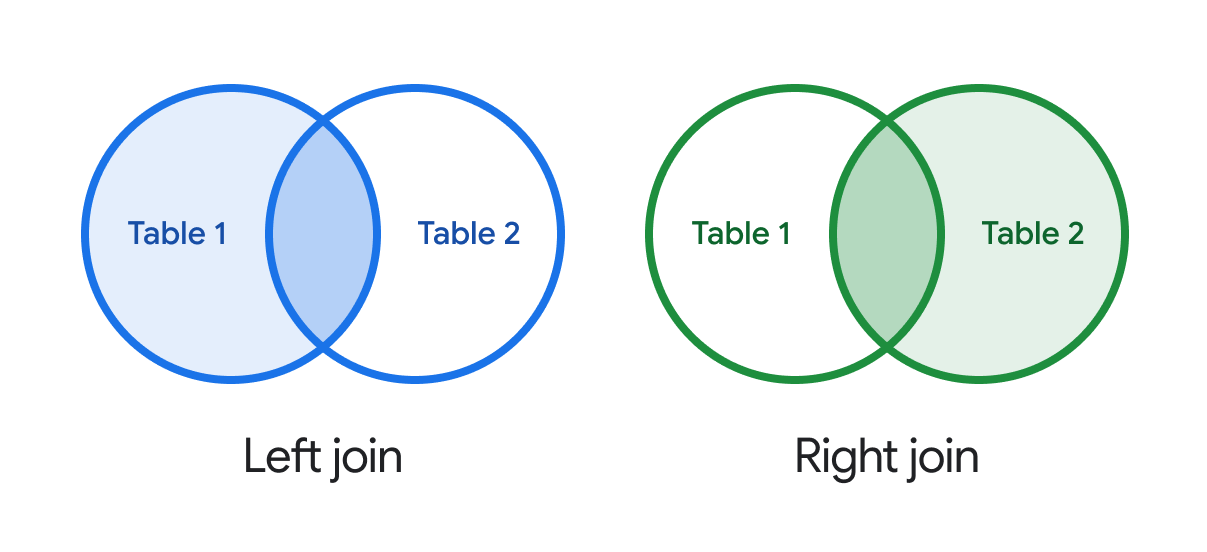
Both of these tables share a field name: **'country\_code'**. As you continue with this activity, you will use this common field for your JOIN as both your primary and foreign key. It’s important to understand that foreign keys don't always have the same names across tables. If you’re ever unsure if the columns are the same, you can always double-check. To do so, select the Details tab for each table and confirm that they contain the same kinds of information.

### Step 3: Review JOINs

Now that you have identified the key field for joining these tables, take a moment to review the various kinds of **JOIN** statements.



Circles represent table 1 and table 2. In the inner join, where the tables are joined is highlighted in blue, which means the results will display only the rows from the two tables where the primary key in table 1 matches the foreign key in table 2. In the full join, tables 1 and 2 are completely green. This means the query results will display all rows from both tables regardless of whether their primary key and foreign key match.



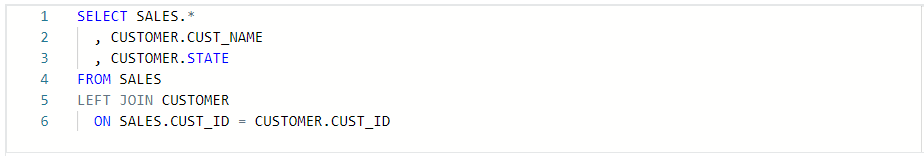
Circles represent table 1 and table 2. In the left join, table 1 is highlighted in blue and the overlap indicates that the join will show all rows in table 1 and only the related rows from table 2. In the right join, table 2 is highlighted green. The overlap shows that the results will contain all rows from table 2 and only the related rows from table 1.

The two most common kinds of JOIN statements are **INNER JOIN** and **OUTER LEFT JOIN** (also known simply as **LEFT JOIN**). As a review:

* **INNER JOIN**: Returns only the rows where the target appears in both tables.
* **LEFT JOIN**: Returns every row from the left table, as well as any rows from the right table with matching keys found in the left table.

Before you work with the BigQuery sample dataset, here’s an example to help solidify your understanding of how to select the appropriate type of **JOIN**.

Imagine that you have one table with a list of customers (**Table 1 = CUSTOMER**) and another with a list of sales (**Table 2 = SALES**). You want to return all of the **SALES** fields, plus the customer name (**field\_name = CUST\_NAME**) and the state (**field\_name = STATE**) that the customer lives in. The query final query should appear like this:



In this case, **SALES.CUST\_ID** is the primary key and **CUSTOMER.CUST\_ID** is the foreign key. This will show you all sales, regardless of whether there is an associated customer ID (**CUST\_ID**).

Now, consider the following: If you only want to query SALES where there is a corresponding customer ID in the CUSTOMER table, how might you change the previous query?

One way to accomplish this is to simply change the type of **JOIN**. **LEFT JOIN CUSTOMER to INNER JOIN CUSTOMER**.

With that simple change, all of the rows from the SALES table that were missing a corresponding customer ID from the table would NOT show.

The type of **JOIN** you use is important. You can inadvertently add or lose records with a simple incorrect selection of your **JOIN** type. When writing a query, it can help to draw out a Venn diagram like the example graphic above to help you decide which sort of **JOIN** you need.

### 

### **Step 4: Query the dataset and incorporate aliases**

Now, it’s time to actually query the dataset. In this section, you’re going to practice using **JOIN**s and explore how aliasing can help develop complex queries.

As a starting point, this is a **JOIN** that does not use any aliasing. In the details pane, compose a new query to open a query window. In the query window, copy, paste, and run the following query:

1 SELECT `bigquery-public-data.world\_bank\_intl\_education.international\_education`.country\_name,

2 `bigquery-public-data.world\_bank\_intl\_education.country\_summary`.country\_code,

3 `bigquery-public-data.world\_bank\_intl\_education.international\_education`.value

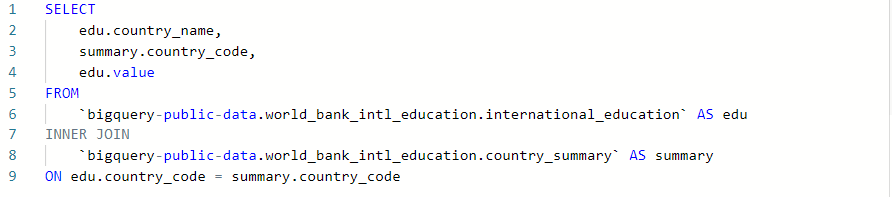
4 FROM `bigquery-public-data.world\_bank\_intl\_education.international\_education`

5 INNER JOIN `bigquery-public-data.world\_bank\_intl\_education.country\_summary`

6 ON `bigquery-public-data.world\_bank\_intl\_education.country\_summary`.country\_code = `bigquery-public-data.world\_bank\_intl\_education.international\_education`.country\_code

This basic query joins the **'country\_summary'** and **'international\_education'** tables on the **'country\_code'** key and returns the country name, country code, and value column. It’s a lot of effort to write out the full schema, table, field names every time you want to reference them. You can make these much easier to work with by using aliasing.

Here is the exact same query, but this time it uses an alias for each table:



This query is much easier to read and understand. You can set aliases for tables and fields by specifying the alias for the table after the table’s name in **FROM** or **JOIN** statements. In this case, the international\_education table was renamed as edu, and the country\_summary table as summary. Using descriptive aliases is a best practice and will help you keep your queries clean, readable, and easy to work with.

### **Step 5: Use a JOIN to answer a question**

Now that you’ve confirmed that the **JOIN** statement works, answer the following question: In 2015, how many people were of the official age for secondary age broken down by region of the world?

For this query, you will need to perform some additional tasks before returning the result:

* Exclude rows with a missing region.
* Use the **SUM(value)** to calculate the total population for a given grain size.
* Sort by highest population region first.

**In a query window, copy, paste, and run the following query:**

SELECT

summary.region,

SUM(edu.value) secondary\_edu\_population

FROM

`bigquery-public-data.world\_bank\_intl\_education.international\_education` AS edu

INNER JOIN

`bigquery-public-data.world\_bank\_intl\_education.country\_summary` AS summary

ON edu.country\_code = summary.country\_code --country\_code is our key

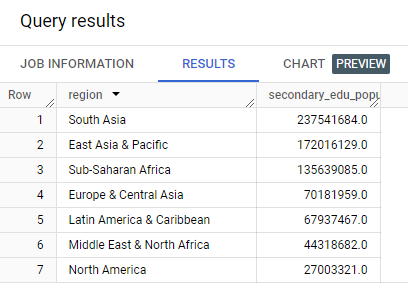
WHERE summary.region IS NOT NULL

AND edu.indicator\_name = 'Population of the official age for secondary education, both sexes (number)'

AND edu.year = 2015

GROUP BY summary.region

ORDER BY secondary\_edu\_population DESC



Notice how, in this query, an alias is also set to give the **AVG(edu.value)** a more descriptive name for the temporary table the query returns.

Also note that the **WHERE** statement excludes rows with any null information. This is necessary to present the data succinctly and display only seven rows for the seven regions present in the data. However, this **WHERE** statement means that the results will return the same regardless of which **JOIN** you use. In the next section, you’ll explore a situation where you need to use a specific kind of join in your query.

### Step 6: Decide when to use INNER JOINs versus OUTER JOINs

In the last query, you used an **INNER JOIN** to find the total population of the official age for secondary education in 2015 broken down by region. Because of the **WHERE** statement that removed null values in our previous query, using any other **JOIN** methods (left outer join, right outer join, full outer) would have produced the same result.

Now, you will write a **LEFT JOIN**, a type of **OUTER JOIN**, for a situation where the type of query you use will change the result you return.

Consider this scenario: You have been tasked to provide data for a feature sports article on NCAA basketball in the 1990s. The writer wants to include a funny twist about which Division 1 team mascots were the winningest.

You’ll need a list of all NCAA Division I colleges and universities; their mascots, if applicable; and their number of wins and losses. You can find this information by typing ncaa\_basketball in the Explorer tab dataset search bar on BigQuery.

Next, start a new query tab by clicking on the blue + button. Your query should join the season statistics from one table with the mascot information from another. You need to use a **LEFT JOIN** instead of an **INNER JOIN** because not all teams have mascots. If you use an **INNER JOIN**, you would exclude teams with no mascot.

To demonstrate this, copy, paste, and run the following query:

SELECT

seasons.market AS university,

seasons.name AS team\_name,

mascots.mascot AS team\_mascot,

AVG(seasons.wins) AS avg\_wins,

AVG(seasons.losses) AS avg\_losses,

AVG(seasons.ties) AS avg\_ties

FROM `bigquery-public-data.ncaa\_basketball.mbb\_historical\_teams\_seasons` AS seasons

LEFT JOIN `bigquery-public-data.ncaa\_basketball.mascots` AS mascots

ON seasons.team\_id = mascots.id

WHERE seasons.season BETWEEN 1990 AND 1999

AND seasons.division = 1

GROUP BY 1,2,3

ORDER BY avg\_wins DESC, university

This is an example of when a **LEFT JOIN** is more helpful than an **INNER JOIN**. With this query, you can review college basketball statistics to get a better sense of which teams (and mascots) were winning most in the 1990s.

### [UPLOAD THE WAREHOUSE DATASET TO BIGQUERY](https://www.coursera.org/learn/analyze-data/supplement/HuXCc/upload-the-warehouse-dataset-to-bigquery)

Coming up, you’re going to learn more about how to use **COUNT** and **COUNT DISTINCT** in SQL to count and return the number of certain values in a dataset.

To prepare for these activities, you will need to log in to your BigQuery account and upload the warehouse data provided below as two .csv files.

## **Upload the data**

To begin, download the two .csv files from the attachments below:

[Warehouse Orders - Warehouse](https://d3c33hcgiwev3.cloudfront.net/PcJNDf5ySCqk7V2xQoiu2Q_90874562d5cd4e7c9ba848fd44d512e1_Warehouse-Orders---Warehouse.csv?Expires=1712880000&Signature=UzNz~cyyNB-GKL38Ng6-GhVCjR3lSNz84uSSzf4zKKKjEQfbNMgLF6gtj5H0YewETL4lu-HcCx~m53ROmNqZymWCqkUi9brg6GHyFHRjci~1KWYwXhlDuhS27wP0WfLce2pcdWZaBDCwMIJ8AJ9LoGrsjvzAatQC~e7BCvWs6Bw_&Key-Pair-Id=APKAJLTNE6QMUY6HBC5A)

[Warehouse Orders - Orders](https://d3c33hcgiwev3.cloudfront.net/helGYTW6Tv-vyOH0l-yIPA_1a019ecb2cce49f48fd69d85f7dadce1_Warehouse-Orders---Orders.csv?Expires=1712880000&Signature=Dj8XVfFJ3DnwNfefaXO6wmyUDta1UU3eTFAAnmaW3cUgP11nrapVaNAzqkMWKj~k9RzSylRatyROwfw1i2RvPOi-V3inCoCPiQh7BfNY9SrtoXSZk8UWtdeM00Xh4ZHgrcY7JyfGQ5S4zVU7MpITI-b3sWxuAfjeb-2m2zewBfE_&Key-Pair-Id=APKAJLTNE6QMUY6HBC5A)

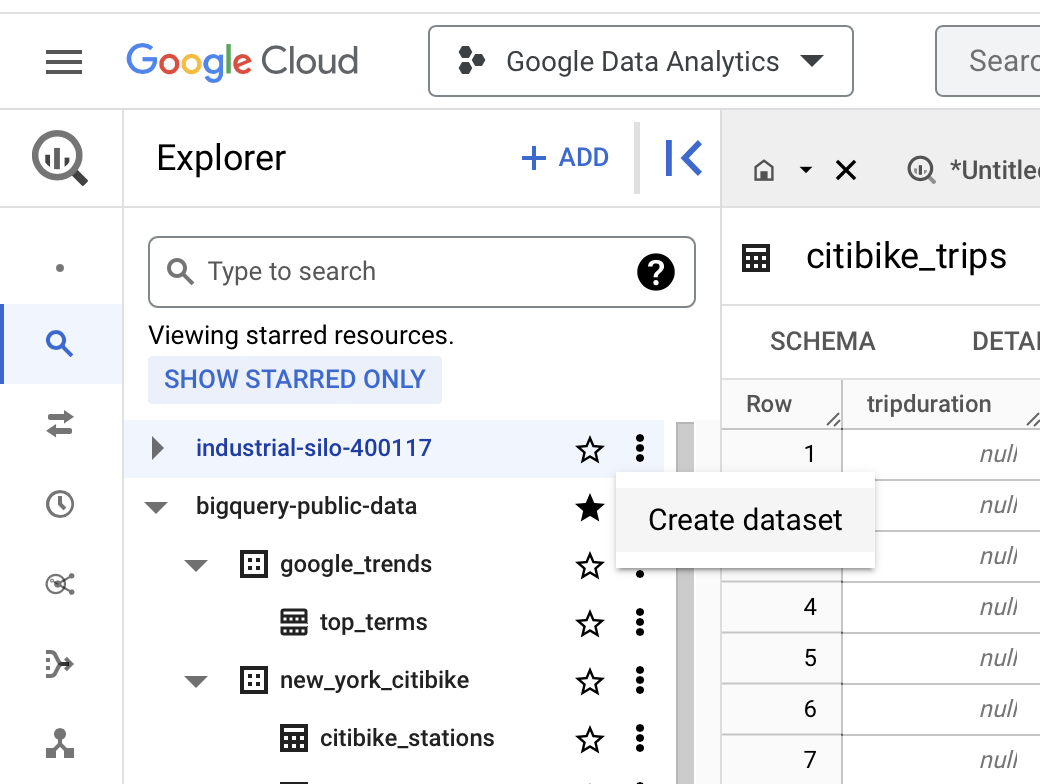
**Note:** If the .csv files open in a new tab instead of downloading to your machine, follow these instructions:

* Select **File** from the menu bar, then select **Save as Google Sheets**. This will open the .csv file as a Google Sheet.
* Select **File** from the menu bar, then select **Download** from the dropdown menu, then select **Comma Separated Values (.csv)**.

Next, complete the following steps in your BigQuery console to upload the Warehouse Orders dataset with the two Warehouse and Orders tables.

1. Open your [BigQuery console](https://console.cloud.google.com/bigquery) and select the project you want to upload the data to.

2. In the Explorer pane, select the Actions icon (three vertical dots) and select **Create dataset**.

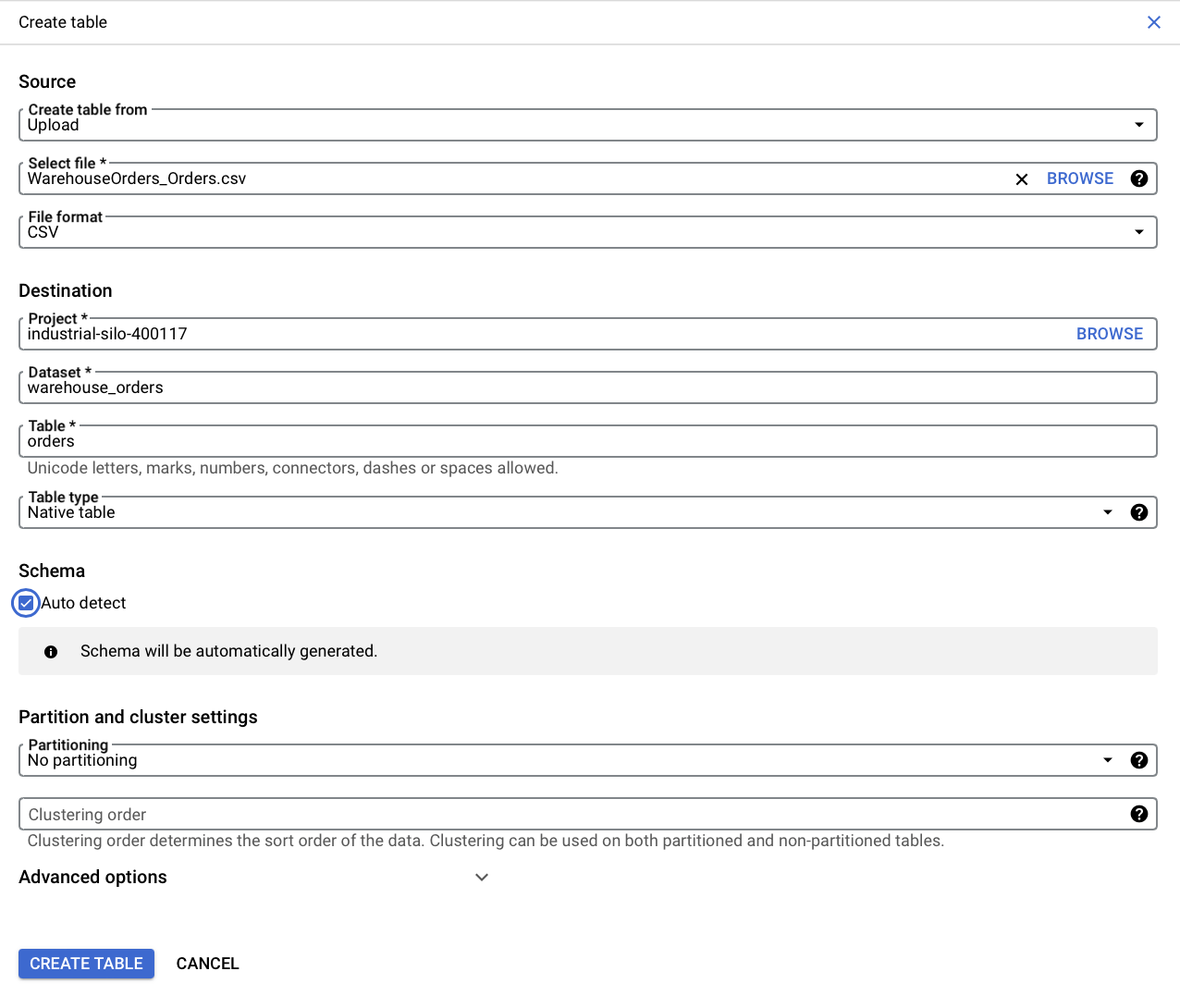


3. The name "warehouse\_orders" will be used for the dataset. Enter **warehouse\_orders** for the Dataset ID in the Create dataset pane. Make sure the **Location type** is set to **Multi-region** and that the region selected is **US (multiple regions in the United States)**. Leave the **Advanced options** at their default settings.

4. Select the **CREATE DATASET** button to add the dataset to your project.

5. In the Explorer, select the **warehouse\_orders** dataset you just created. You will then view the dataset info window in the main editor window.

6. Navigate to the **+ CREATE TABLE** button to open the **Create table** pane.



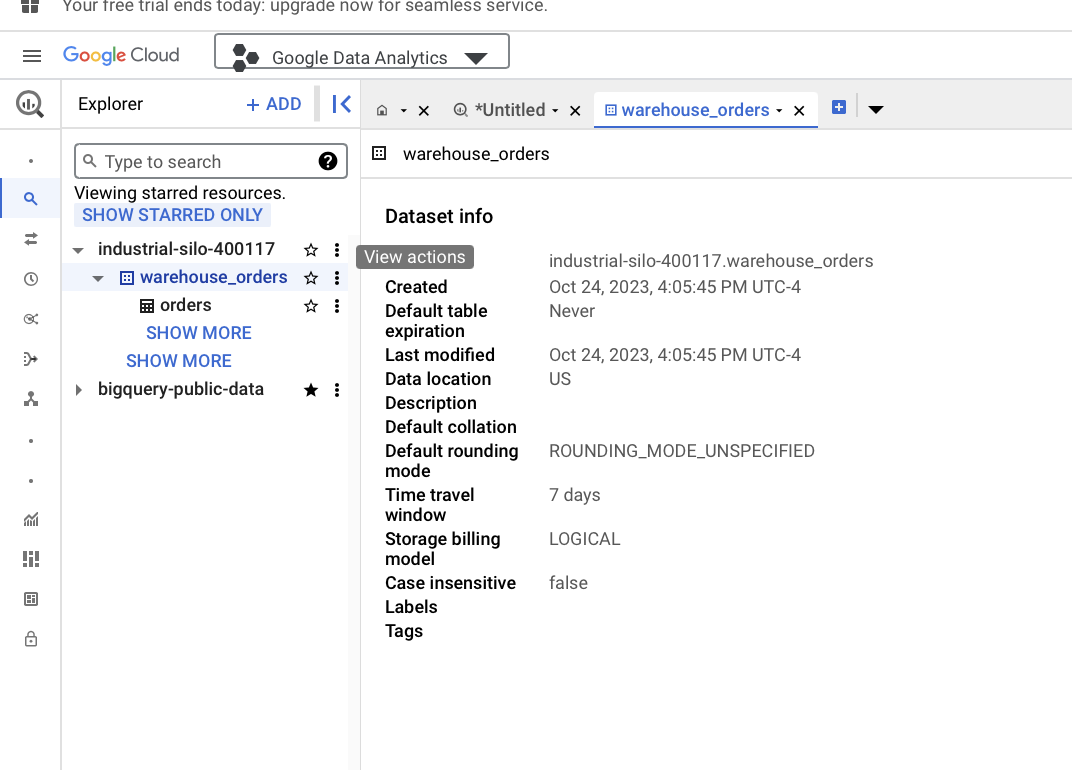
7. From the **Create table from the dropdown list**, choose where the data will be coming from.

* Select **Upload**.
* Select the **Browse** button to select the **Warehouse Orders - Orders.csv** file you downloaded.
* Choose **CSV** from the file format dropdown list (file type may automatically be detected).

8. In the **Table** text box, enter **orders** if you plan to follow along with the video.

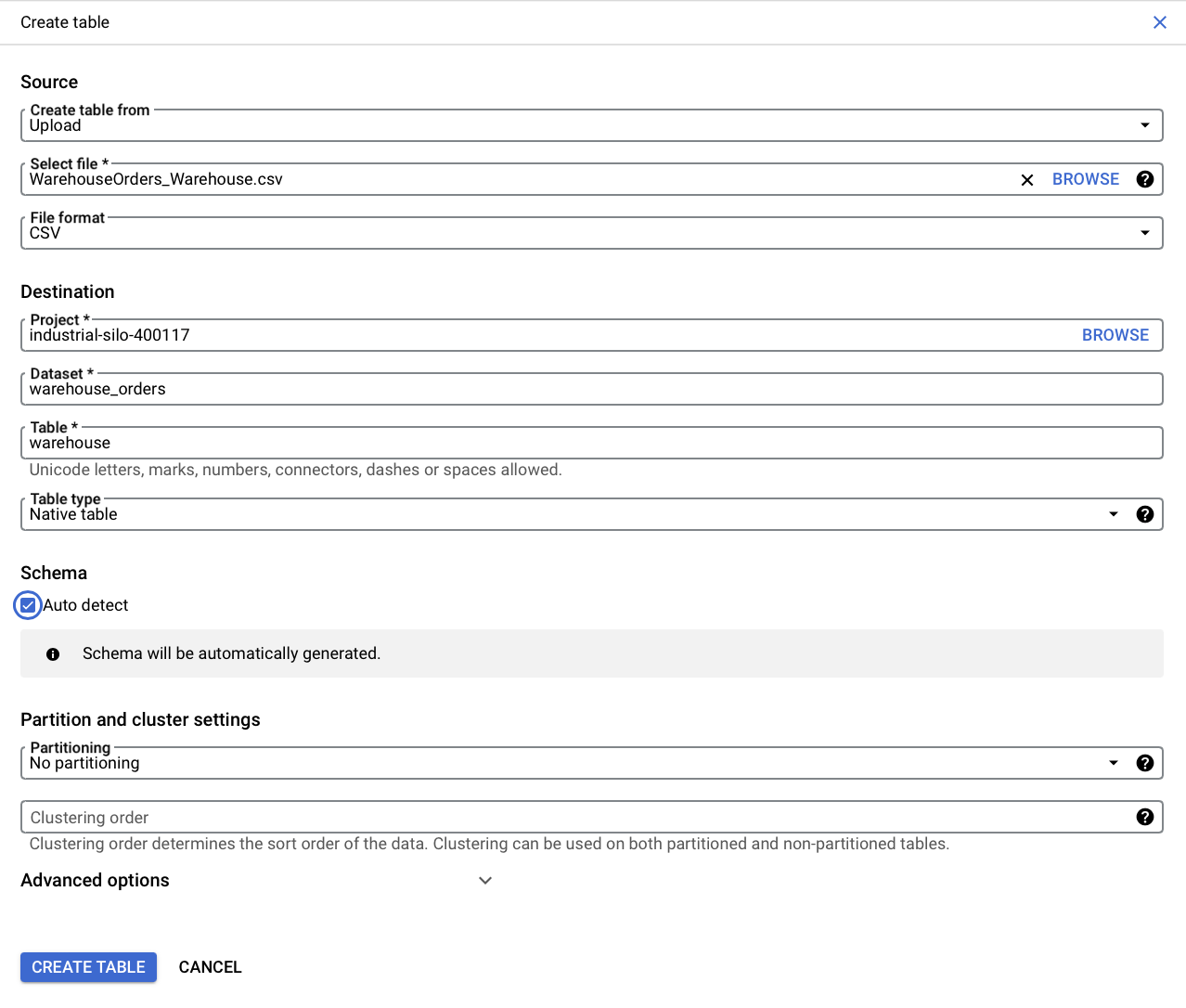
9. Below **Schema**, select the **Auto detect** checkbox.

10. Select **CREATE TABLE**. You will now find the **orders** table below your **warehouse\_orders** dataset in your project.



11. Select the **warehouse\_orders** dataset again.

12. Navigate to the **+ CREATE TABLE** button to open the **Create table pane** again.



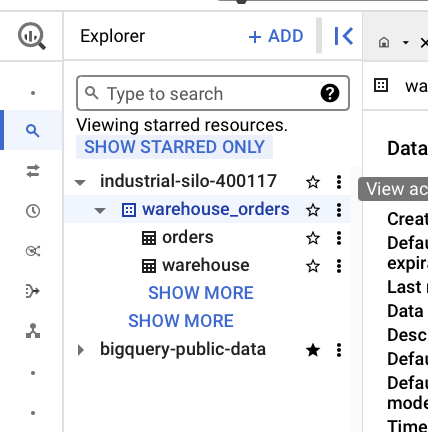
13. For the Create table from selection, choose where the data will be coming from.

* Select **Upload**.
* Select **Browse** to select the **Warehouse Orders - Warehouse.csv** file you downloaded.
* Choose **CSV** from the **File format** dropdown list.

14. For Table name, enter **warehouse** if you plan to follow along with the video.

15. For Schema, select the **Auto detect** checkbox.

16. Select the **CREATE TABLE** button. You will now find the **warehouse** table under your **warehouse\_orders** dataset in your project.



17. In the Explorer pane, select the **orders** table and then select the **Preview** tab to verify that you retrieve the first 50 rows of data. You may have to scroll down the page to view all 50 rows (there are 9999 pages total).



18. Select the **warehouse** table and select the **Preview** tab to verify that you can view 10 rows of data. If both your data previews match, you are ready to move along to the [COUNT and COUNT DISTINCT](https://www.coursera.org/teach/analyze-data/authoringBranch~4f7eJX9NEe6L0QpSVaIh4Q/content/item/project/3K5St) activity.

### [HANDS-ON ACTIVITY: COUNT AND COUNT DISTINCT](https://www.coursera.org/learn/analyze-data/quiz/IDPTu/hands-on-activity-count-and-count-distinct)

## **Activity Overview**

****

You have learned that spreadsheets and SQL have a lot in common. In a spreadsheet, the **COUNT** function is used to count the number of cells that contain numerical values in a specified range or in an array of cells.   
In SQL, **COUNT** and **COUNT DISTINCT** are similar tools.

The **COUNT** function returns the number of records that are returned by a query. **COUNT DISTINCT** performs the same function as **COUNT**, but it also removes both duplicate rows of the same data and null values from the result set. In this activity, you will practice using both **COUNT** and **COUNT DISTINCT** in your queries.

**By the time you complete this activity, you will be able to use COUNT and COUNT DISTINCT in your queries to determine the amounts of things**.

Remember, **COUNT** and **COUNT DISTINCT** will return numerical values found within a dataset, **helping you answer questions like, “How many customers did this?” Or, “How many transactions were there this month?” Or, “How many dates are in this dataset?”**

### 

### **Step-By-Step Instructions**

### 

### **Step 1: Access the dataset**

For this activity, you will need the warehouse dataset uploaded into your BigQuery project space. If you haven’t already uploaded this data, follow the instructions in the [Upload the warehouse dataset to BigQuery](https://www.coursera.org/learn/analyze-data/supplement/HuXCc/optional-upload-the-warehouse-dataset-to-bigquery) reading.

### 

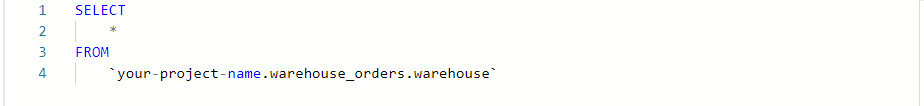
### **Step 2: Examine the warehouse\_orders.warehouse table**

In this scenario, you are a junior data analyst working for a company that manufactures socks. You have access to data on the company’s customers, orders, warehouses, and products. Within the dataset, there are two tables: **warehouse** and **orders**.

Begin by examining the **warehouse** table:

1. In line 1, enter **SELECT**, then press Enter/Return on your keyboard.
2. In line 2, press Tab on your keyboard. Then, add an asterisk (**\***), then press Enter/Return.
3. In line 3, press Backspace to remove the indentation, then enter **FROM**. Then, press Enter/Return.
4. In line 4, press Tab, then enter **`your-project-name.warehouse\_orders.warehouse`**. (Replace **your-project-name** with the unique name of your project).
5. Select RUN.

You can also copy and paste the following query into the query window instead, making sure to replace **your-project-name** with your unique project name.



After running the query, the five columns from the warehouse table will load in the Query results window:

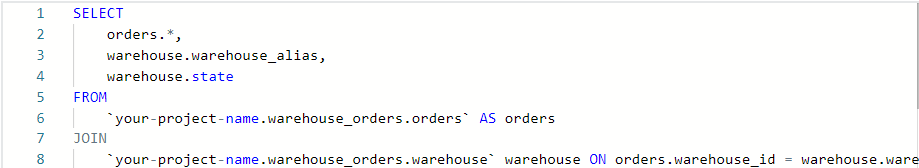
* warehouse\_id: indicates the ID number of each warehouse
* warehouse\_alias: indicates the alias, or name, of each warehouse
* maximum\_capacity: indicates the maximum capacity at each warehouse
* employee\_total: indicates the total number of employees at each warehouse
* state: indicates the postal abbreviation for the U.S. state each warehouse is located in

### **Step 3: Examine the warehouse\_orders.order table**

Next, create a new query to retrieve the first 100 rows of the orders table. Use LIMIT to limit the number of rows returned. This is useful if you're working with large datasets, especially if you just want to explore a small sample of that dataset.

1. Start a new query.
2. In line 1, enter **SELECT**, then press Enter/Return.
3. In line 2, press Tab on your keyboard. Then, add an asterisk (**\***), then press Enter/Return.
4. In line 3, press Backspace to remove the indentation, then enter **FROM**. Then, press Enter/Return.
5. In line 4, press Tab, then enter **`your-project-name.warehouse\_orders.orders`**. (Replace **your-project-name** with the unique name of your project). Then press Enter/Return.
6. In line 5, press Backspace to remove the indentation, then enter **LIMIT 100**.
7. Select RUN.

You can also copy and paste the following query into the query window instead, making sure to replace **your-project-name** with your unique project name.





After running the query, the five columns from the orders table will load in the Query results window:

* order\_id: indicates the ID number of each order
* customer\_id: indicates the ID number of each customer
* warehouse\_id: indicates the ID number of each warehouse
* order\_date: indicates the date on which the order was placed
* shipper\_date: indicates the date on which the order was shipped

Also notice the number of results returned in the query: 100 results.

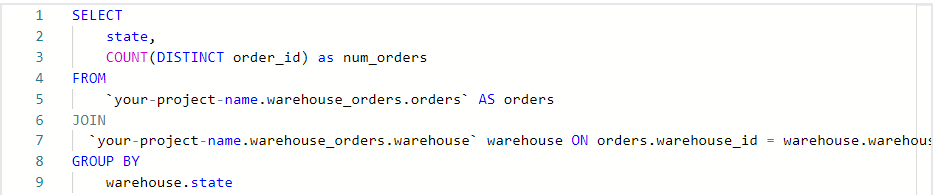
### 

### **Step 4: Create aliases and JOIN the tables**

Aliasing involves temporarily naming a table or column in your query to make it easier to read and write. Because these names are temporary, they only last for the given query. Use a **FROM** statement to write in what the tables' aliases are going to be. This will save time in other parts of the query. To alias the **warehouse\_orders.orders** table as **orders**:

1. In line 1, enter **SELECT**, then press Enter/Return.
2. In line 2, press Tab on your keyboard. Then add an asterisk (**\***), then press Enter/Return.
3. In line 3, press Backspace to remove the indentation, then enter FROM. Then press Enter/Return.
4. In line 4, press Tab, then enter **`your-project-name.warehouse\_orders.orders` orders**. (Replace **your-project-name** with the unique name of your project). Then press Enter/Return.

Note: An alternate syntax uses the **AS** keyword to assign an alias name:



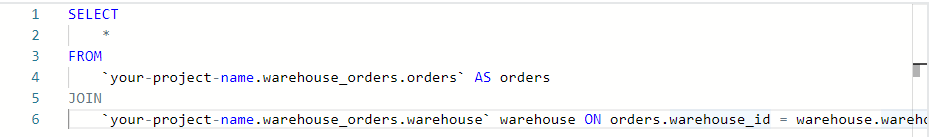
Queries can run with or without the **AS** keyword, but using **AS** enables an alias to stand out so the query is easier to read.

Perhaps you need both the warehouse details and the order details so you can report on the distribution of orders by state. Use **JOIN** to join the two tables together to get data from both of them and alias the warehouse table in the process. In this case, use **JOIN** as shorthand for **INNER JOIN** to get corresponding data from both tables.

6. In line 5, press Backspace, then enter **JOIN**, then press Enter/Return.

7. In line 6, press Tab, then enter **warehouse\_orders.warehouse warehouse ON orders.warehouse\_id = warehouse.warehouse\_id**.

Your query text should read as follows:



With the aliases in place and the two tables joined, circle back and update the **SELECT** statement at the beginning of the query:

8. In line 1, press Enter/Return to create a new line after **SELECT**.

9. In line 2, press Tab, then enter **orders.\*,** (that is, “orders” followed by a period, asterisk, and comma). Press Enter/Return.

10. In line 3, enter **warehouse.warehouse\_alias** followed by a comma, then press Enter/Return.

11. In line 4, enter **warehouse.state**.

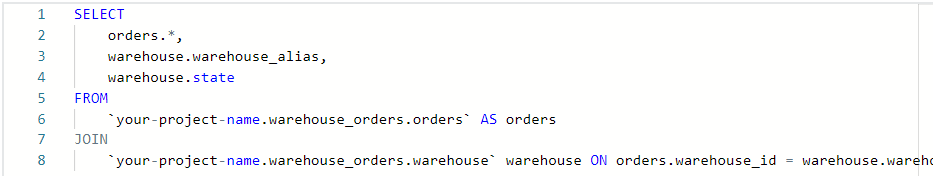
12. Line 5 should now contain a single asterisk. Delete this line before running the query.

13. Select RUN.

After running the query, the data from both tables are now joined together in the Query results window with these seven columns:

* order\_id: indicates the ID number of each order.
* customer\_id: indicates the ID number of each customer.
* warehouse\_id: indicates the ID number of each warehouse.
* order\_date: indicates the date on which the order was placed.
* shipper\_date: indicates the date on which the order was shipped.
* warehouse\_alias: indicates the names given to each warehouse as an alias.
* state: indicates which state the warehouse is located in.

You can also copy and paste the completed query text below into the query window. Just remember to replace **your-project-name** with your unique project name.

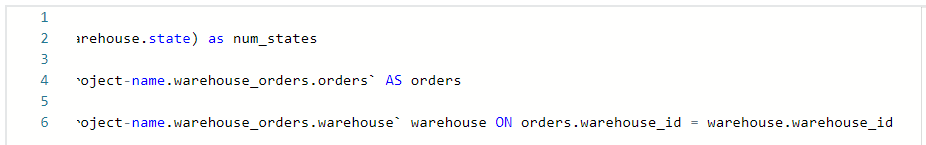


### **Step 5: Find the number of states with warehouses that shipped orders with COUNT DISTINCT**

As a data analyst, you might be interested in finding the number of states with warehouses that have shipped orders.

First, you'll try to do this with the COUNT function. Begin by modifying the query you wrote in the previous step to create aliases and **JOIN** the tables.

1. Delete lines 2-4. Line 2 should be blank, in between **SELECT** in line 1 and **FROM** in line 3.
2. In line 2, press Tab to create an indentation. Then enter **COUNT(warehouse.state) as num\_states**.
3. Select RUN.

Your query text should read as follows:

Notice how the query returned more than 9,000 results. There are only 50 states, so this is clearly not the answer you're looking for! This is because the query counted every single record (row) that included a state, regardless of duplicates or null values.

**Don't worry! You can modify the existing query to remove duplicates and null values, and only count the distinct states with COUNT DISTINCT:**

1. In line 2, enter **DISTINCT** after the open parenthesis and add a space before **warehouse.state** so that line 2 reads: **COUNT (DISTINCT warehouse.state) as num\_states** . This will modify **COUNT** to operate as **COUNT DISTINCT**, and it will remove all the repeated instances from the results.
2. Select RUN.

According to the results, there are three distinct states in the **orders** data. Nice work! You've successfully used **COUNT DISTINCT**.

### 

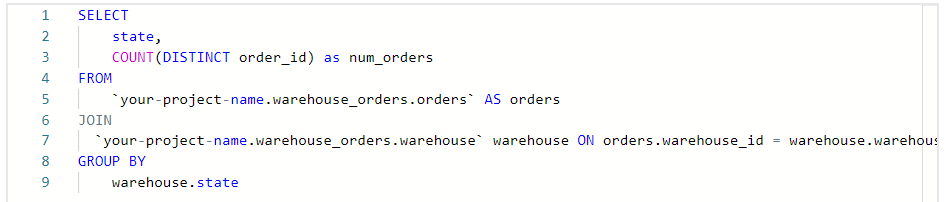
### 

### **Step 6: Use GROUP BY to group the number of orders by state**

Next, you might want to find the number of orders shipped from warehouses in each state, instead of the number that shipped orders. You can find this information by using **GROUP BY** to group the **state** column in the warehouse table. Use **JOIN** and **GROUP BY** in the **FROM** statement.

1. In line 7, enter **GROUP BY**, then press Enter/Return.
2. In line 8, press Tab to create an indentation, then enter **warehouse.state**.
3. Select **RUN**.

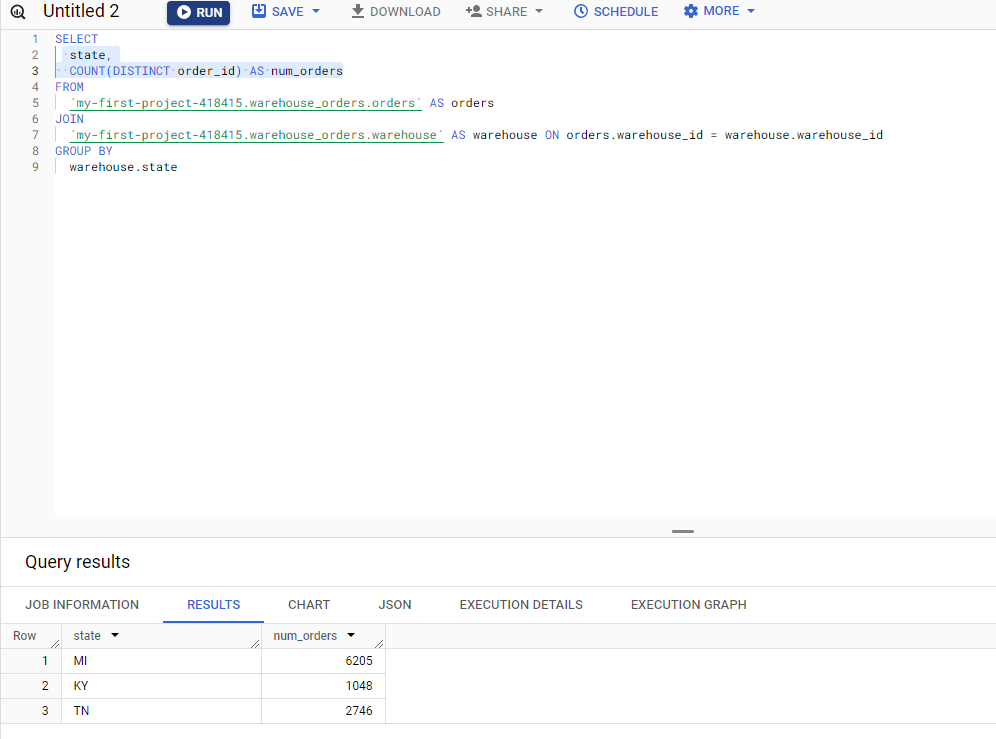
The complete query text should read as follows:



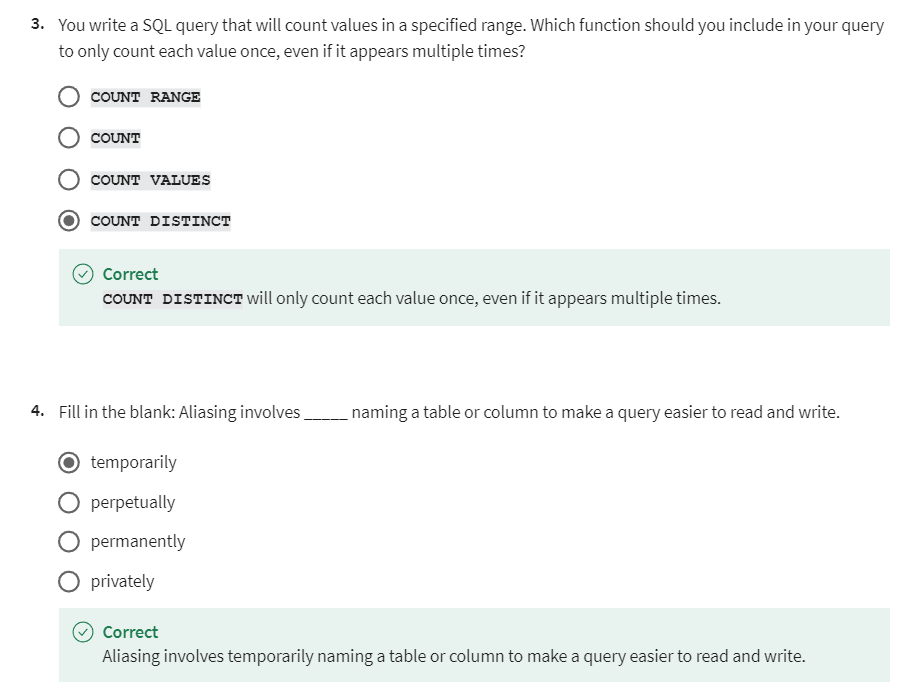
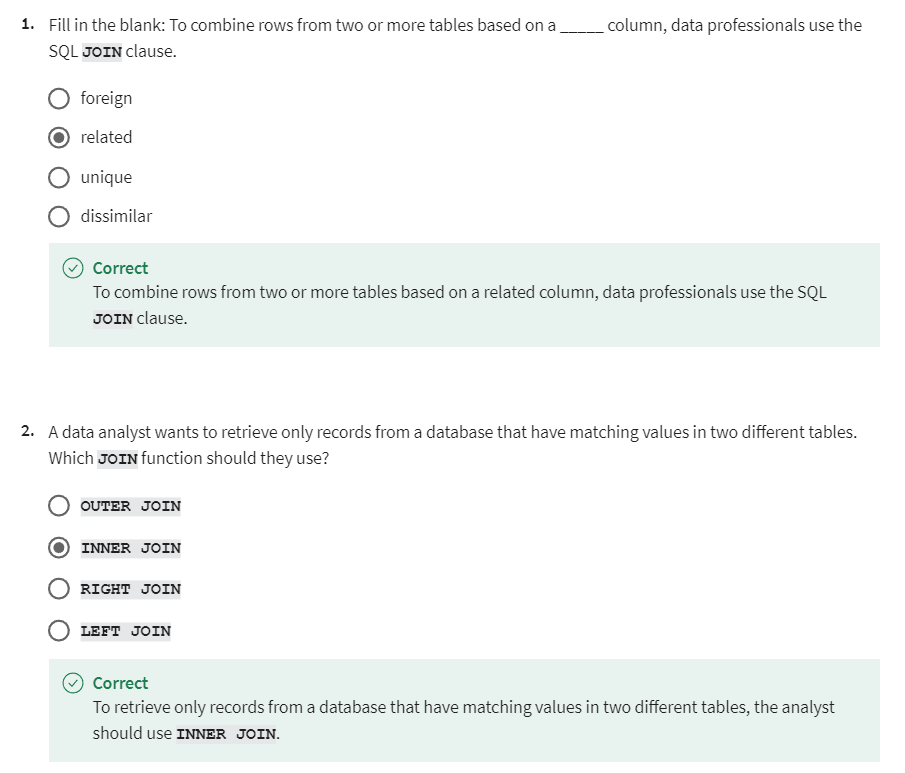
After running the query, there are now three rows listed in the results table: one for each state represented within the **orders** data. The Query results window now displays two columns:

* Column one is state, which indicates which state the warehouse is located in.
* Column two is num\_orders, which indicates the number of orders. These three numbers add up to the count that you ran previously—9,999.

Congratulations! You have successfully executed queries using **COUNT** and **COUNT DISTINCT**, as well as **SELECT**, **FROM**, and **GROUP BY** statements to create aliases and **JOIN** tables, return numerical values within a specific range, and group by specific columns within a table.

You'll find yourself using **COUNT** and **COUNT DISTINCT** during every stage of the data analysis process. Understanding what these queries are and how they are different is crucial in your role as a data analyst.

### [TEST YOUR KNOWLEDGE ON USING JOINS TO AGGREGATE DATA](https://www.coursera.org/learn/analyze-data/quiz/KEW16/test-your-knowledge-on-using-joins-to-aggregate-data)



## WORK WITH SUBQUERIES

### [STEP-BY-STEP: QUERIES WITHIN QUERIES](https://www.coursera.org/learn/analyze-data/supplement/SdGce/step-by-step-queries-within-queries)

This reading outlines the steps the instructor performs in the next video, [Queries within queries](https://www.coursera.org/learn/analyze-data/lecture/yVQoh/queries-within-queries). In the video, the instructor introduces subqueries, another type of SQL query, and demonstrates how to use them to build more complex queries.

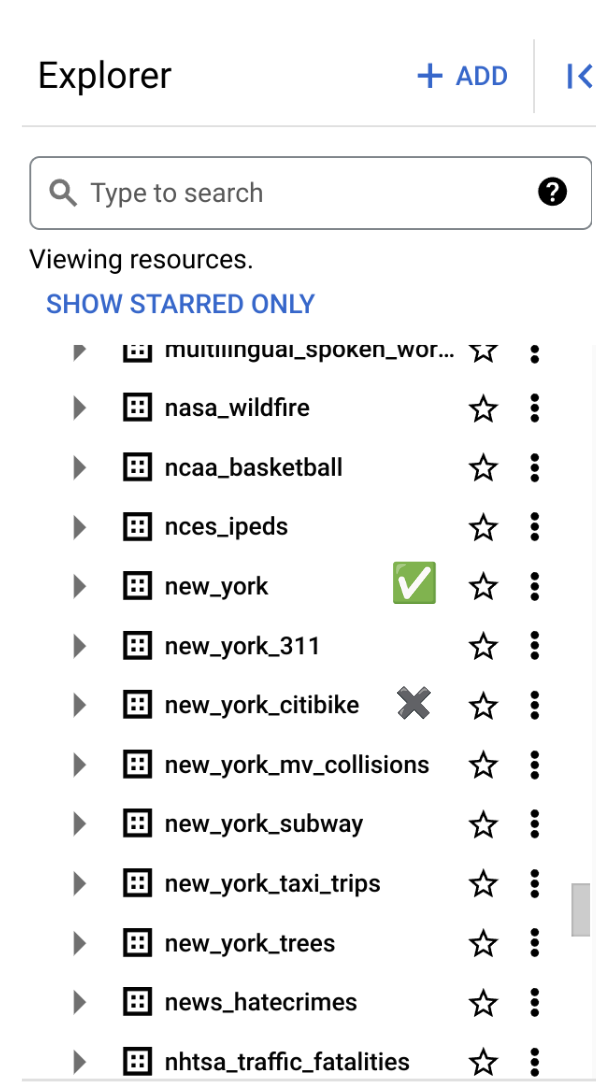
Keep this step-by-step guide open as you watch the video. It can serve as a helpful reference tool if you need additional context or clarification while following the video steps. This is not a graded activity, but you can complete these steps to practice the skills demonstrated in the video.

**What you will need**

In order to follow along with the instructor, you will need to log in to your BigQuery account and access the public database called **new\_york**. From this database, you will use the tables called **citibike\_stations** and **citibike\_trips**.

**Important!**

BigQuery has two different databases that contain very similar information: **new\_york** is one database and **new\_york\_citibike** is another. Both of these databases contain tables called **citibike\_stations** and **citibike\_trips**. However, these tables are not exactly the same between both databases. This step-by-step uses the **new\_york** database.

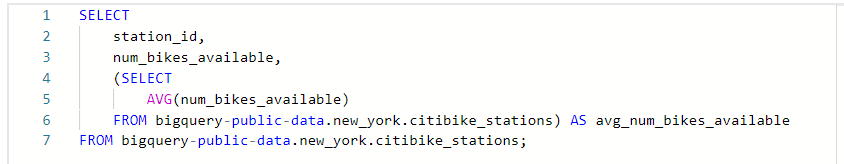


Further, as with many of the public databases in BigQuery, these tables are regularly updated, so if you find that your results do not exactly match the results in the video, this is one probable explanation why.

## **Example 1: Use a subquery in a SELECT statement**

In this query, you will compare the number of bikes available at a particular station to the overall average number of bikes available at all stations.

Type or copy and paste the following query into a new BigQuery query window.

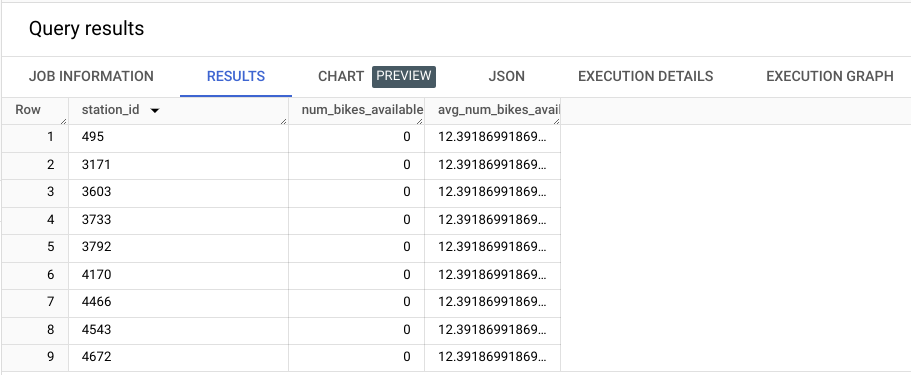


In this example, the subquery was used in a **SELECT** statement. The outer **SELECT** statement (beginning on line 1) lists column names to be retrieved from the **citibike\_stations** table. The inner **SELECT** statement (beginning on line 4) is the subquery, which is used to make a new column that is not already available in the table.

Notice that in the video the **SELECT** statement in lines 4–6 was written first. This is a subquery to calculate the average of the **num\_bikes\_available** column and alias the results as a new column in the results called **avg\_num\_bikes\_available**. The subquery is enclosed in parentheses, which mark it as a subquery.

Then, the whole subquery is incorporated into an outer query. The subquery is indented to the same level as **station\_id** and **num\_bikes\_available**, which are the other columns to be returned in the query results.

So, the final query should return a table containing columns for **station\_id**, **num\_bikes\_available**, and **avg\_num\_bikes\_available**. Here is an example output, but remember, your results might differ due to table updates.

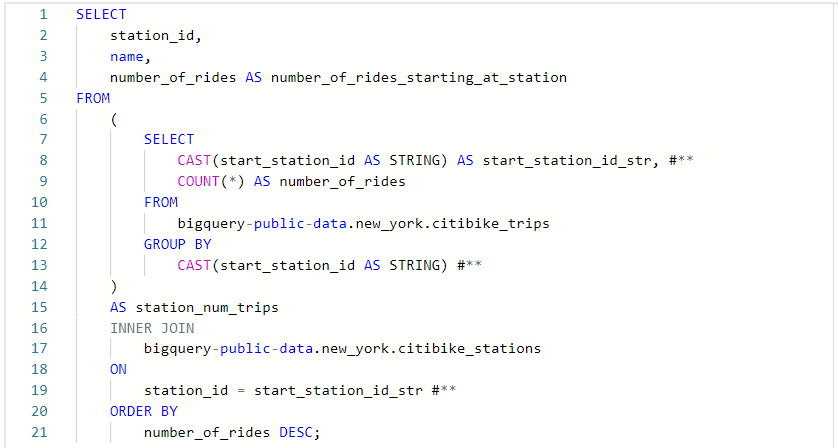


## **Example 2: Use a subquery in a FROM statement**

In this query, you will use the **citibike\_trips** table to calculate the total number of rides that started at each station and return this as a column called **number\_of\_rides\_starting\_at\_station** along with the columns **station\_id** and **name** from the **citibike\_stations** table.

Type or copy and paste the following query into a new BigQuery query window.

**Note:** The lines tagged with **#\*\*** differ from the code in the accompanying video. This is due to changes to the data tables that resulted in a mismatched data type (**Int64** & **STRING**) between the **start\_station\_id** column and the **station\_id** column in the respective tables. To make them the same datatype, the **start\_station\_id** column is converted to **STRING** using the **CAST** keyword.



Here's what's happening in this example. Lines 1–5 are the outer query. They begin with a **SELECT** statement followed by the names of the columns you want returned in the final query results table: **station\_id**, **name**, and **number\_of\_rides\_starting\_at\_station**.

The problem is that the **number\_of\_rides\_starting\_at\_station** column doesn't exist in either table. It must be created. Adding further complication is the fact that the **station\_id** and **name** columns exist in the **citibike\_stations** table, while the information needed to create the **number\_of\_rides\_starting\_at\_station** is in the **citibike\_trips** table.

Lines 6–19 solve this problem. First, notice the subquery from lines 6–14. This subquery is taking the **citibike\_trips** table (line 11) and grouping it by **start\_station\_id** (converted to **STRING**, lines 12–13).

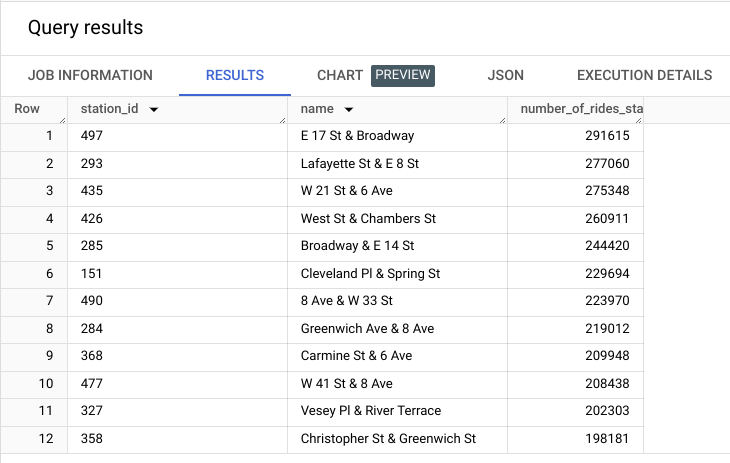
From that grouped data, it's selecting (line 7) the **start\_station\_id** column (converted to string and aliased as **start\_station\_id\_str**, line 8) and the **COUNT** of all rows that begin with each **start\_station\_id**. The count is aliased as a new column called **number\_of\_rides** (line 9). The entire subquery is enclosed in parentheses (lines 6 and 14) and the resulting table is aliased as **station\_num\_trips** (line 15).

**station\_num\_trips** is a helper table. By itself, it contains two columns: **start\_station\_id** and **number\_of\_rides**. There is one ID for each unique station and the corresponding number of rides from that station.

All the data in that subquery came from the **citibike\_trips** table. You still need to connect it to the **citibike\_stations** table. Lines 16–19 make the connection. You **INNER JOIN** (line 16) the **station\_num\_trips** helper table with the **citibike\_stations** table (line 17) using the **station\_id** column in the **citibike\_stations** table and the **start\_station\_id\_str** column in the **station\_num\_trips** helper table as common keys (lines 18–19).

This results in a big table that contains *all* the columns in the **citibike\_stations** table as well as the **start\_station\_id** and **number\_of\_rides** columns from the **station\_num\_trips** helper table. However, you don't need all these columns. You only need three: **station\_id**, **name**, and **number\_of\_rides\_starting\_at\_station**. These are the columns that are selected in lines 1–4.

The final query results should contain these three columns, with rows in descending order by number of rides. Here is an example output, but remember, your results might differ due to table updates.

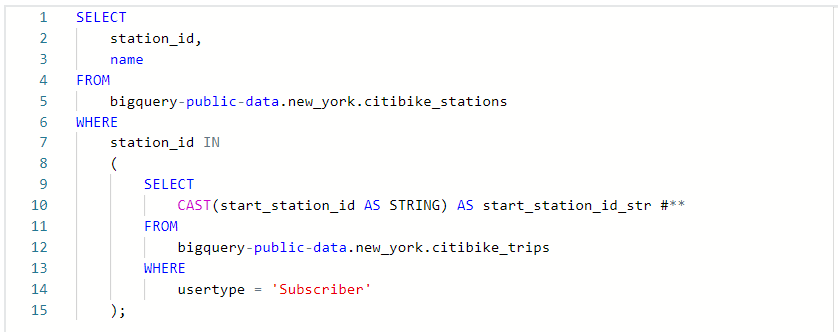


## **Example 3: Use a subquery in a WHERE statement**

Finally, you will write a query that returns a table containing two columns: the **station\_id** and **name** (from the **citibike\_stations** table) of only those stations that were used by people classified as subscribers, which is information found in the **citibike\_trips** table.

Type or copy and paste the following query into a new BigQuery query window.

**Note:** Line 10 (tagged with **#\*\***) differs from the code in the accompanying video. This is due to changes to the data tables that resulted in a mismatched data type (**Int64** & **STRING**) between the **start\_station\_id** column and the **station\_id** column in the respective tables. To make them the same datatype, the **start\_station\_id** column is converted to **STRING** using the **CAST** keyword.



To understand this query, break it into three sections.

### **Section 1:**

The first section begins with lines 8–15. This is the subquery, which is indicated by the parentheses in lines 8 and 15. This segment takes the **citibike\_trips** table (lines 11–12) and filters it using the **WHERE** clause (line 13) so it only contains rows where the **usertype** column contains **Subscriber** as a value (line 14).

From this filtered table, you select the **start\_station\_id**, which is converted to string and aliased as **start\_station\_id\_str** (lines 9–10).

At this point, you have an intermediary table with just a single column—**start\_station\_id\_str**—containing the IDs of every row that had **Subscriber** in the **usertype** column of the original table.

### **Section 2:**

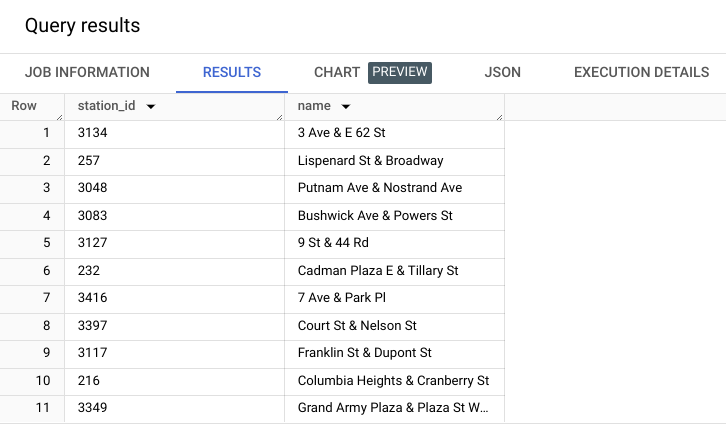
The second section of the query is in lines 4–7. This part uses the information in the intermediary table from section 1 to filter the **citibike\_stations** table. It begins with the full **citibike\_stations** table (line 5). Then, it filters this table using a **WHERE** clause (line 6) so it only contains rows where the values in its **station\_id** column also are found in the list of **start\_station\_id\_str**s that resulted from section 1.

At this point, you now have an intermediary table containing all the columns of the **citibike\_stations** table, but only the rows of stations that were the starting station of a subscriber.

### **Section 3:**

The last part is the simplest. You just need to select the relevant columns from the intermediary table from section 2. This happens in lines 1–4, where you select the **station\_id** and **name** columns and add the **FROM** clause in line 5. Everything you're selecting from follows, which was explained in sections 1 and 2.

The final query results should contain two columns: **station\_id** and **name**. Here is an example output, but remember, your specific results might differ due to table updates.



### [QUERIES WITHIN QUERIES](https://www.coursera.org/learn/analyze-data/lecture/yVQoh/queries-within-queries)

In this video I'll introduce you to another SQL query: subqueries. A subquery is a SQL query that is nested inside of a larger query. Have you ever seen one of those nesting doll toys? They're also known as matryoshka Russian nesting dolls. Subqueries are a lot like nesting dolls.

No, really. Your larger query can have a subquery in it and then that subquery could have a subquery, and then that subquery can have another subquery. But when you stack them all together, they make one query.

With subqueries you can combine different pieces of logic together. Because the logic of your outer query relies on the inner query, you can get more done with a single query. **This means all of the logic is in one place, which makes it more efficient and easier to read**.

The statement containing the subquery can also be called the **outer query** or the **outer select**.

This makes the subquery the inner query or inner select. **The inner query executes first so that the results can be passed on to the outer query to use.**

Subqueries can get a little confusing because there's so many layers. But **if you keep in mind that the innermost query executes first, it'll be easier to order your subqueries when you want them to execute**.

**Subqueries can also be nested inside all sorts of other queries**. Usually you'll find subqueries nested in FROM or WHERE clauses. Let's try out some common subqueries.

We'll start with a subquery in a SELECT statement using the bike-sharing data from an earlier example. For the first statement, let's say **we want to compare the number of bikes available at a station to the average number of bikes available**. We're going to use this query to pull the average number of bikes available. Then we're going to incorporate it as a subquery.



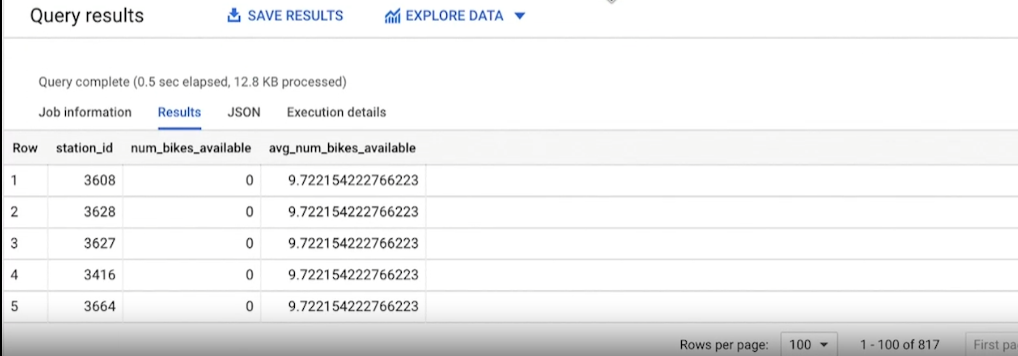
Now let's build our outer SELECT query. We want to select the station ID and the number of bikes available. Then we'll put the SELECT query that's pulling the average number of bikes inside that outer query by using parentheses. We'll also build FROM into the subquery before closing it with another parenthesis and completing the outer query.



The end of the outer join query has AS to show what we want to call this column and a final FROM statement to indicate which table we're referring to.



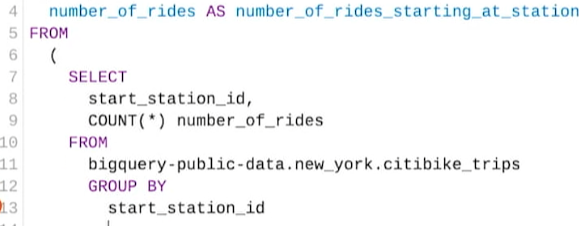
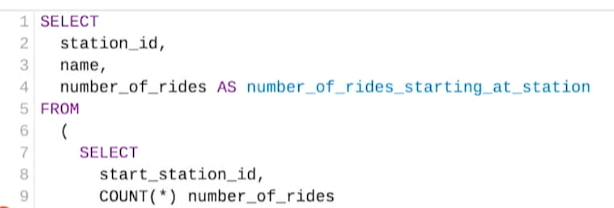
Now let's run it. And there!



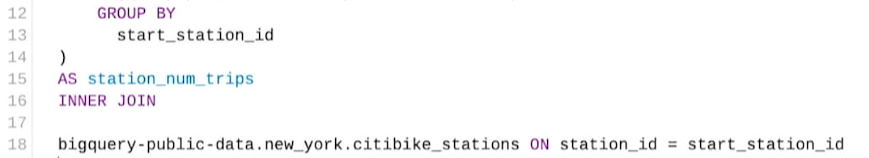
We've got a table with both the number of bikes available and the average number of bikes available at different stations.

It's really common to see subqueries nested in FROM and WHERE statements. So let's try those next.

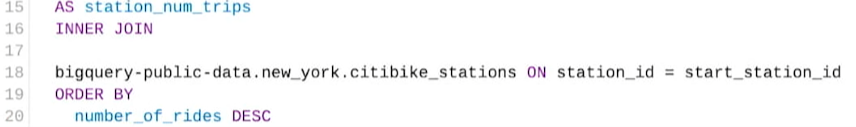
We could use a FROM statement to calculate the number of rides that have started at each station over time. We'll start with our outer query and input SELECT station\_id, name, and number\_ of\_ rides. We'll use AS to tell it how we want the table labeled, and FROM to tell it where we're pulling data from. But before we finish that query, we'll add a subquery. We'll put our parenthesis here and then SELECT the start\_station\_id. Then we can tell it to COUNT the number\_of \_rides FROM the trip data and group it by the start\_station\_id.



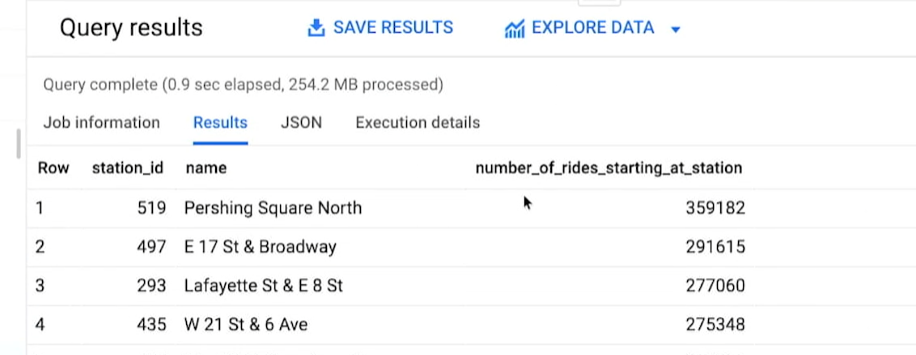
After that, we'll close the subquery with a parenthesis so that we can continue building the outer query. We'll use AS again and then use INNER JOIN and ON to join it with the station ID data.



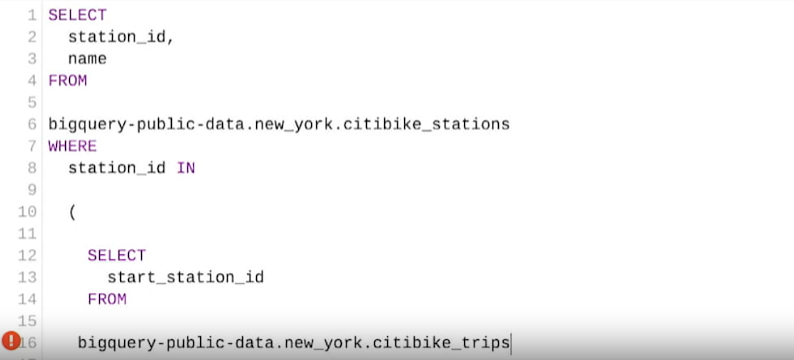
Finally we'll tell it to put it in descending order.



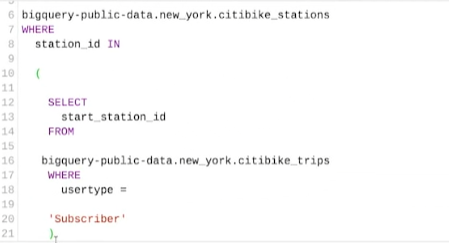
Let's see what happens when we run that. We now have the number of rides started at each station.



**One last example**. Let's use a WHERE statement. The bike-sharing company has two kinds of users: subscribers and one-time customers. Let's say we wanted a list of stations subscribers used.   
  
As always, we start with the outer query. SELECT the station\_ id and name FROM the public dataset we're using. This time we'll use a WHERE statement. We'll also use IN so that we can specify multiple values and this WHERE statement. Then we'll put our subquery in the parenthesis. We'll add SELECT, FROM, and WHERE again. But this time we'll tell it that we only want data on specific customers.



It's good to note that you can use comparison operators in subqueries, even multiple row operators like IN, ANY, or ALL. In this case we'll use equals to indicate that we only want the subscriber user data.



Now let's run the query and we've got the station id and names for stations that fit our criteria.



**That's subqueries in action.** Subqueries can be challenging. There's a lot of layers to think through and you might find yourself running into errors when you practice. That's totally okay. Having to go through that challenge means you're growing. **If everything was easy, we wouldn't find new ways to grow.**

For me it's all about how much work and how much time I need to put in to do it. Give yourself time to practice this new concept.

Coming up you'll get a chance to use subqueries to aggregate data or you can move on to the weekly challenge. You'll take everything you've learned, using VLOOKUP, different JOINS, and subqueries and apply it to this upcoming assessment. We've been doing a lot of complex work. If you want to take a moment to review these videos before moving on, feel free. Once you've finished a challenge, I'll see you again for our next big learning adventure. See you soon.

### [HANDS-ON ACTIVITY: USE SUBQUERIES TO REFINE DATA](https://www.coursera.org/learn/analyze-data/quiz/nXD39/hands-on-activity-use-subqueries-to-refine-data)

## **Activity Overview**

****

As you’ve learned, a subquery is a query that is nested inside of another query. The subquery filters or sorts data to prepare it to be used by the outer query to produce its final result. This allows data professionals to create more nuanced queries that provide specific insights from the data!

For example, perhaps a data analyst working in human resources has been asked to determine the average salary of employees working within a specific department. The analyst can use a subquery to first find the total salary and number of employees within the department. Then, the outer query will use these numbers to calculate the average salary within a department. It's a step-by-step process, each step relying on the one before it.

In this activity, you will practice using **SELECT** statements with **FROM**, **WHERE**, and **GROUP BY** clauses to build your subqueries. By the time you complete this activity, you will be able to create a subquery using **SELECT** statements with **FROM**, **WHERE**, and **GROUP BY** clauses and analyze the query’s results.

### 

### **Scenario**

Review the following scenario. Then complete the step-by-step instructions.

You work for an organization that is responsible for the safety, efficiency, and maintenance of transportation systems in your city. You have been asked to gather information around the use of Citi Bikes in New York City. This information will be used to convince the mayor and other city officials to invest in a bike sharing and rental system to help push the city toward its environmental and sustainability goals.

To complete this task, you will create three different subqueries, which will allow you to gather information about the average trip duration by station, compare trip duration by station, and determine the five stations with the longest mean trip durations.

### 

### **Step-By-Step Instructions**

### 

### **Step 1: Access the dataset**

For this activity, you will need to log in to your BigQuery account to use the public new\_york\_citibike dataset. Then, use the table called citibike\_trips. For a refresher on how to access this data, refer to the optional reading [Prepare to use the bike sharing dataset in BigQuery](https://www.coursera.org/learn/analyze-data/supplement/vYBY0/optional-prepare-to-use-the-bike-sharing-dataset-in-bigquery).

Note: Many of the public databases on BigQuery are living records and, as such, are periodically updated with new data. Throughout this course (and others in this certificate program), if your results differ from those you encounter in videos or screenshots, there's a good chance it is due to a data refresh.

### 

### **Step 2: Create a subquery to find the average trip duration by station**

First, find the average trip duration for a particular station. This subquery will be applied to the **SELECT** statement, but first you need to build the outer query:

1. Select COMPOSE A NEW QUERY from the SQL workspace in BigQuery. Or, if you have the new\_york\_citibike.citibike\_trips table open, select Query from the table's menu to compose a new query.

2. In line 1, enter **SELECT**, then press Enter/Return.

3. In line 2, press Tab, then enter **subquery.start\_station\_id** followed by a comma. Then press Enter/Return.

4. In line 3, enter **subquery.avg\_duration**, then press Enter/Return.

Lines 1–3 in the query should read as follows:



This **SELECT** statement is used to create the outer query. The fields identified in lines 2 and 3 allow the **SELECT** statement to function similarly to a table with an alias, e.g. **SELECT alias.column\_name1, alias.column\_name2**. This relies on the subquery below to populate the Query results table.

5. In line 4, press Backspace to remove the indentation to align the text with the SELECT statement. Enter FROM, then press Enter/Return.

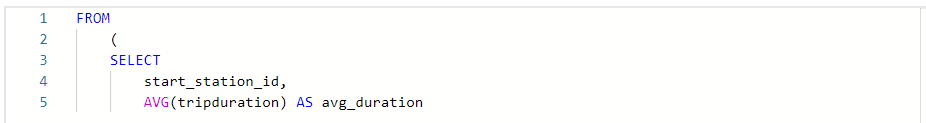
6. In line 5, press Tab, then enter an open parenthesis [(]. Then press Enter/Return.

7. In line 6, press Tab, then enter SELECT. Then press Enter/Return.

8. In line 7, press Tab, then enter start\_station\_id followed by a comma. Then press Enter/Return.

9. In line 8, enter AVG(tripduration) as avg\_duration. Then press Enter/Return.

Lines 4–8 in the query should read as follows:



The subquery is created by using another **SELECT** statement, located within a **FROM** clause. It is used to determine the average trip duration by station. The station is indicated by the **start\_station\_id**. The average is calculated using the **AVG** function and is given the alias **avg\_duration** using **AS**. This will appear as a column title in the results table after you run the query.

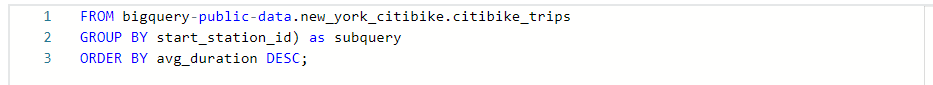
10. In line 9, press Backspace to remove the indentation to align the text with the **SELECT** statement.

11. Still in line 9, enter **FROM bigquery-public-data.new\_york\_citibike.citibike\_trips**, then press Enter/Return.

12. In line 10, enter **GROUP BY start\_station\_id) as subquery;** Notice that this line closes the parentheses that opened in line 5.

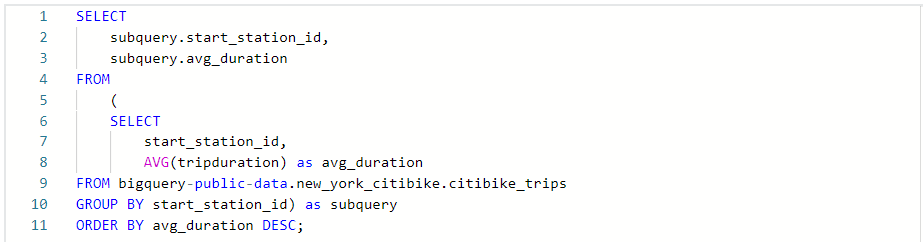
13. In line 11, add **ORDER BY avg\_duration DESC;**.

Lines 9, 10, and 11 in the query should read as follows:



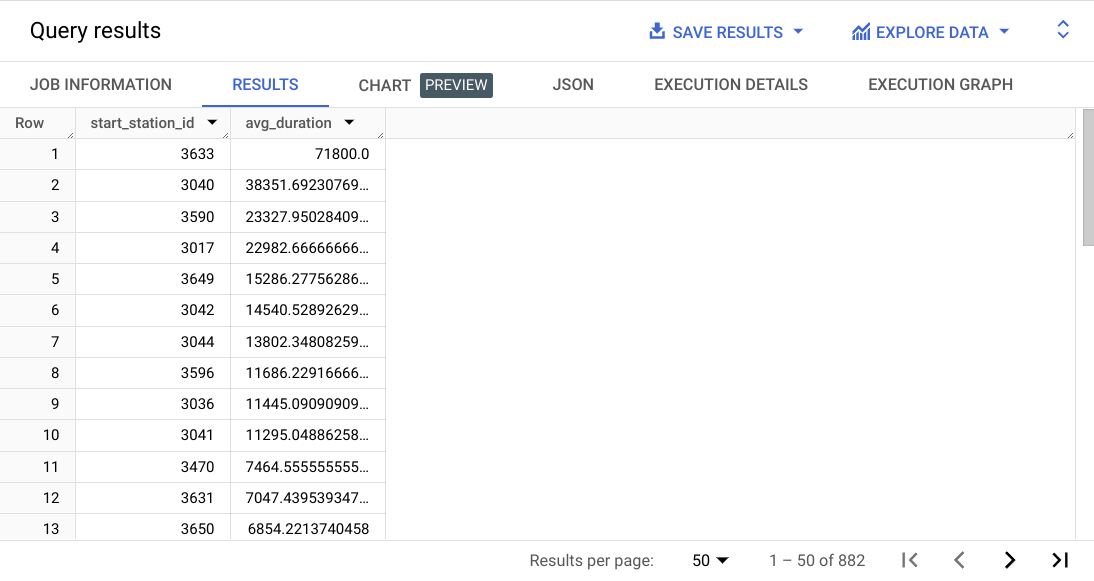
**FROM** will tell the subquery where to pull the data from. In this case, it’s pulling from the **citibike\_trips** table. **GROUP BY** will group the rows by columns. Notice how it has the alias, **subquery**. This is what links to the two fields designated to be the titles of the results columns, as indicated in the outer query.

The complete query should read as follows:



13. Select RUN. The results will appear in the Query results window.

Note: Your results should contain two columns: **start\_station\_id** and **avg\_duration**, however, the specific rows may differ from what is depicted here due to periodic refreshes of the source tables on BigQuery.



Take some time to examine the query results. In this case there are 882 lines of data that you can use to analyze and compare the average trip durations between each station. You will continue to work with the average trip duration per station in the next query.

### 

### **Step 3: Create a subquery to compare trip duration by station**

Create a new query to compare the average trip duration per station to the overall average trip duration from all stations. This will provide insights about how long people typically use the bikes that they get from a particular station in comparison to the overall average. This subquery will be applied to the FROM clause in a SELECT statement. To begin:

1. Select the blue plus sign tab to compose a new query.

2. In line 1, enter **SELECT**, then press Enter/Return.

3. In line 2, press Tab, then enter **starttime** followed by a comma. Then press Enter/Return.

4. In line 3, enter **start\_station\_id** followed by a comma, then press Enter/Return.

5. In line 4, enter **tripduration** followed by a comma, then press Enter/Return.

Lines 1–4 in the query should read as follows:



To begin the query, note there are three fields from which to gather data in order to request the average trip durations. Here, use **starttime** (a somewhat unique identifier for the results) to mark when trips from a particular station started; the **start\_station\_id**, which identifies the station ID for each trip; and the **tripduration**, which measures the length of each trip in seconds. These will be the first three columns in your results table.

You'll generate two additional columns in the results table: **avg\_duration\_for\_station** and **difference\_from\_average**. Begin with the subquery for **avg\_duration\_for\_station**:

6. In line 5, enter an open parentheses **(**

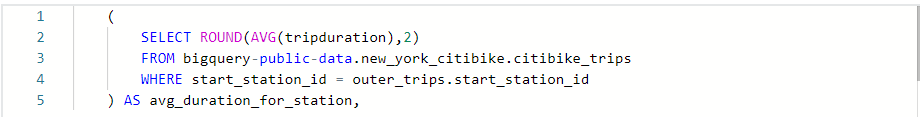
7. In line 6, enter **SELECT ROUND(AVG(tripduration), 2)**

8. In line 7, enter **FROM bigquery-public-data.new\_york\_citibike.citibike\_trips**.

8. In line 8, enter **WHERE start\_station\_id = outer\_trips.start\_station\_id**.

9. In line 9, close the parentheses **)** that were opened in line 5, making sure that both align in indentation. Then, on the same line, name the column by typing **AS avg\_duration\_for\_station,**

Lines 5–9 in the query should read as follows:



A **SELECT** statement is used to begin a subquery that will return the average trip duration for each station. Notice that the **ROUND** function is used to round the numerical value of each trip duration to the nearest hundredth of a second. (Remember: trip duration is measured in seconds.) **FROM** indicates that you want the query to pull from the **citibike\_trips** table within the **new\_york\_citibike** dataset inside the **bigquery-public-data** database. **WHERE** tells the query to link the **start\_station\_id** with the output of the query: a new column labeled **avg\_duration\_for\_station**.

Now, begin the subquery to create the second new column in the results table: **difference\_from\_avg**.

10. In line 10, enter **ROUND(tripduration - (**. Make sure it begins on the same indentation level as the previous columns that you selected.

11. In lines 11, 12, and 13 enter:

**SELECT AVG(tripduration)**

**FROM bigquery-public-data.new\_york\_citibike.citibike\_trips**

**WHERE start\_station\_id = outer\_trips.start\_station\_id), 2) AS difference\_from\_avg**

This closes the subquery that creates the **difference\_from\_avg** column. Now, what remains is to close out the outer query.

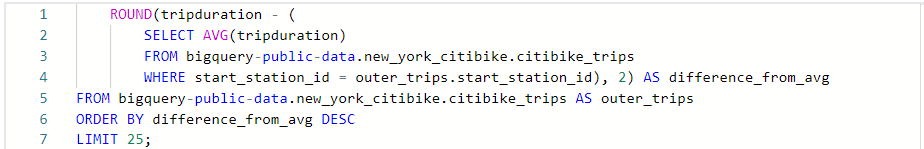
12. In lines 14, 15, and 16 outdent all the way back to the left and enter:

**FROM bigquery-public-data.new\_york\_citibike.citibike\_trips AS outer\_trips**

**ORDER BY difference\_from\_avg DESC**

**LIMIT 25;**

Lines 10–16 in the query should read as follows:

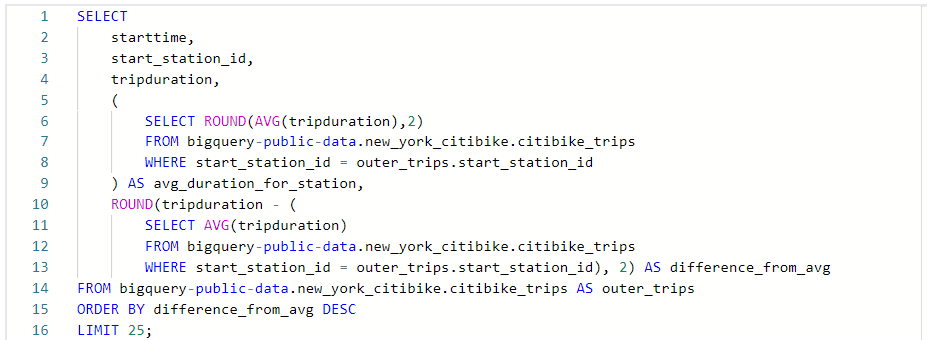


A second **SELECT** statement is used to return the difference between the specific station's average trip duration and the overall average trip duration. Notice that the **ROUND** function is used again to round to the nearest hundredth of a second. The average trip duration is gathered from the citibike\_trips table using a second **FROM** clause. **WHERE** tells the query to link the start\_station\_id with the output of the query: a new column labeled difference\_from\_avg.

This will contain the difference between the overall average trip duration and the average trip duration per station. The third **FROM** clause pulls the data from the citibike\_trips table and labels it as **outer\_trips**. **LIMIT** is used to limit the output in the results table to 25 results. Depending on the amount of information you need from the dataset, you can change this number as needed to increase or decrease the results, or you can remove it completely to see all results.

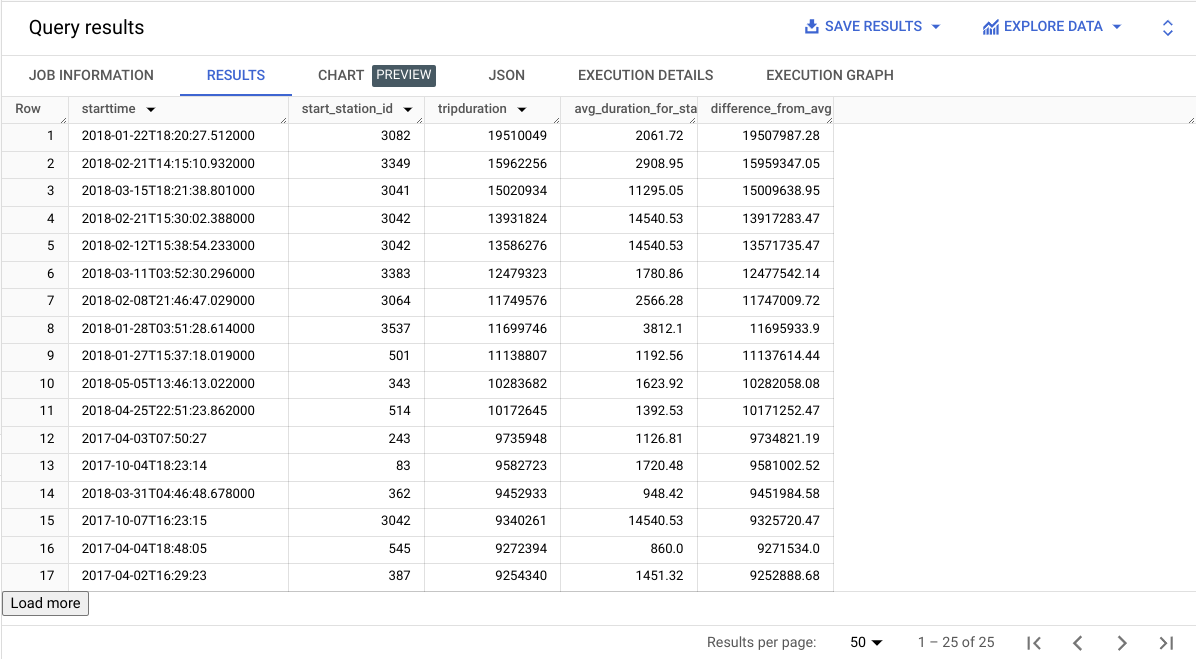
Note: If you remove the limit, it may take longer for the query to run. You may also have to perform another query to find specific data.

The complete query should read as follows:



13. Select RUN. The results will appear in the Query results window.

Note: Your results should contain five columns: **starttime**, **start\_station\_id**, **tripduration**, **avg\_duration\_for\_station**, and **difference\_from\_avg**, however, the specific rows may differ from what is depicted here due to periodic refreshes of the source tables on BigQuery.



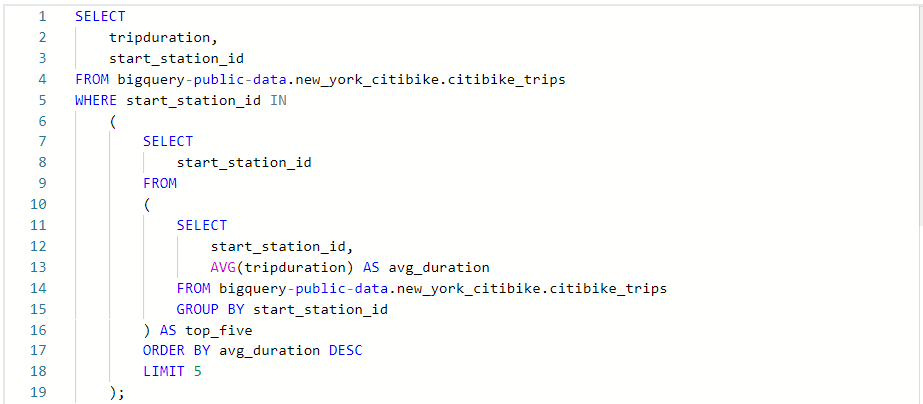
14. Examine the results table. In the last column, difference\_from\_avg, there are some very large differences from the average duration, indicating that these stations have some significant outliers. It would probably be worthwhile to examine that further.

15. To continue with this activity, you can save this query so you can come back to it later, then create a new query. Alternatively, you can overwrite the existing query.

### **Step 4: Create a subquery to determine the five stations with the longest mean trip duration**

Now, you'll compose a new query to filter the data to include only the trips from the five stations with the longest mean trip duration.

Write the following code and consider what it's doing:



Lines 1–4: This query begins by identifying the columns that will appear in the query results table: **tripduration** and **start\_station\_id**.

However, you don't want all of the records to be selected. You only want those records where the **start\_station\_id** matches one of the top five stations with the greatest average **tripduration**. To find this, you need a subquery.

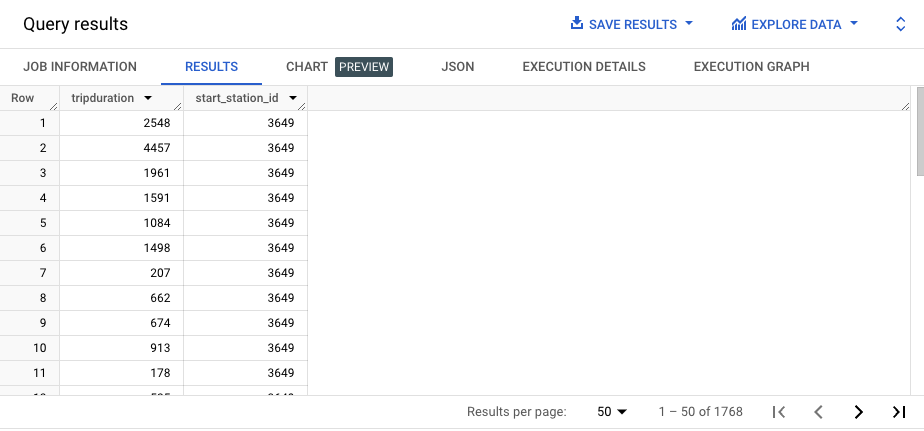
Line 5: This is the beginning of the subquery. It uses an **IN** operator to filter records based on whether the **start\_station\_id** is in a list of **start\_station\_id**s produced by the subquery.

Lines 6–19: This is an inner subquery. It creates a derived table called **top\_five** by performing the following steps:

1. It selects **start\_station\_id** from the main table.
2. For each **start\_station\_id**, it calculates the average **tripduration** and labels this average as **avg\_duration**.
3. It groups the results by **start\_station\_id**, meaning the average trip duration is calculated separately for each unique **start\_station\_id**.
4. From this grouped and aggregated data, it orders the results by **avg\_duration** in descending order.
5. It limits the results to the top five. In other words, it only keeps the records for the five **start\_station\_id**s with the highest average trip durations.

The result of the entire query is a list of records from the main table—specifically the **tripduration** and **start\_station\_id** for each record, but only those for records where the **start\_station\_id** is among the five stations with the greatest average trip durations.   
  
If you examine the query results, you will discover that only five of the **start\_station\_id** values are listed in column two. The output of the query should appear similar to this:

Note: Your results should contain two columns: **tripduration** and **start\_station\_id**, however, the specific rows may differ from what is depicted here due to periodic refreshes of the source tables on BigQuery.



Column one is tripduration; column two is start\_station\_id.

Congratulations! You successfully built and ran three different sets of queries with subqueries.

SELECT

starttime,

start\_station\_id,

tripduration,

(

SELECT ROUND(AVG(tripduration),2)

FROM bigquery-public-data.new\_york\_citibike.citibike\_trips

WHERE start\_station\_id = outer\_trips.start\_station\_id

) AS avg\_duration\_for\_station,

ROUND(tripduration - (

SELECT AVG(tripduration)

FROM bigquery-public-data.new\_york\_citibike.citibike\_trips

WHERE start\_station\_id = outer\_trips.start\_station\_id), 2) AS difference\_from\_avg

FROM bigquery-public-data.new\_york\_citibike.citibike\_trips AS outer\_trips

ORDER BY difference\_from\_avg DESC

LIMIT 25;

### [STEP-BY-STEP: USE SUBQUERIES TO AGGREGATE DATA](https://www.coursera.org/learn/analyze-data/supplement/gtTrS/step-by-step-use-subqueries-to-aggregate-data)

This reading provides you with the steps the instructor performs in the following video, [Use subqueries to aggregate data](https://www.coursera.org/learn/analyze-data/lecture/JjPZ5/using-subqueries-to-aggregate-data). The video teaches you how to aggregate or combine data using subqueries in SQL.

Keep this step-by-step guide open as you watch the video. It can serve as a helpful reference tool if you need additional context or clarification while following the video steps. This is not a graded activity, but you can complete these steps to practice the skills demonstrated in the video.

**What you’ll need**

In order to follow along with the instructor, you will need the **warehouse\_orders** dataset uploaded into your project space. From that dataset, you'll need the **warehouse** and **orders** tables. If you haven’t already uploaded this data, follow the instructions in the [Upload the warehouse dataset to BigQuery](https://www.coursera.org/learn/analyze-data/supplement/HuXCc/optional-upload-the-warehouse-dataset-to-bigquery) reading.

## 

## **Objective**

The objective of this query is to aggregate the data into a table containing each warehouse's ID, state and alias, and number of orders; as well as the grand total of orders for all warehouses combined; and finally a column that classifies each warehouse by the percentage of grand total orders that it fulfilled: 0–20%, 21-60%, or > 60%.

**Note:** This activity breaks out the steps into manageable chunks. The final query is only intended to be run at the end. If you try to run the query before reaching the end of this guide you will likely get an error.

## **Example: Combine and alias the tables**

As a refresher, aliasing is when you temporarily name a table or column in your query to make it easier to read and write. To alias the **warehouse** and **orders** tables and join the tables, follow these steps. Remember, these statements require that you enter your unique individual project name or else they won't run. Be sure to substitute your project name in the code wherever you encounter **your-project** written. If you haven't explicitly assigned a project name, BigQuery generates one for you automatically. It typically looks like two words and a number, each separated by a hyphen, for example **august-west-100777**.

Begin with the **FROM** statement a few rows down. Later, you'll return to the top of the query to fill it in.

1. In row 3, enter **FROM your-project.warehouse\_orders.warehouse AS Warehouse**
2. In row 4, enter **LEFT JOIN your-project.warehouse\_orders.orders AS Orders**
3. In row 5, enter **ON Orders.warehouse\_id = Warehouse.warehouse\_id**

These statements will combine the two tables (**warehouse** and **orders**) using **warehouse\_id** as the common key (the column shared by both tables).

## **Example: Organize your new table**

Use the **GROUP BY** clause in SQL to group rows that have the same values in specified columns into aggregated data, such as sum, count, average, maximum, or minimum, based on the values in another column. This operation is particularly useful in databases where there is a need to analyze data based on certain criteria.

1. In row 6, enter **GROUP BY**
2. In row 7, enter **Warehouse.warehouse\_id,**
3. In row 8, enter **warehouse\_name**

Here, the combined table is grouped first by the warehouse ID and then by its name.

## **Example: Build subquery logic**

Now that you have the **FROM** statement and **JOIN**, go back up to the first lines and define the rows to select and operations to perform on them. From the objective, you know you want to return **five columns**: each warehouse's ID (**warehouse\_id**—column 1), state and alias (this info will be combined into a single column: **warehouse\_name**— column 2), and number of orders (**number\_of\_orders**—column 3); as well as the grand total of orders for all warehouses combined (**total\_orders**—column 4); and finally a column that classifies each warehouse by the percentage of grand total orders that it fulfilled: 0–20%, 21-60%, or > 60% (**fulfillment\_summary**—column 5).

Above everything you've written so far, write:

1. In row 1, enter **SELECT**
2. In row 2, enter **Warehouse.warehouse\_id,** # (This is the first column.)
3. In row 3, enter **CONCAT(Warehouse.state, ': ', Warehouse.warehouse\_alias) AS warehouse\_name,** # (This is the second column. Notice you're concatenating two existing columns into a new one)
4. In row 4, enter **COUNT(Orders.order\_id) AS number\_of\_orders,** # (This is the third column.)
5. In row 5, enter **(SELECT COUNT(\*) FROM your-project.warehouse\_orders.orders AS Orders) AS total\_orders,** # (This is the fourth column.)

To create the final column, you'll need to use a special keyword.

## **Example: Create categories using CASE**

Use the **CASE** keyword in SQL to create categories or group data based on specific conditions. This is valuable when dealing with numerical or textual data that needs to be segmented into different groups or categories for analysis, reporting, or visualization purposes.

For the final column, you'll use **CASE** to define which label to apply to each warehouse's fulfillment percentage (the percentage of the grand total of orders that it fulfilled). There will be three conditions, and thus three possible labels: "Fulfilled 0–20% of Orders", "Fulfilled 21–60% of Orders", or "Fulfilled more than 60% of Orders".

1. In row 6, enter **CASE**
2. In row 7, enter **WHEN COUNT(Orders.order\_id)/(SELECT COUNT(\*) FROM your-project.warehouse\_orders.orders AS Orders) <= 0.20** # (This defines the first possible condition.)
3. In row 8, enter **THEN 'Fulfilled 0-20% of Orders'** # (**THEN** defines the label to apply when the first condition is true.)
4. In row 9, enter **WHEN COUNT(Orders.order\_id)/(SELECT COUNT(\*) FROM your-project.warehouse\_orders.orders AS Orders) > 0.20** # (This is the first part of the second condition.)
5. In row 10, enter **AND COUNT(Orders.order\_id)/(SELECT COUNT(\*) FROM your-project.warehouse\_orders.orders AS Orders) <= 0.60** # (This is the second part of the second condition.)
6. In row 11, enter **THEN 'Fulfilled 21-60% of Orders'** # (This defines the label to apply when the second condition is true.)
7. In row 12, enter **ELSE 'Fulfilled more than 60% of Orders'** # (This defines the label to apply when neither of the first two conditions is true.)
8. In row 13, enter **END AS fulfillment\_summary** # (The **END** keyword terminates the **CASE** declaration. Then the **AS** keyword indicates what the resulting column should be named.)

## **Example: Filter using HAVING**

Use the **HAVING** clause in SQL in combination with the **GROUP BY** clause to filter the results of aggregate functions in a query. While the **WHERE** clause filters individual rows *before* they are grouped, the **HAVING** clause filters groups of rows *after* they have been grouped. To filter out the warehouses that are currently being built (and therefore have no orders), enter the following lines below everything you've written so far:

1. In row 20, enter **HAVING**
2. In row 21, enter **COUNT(Orders.order\_id) > 0**

Here is the final query:  
  
SELECT

Warehouse.warehouse\_id,

CONCAT(Warehouse.state, ': ', Warehouse.warehouse\_alias) AS warehouse\_name,

COUNT(Orders.order\_id) AS number\_of\_orders,

(SELECT COUNT(\*) FROM your-project.warehouse\_orders.orders AS Orders) AS total\_orders,

CASE

WHEN COUNT(Orders.order\_id)/(SELECT COUNT(\*) FROM your-project.warehouse\_orders.orders AS Orders) <= 0.20

THEN 'Fulfilled 0-20% of Orders'

WHEN COUNT(Orders.order\_id)/(SELECT COUNT(\*) FROM your-project.warehouse\_orders.orders AS Orders) > 0.20

AND COUNT(Orders.order\_id)/(SELECT COUNT(\*) FROM your-project.warehouse\_orders.orders AS Orders) <= 0.60

THEN 'Fulfilled 21-60% of Orders'

ELSE 'Fulfilled more than 60% of Orders'

END AS fulfillment\_summary

FROM your-project.warehouse\_orders.warehouse AS Warehouse

LEFT JOIN your-project.warehouse\_orders.orders AS Orders

ON Orders.warehouse\_id = Warehouse.warehouse\_id

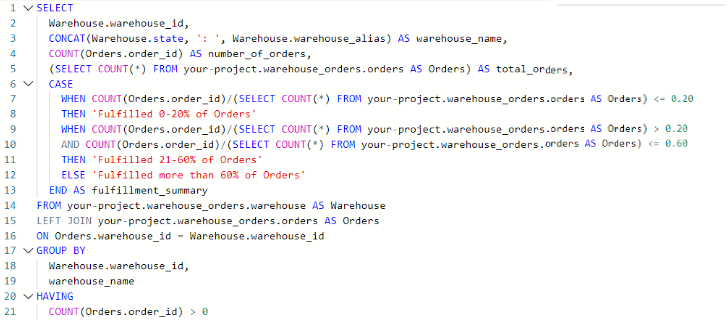
GROUP BY

Warehouse.warehouse\_id,

warehouse\_name

HAVING

COUNT(Orders.order\_id) > 0

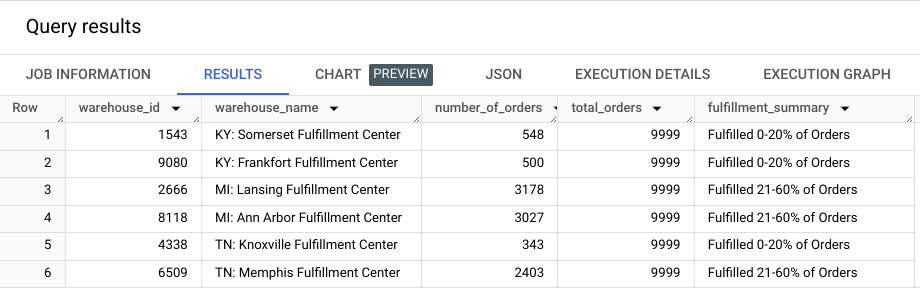


## **Example: Run the new query**

It’s important to run the new queries that you write to test that they are working correctly.

1. Select the **Run** button
2. Now, you can identify what percent of our company’s total orders are being fulfilled by each warehouse

The query results should be:



### [USE SUBQUERIES TO AGGREGATE DATA](https://www.coursera.org/learn/analyze-data/lecture/JjPZ5/use-subqueries-to-aggregate-data)

Now it's time to talk about how to aggregate data with subqueries. Coming up, we'll learn about some new subquery statements and use them to aggregate data.

**The query we're going to build in this video is pretty advanced**. It's going to be a little complicated, but I know you've got what it takes. Let's get started. We've used functions like WHERE to filter our data before, but the WHERE function can't be used with aggregate functions. For example, you can use WHERE on a statement and follow it with GROUP BY. But when you want to use GROUP BY first and then use WHERE on that output, you'll need a different function.

This is where HAVING comes in. **HAVING basically allows you to add a filter to your query instead of the underlying table when you're working with aggregate functions**. That way it only returns records that meet your specific conditions.

Similarly, **CASE returns records with your conditions by allowing you to include if/then statements in your query**. Let's try to aggregate our data with subqueries and test out these new functions.

Let's say we're working with a company that makes socks that we talked about earlier. We've been asked to calculate what percentage of the orders are fulfilled by each warehouse. **Basically, we're interested in knowing which warehouses are delivering the most orders.** We've seen these tables before, but as a quick refresher, here's the Orders table.

You can see the columns here: order\_id, customer\_id, warehouse\_id, order\_date, and ship\_date. If we pull up the warehouse table, we can check out its columns. We have the warehouse\_id, warehouse\_alias, the maximum\_capacity, the total number of employees, and the state where the warehouse is located.

Before we start building the rest of our query, we'll want to alias our table names. As a reminder, **aliasing is when you temporarily name a table or column in your query to make it easier to read and write**. This example query is a little bit more complicated than the ones we've seen before. Aliasing will help save us some time. We'll start by aliasing the Warehouse table in our FROM statement. The FROM statement in this query is near the end, but we'll build this first so that we can use the alias everywhere else. We'll simplify it to just Warehouse for the rest of this query. We know that we're going to JOIN these tables together. Let's add that while we're working on this part of the query anyway. We're using a LEFT JOIN here because we want all the information from our Warehouse data, even if it doesn't show up in the Orders table. Then we'll alias the Orders table as part of this statement.

Now both of our tables have temporary names we can use. We've already finished a JOIN statement. But before we can build the beginning of this query, let's go ahead and add our GROUP BY statement after this JOIN. We'll group these by the warehouse\_id and name. Now we'll go back to the beginning of the query. We'll select the warehouse \_id. Then we'll use CONCAT to combine the strings with the warehouse's state and alias AS the warehouse name. Then we'll use COUNT to get the number of orders per warehouse.

Next, we'll build in a subquery to pull the total number of orders placed across all warehouses. We'll input SELECT again and then write the subquery in parentheses. We'll put an asterisk after COUNT to indicate that we want to include everything from the Orders table. Finally, we'll close out the subquery and use AS to name this column total\_orders.

Now that our subquery logic is complete, we can use a CASE statement to create categories for our warehouses based on how many orders they will fulfill. We'll represent these as percentages. You should notice COUNT in the statement a few times. We'll start by saying WHEN the number of orders FROM our Orders table is less than or equal to 0.2, THEN the table will say "Fulfilled 0-20% of Orders." Then we'll use WHEN again to indicate that when the number of orders is greater than 0.2 and less than or equal to 0.6,

it'll say, "fulfilled 21-60% of Orders." After that, we can use ELSE to have everything that doesn't meet the criteria of our CASE statement say, "Fulfilled more than 60% of Orders." Then we'll use END AS to name this column fulfillment\_summary. That brings us back to the portion of the query we've already written. But we're going to add a HAVING statement at the very end of this query. Our Warehouse table has warehouses that are currently being built, and we want to filter those out since they aren't fulfilling orders yet. We can use HAVING to only include warehouses that have at least one order.

Now, before we execute this query, let's take a moment to look at the whole thing. We have an outer SELECT, a COUNT subquery, a CASE statement, a JOIN and HAVING, all wrapped into one query. We've built a really complex query. So let's run it to see the new table. There. Now we can easily identify what percent of our company's total orders are being fulfilled by each warehouse. These warehouses met our criteria. And we can see here in the fulfillment\_summary column the percentage categories we outlined in our CASE statement. Obviously, since we included a HAVING statement to specify only warehouses with at least one order, there aren't any warehouses currently under construction in this table. That really complicated query we wrote created this specific table of data we can use to easily compare how these warehouses are performing.

Clauses like HAVING and CASE paired with subqueries will help you build more and more complex queries, which lets you do more and more complex things in SQL.

### [SQL FUNCTIONS AND SUBQUERIES: A FUNCTIONAL FRIENDSHIP](https://www.coursera.org/learn/analyze-data/supplement/SthVU/sql-functions-and-subqueries-a-functional-friendship)

As you’ve been learning, **SQL functions** are tools built into SQL to facilitate performing calculations. For example, you could use the **AVG()** function to calculate the average salary of employees in a table so management knows what to budget for next year. Another example might be using the **COUNT()** function to count the number of orders in a table to track daily order inventory.

A **subquery**, **also called an inner or nested query**, is a SQL query that is nested inside a larger query. Going back to the previous example, you could add a subquery to your average calculation to identify the names of employees who earn more or less than the average salary to include that information in performance reviews. Subqueries allow more complex questions to be answered in a single query, making data retrieval more efficient. In this reading, you will learn about SQL functions and how they might be used with subqueries.

## **How do SQL functions function?**

**SQL functions help make data aggregation possible.**

As a refresher, **data aggregation is the process of gathering data from multiple sources in order to combine it into a single, summarized collection**. Take a moment to review some of these functions to better understand how to run these queries:

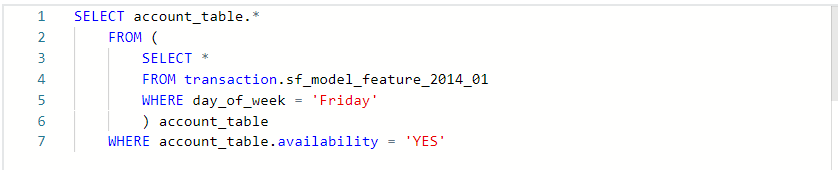
* **HAVING:** The **HAVING** clause filters the results of a SQL query based on conditions applied after the grouping. Check out [W3School’s HAVING overview](http://www-db.deis.unibo.it/courses/TW/DOCS/w3schools/sql/sql_having.asp.html) for a tutorial on this clause
* **CASE: CASE** provides conditional logic in SQL queries, similar to an 'if-else' structure in programming languages. The [W3School’s CASE overview](https://www.w3schools.com/sql/sql_case.asp) explores the use of the **CASE** statement and how it works.
* **IF:** IFperforms a simple conditional test and returns a value depending on the outcome. Review [W3School’s IF overview](https://www.w3schools.com/sql/func_mysql_if.asp) for a tutorial of the **IF** function and examples that you can practice with.
* **COUNT: COUNT** performs a simple conditional test and returns a value depending on the outcome. Though it seems simple, the **COUNT** function is just as important as all the rest. The [W3School’s COUNT overview](http://www-db.deis.unibo.it/courses/TW/DOCS/w3schools/sql/sql_func_count.asp.html) provides a tutorial and examples.

## **Subqueries**

**Subqueries can make projects easier and more efficient by allowing complex operations to be performed in a single query, reducing the need for multiple trips to the database.** Subqueries also make your code more readable and maintainable.

Take the employee salary example mentioned before.:The original query was used to find the average employee salary. By adding a subquery, you can learn this plus identify employees who earn more than the average—all in a single query.

Usually, you will find subqueries nested in the **SELECT**, **FROM**, and/or **WHERE** clauses. There is no general syntax for subqueries, but the syntax for a basic subquery follows a similar pattern:



Basically, there’s another **SELECT** clause inside the first **SELECT** clause. The second **SELECT** clause marks the start of the subquery in this statement. There are many different ways in which you can use subqueries, **but there are a few rules to follow:**

* Subqueries must be enclosed within parentheses.
* A subquery can have one or more columns specified in the **SELECT** clause.
* Subqueries that return more than one row can only be used with multiple value operators, such as the **IN** operator which allows you to specify multiple values in a **WHERE** clause.
* A subquery can’t be nested in a **SET** command. The **SET** command is used with **UPDATE** to specify which columns (and values) are to be updated in a table.

### 

### **Additional resources**

The following resources offer more guidance into subqueries and their usage:

* [**SQL subqueries**](https://www.w3resource.com/sql/subqueries/understanding-sql-subqueries.php):This detailed introduction includes the definition of a subquery, its purpose in SQL, when and how to use it, and what the results will be.
* [**Writing subqueries in SQL**](https://mode.com/sql-tutorial/sql-sub-queries/): Explore the basics of subqueries in this interactive tutorial, including examples and practice problems that you can work through.

As you continue to learn more about using SQL, functions, and subqueries, you will realize how much time you can truly save when memorizing these tips and tricks.

### [REVIEW SUBQUERIES](https://www.coursera.org/learn/analyze-data/ungradedWidget/OQqEx/review-subqueries)

### [JUSTIN: WHERE DATA ANALYSIS TAKES YOU](https://www.coursera.org/learn/analyze-data/lecture/t5PLC/justin-where-data-analysis-takes-you)

Hi, I'm Justin. I work here at Google in the Google Cloud space. I lead a small team of data analysts who answer business problems for our executive team. The first thing I would tell you about my journey to analytics was, it was not direct. I came to Google three years ago and I have been doing data analytics and I've really been enjoying that role. It's tied together through a line of excitement about data and answering questions that have an impact.

**Your career path is not always straightforward**.

Maybe data analytics won't be my final destination. But what I would say is just keep changing little by little, figure out what's exciting about your role right now, in my case, it was, I loved avoiding politics and coalition-building and really just bringing better facts and better insights to really motivate decisions. So figure out what you like about your current role, your current job, and then, figure out what different role you could take that would build on that, but maybe get you more of what you like.

**Be curious. The number one skill is really just asking why and then going and trying to answer that question**.

It will lead you down whether it's Wikipedia to understand this model there is someone usually saying, "Why did they use that model?" You go and look up that model and sort of follow that thread. Or there's so many great resources for different languages, if you want to understand SQL. There's so many great tools, but I guess the number one thing is just sort of follow your curiosity.

When I'm reviewing resumes, **the first thing obviously I'm looking for is those core skills**, the ability to analyze data, demonstrate it, experience with some of the tooling we use. But **I'm also looking for real passion in answering questions.**

**Example** is where someone's really dug in and tried to understand the why and they just kept asking,

"Why is this happening, why is this happening?" and really dig in.

### [HANDS-ON ACTIVITY: USE SUBQUERIES](https://www.coursera.org/learn/analyze-data/quiz/eXU5l/hands-on-activity-use-subqueries)



## **Activity Overview**

****

As a data analyst, you will sometimes want to analyze a small subset of data that is contained within a much larger dataset. This is when subqueries can be really useful. As you’ve been learning, subqueries are queries that are nested inside of another query. Some queries can have many subqueries, while others may have only one. It’s important to know how subqueries function and to understand the different components you can use to create them.

In this activity, you will work with queries and subqueries to examine a public health dataset. You will create subqueries to discover which industries receive an unusual number of complaints and, more importantly, which industries are connected to serious health issues. This will allow you to allocate your resources more effectively to safeguard public health.

### 

### **Scenario**

Review the following scenario. Then complete the step-by-step instructions.

In this scenario, you are a junior data analyst for a multinational food and beverage manufacturer. You and your team are responsible for maintaining the safety of a wide array of food products. Because of the overwhelming number of products on the market, you have been asked to prioritize which products need to be reviewed by your stakeholders.

While it's useful to know which food industries receive the most complaints, the more critical aspect to consider is identifying the complaints that lead to severe health consequences, such as hospital visits.

To complete this task, you'll analyze food event reports for targeted health interventions.

### 

### **Step-By-Step Instructions**

### 

### **Step 1: Access the public dataset**

The first step is to access the public dataset. For this activity, you will use a BigQuery public dataset titled fda\_food, which contains details about food recalls, consumer reactions, product descriptions, and product distribution and quantities.

1. Log in to BigQuery and open the [BigQuery Console](https://console.cloud.google.com/bigquery).

Note: BigQuery frequently updates its user interface. The latest changes may not be reflected in the screenshots presented in this activity, but the principles remain the same. Adapting to changes in software updates is an essential skill for data analysts, and it’s helpful for you to practice troubleshooting. You can also reach out to your community of learners on the discussion forum for help.

2. Before you can load the fda\_food dataset, you need to have bigquery-public-data pinned in the Explorer menu of your SQL Workspace. Follow these steps to pin the dataset:

a. Enter public in the Explorer menu search box, and press the Enter/Return on your keyboard.

b. Select SEARCH ALL PROJECTS.

c. Scroll to find bigquery-public-data and select the star to pin it.

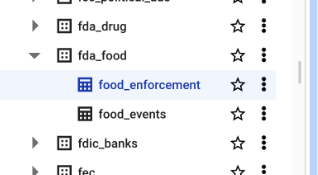
3. Add the FDA food dataset. Follow these steps:

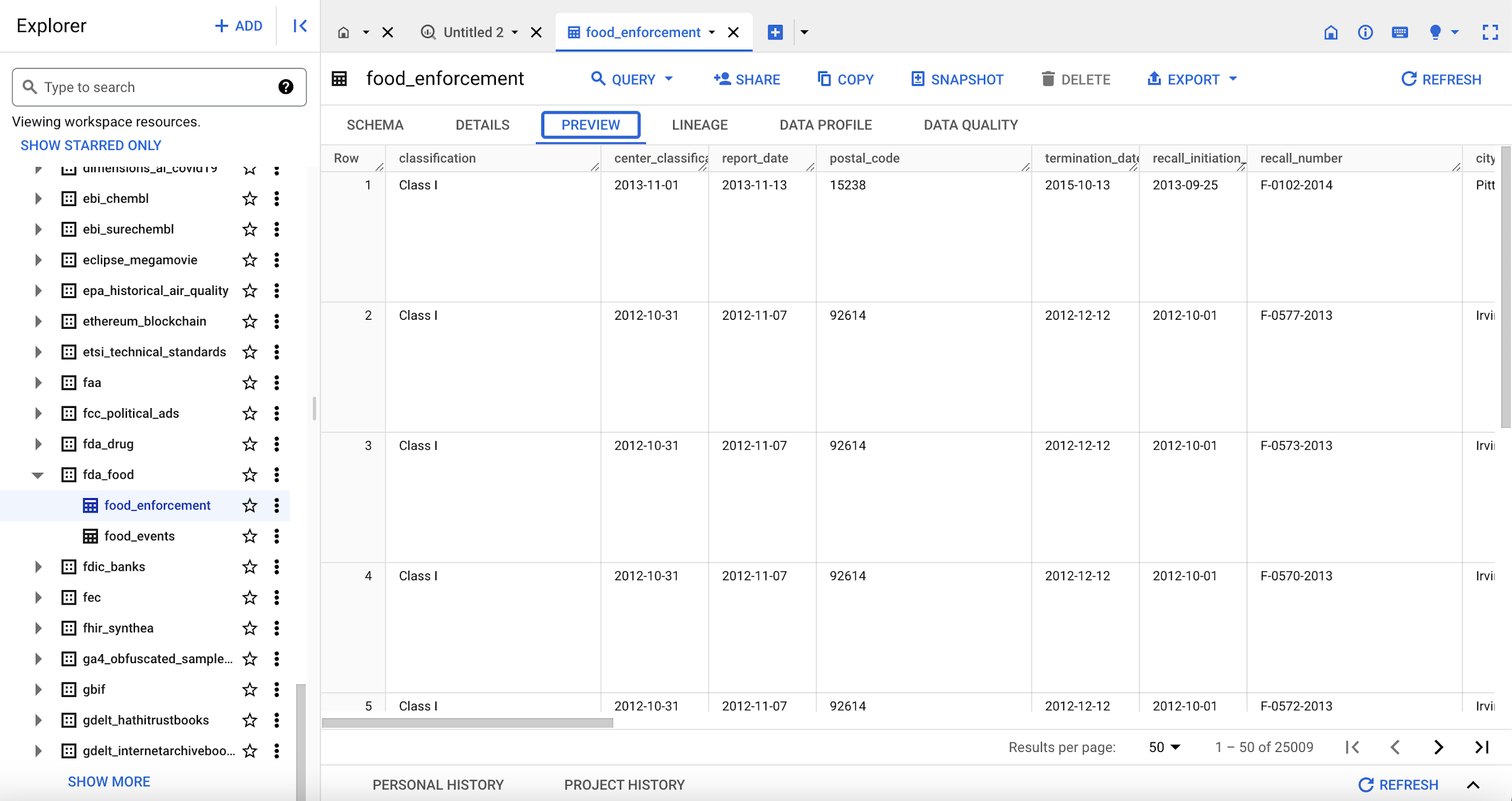
a. Select the arrow next to the bigquery-public-data to expand its contents. T

b. Scroll to find fda\_food. You may have to select SHOW MORE to view the dataset listed.

c. Expand fda\_food and select food\_enforcement.

d.Select the Preview tab in the viewer to examine the table data.





4. Next, select the food\_events table from the fda\_food dataset, then select Preview to examine the table data.

Once you have taken some time to explore the tables within the fda\_food dataset, you are ready to begin working with subqueries!

**Note:** Many of the public databases on BigQuery are living records and, as such, are periodically updated with new data. Throughout this course (and others in this certificate program), if your results differ from those you encounter in videos or screenshots, there's a good chance it is due to a data refresh.

### 

### **Step 2: Gather an initial overview**

To get an initial overview, run a SQL query to identify which food industries have the highest number of complaints. This will provide an initial dataset to work from. To find the top industries for recalls, perform the following steps:

1. Create a new query.

2. Enter the following in the query editor:

SELECT

products\_industry\_name,

COUNT(report\_number) AS count\_reports

--SELECT is used to identify the product industries by name. COUNT will count the number of reports and label them as count\_reports.

FROM bigquery-public-data.fda\_food.food\_events

GROUP BY products\_industry\_name

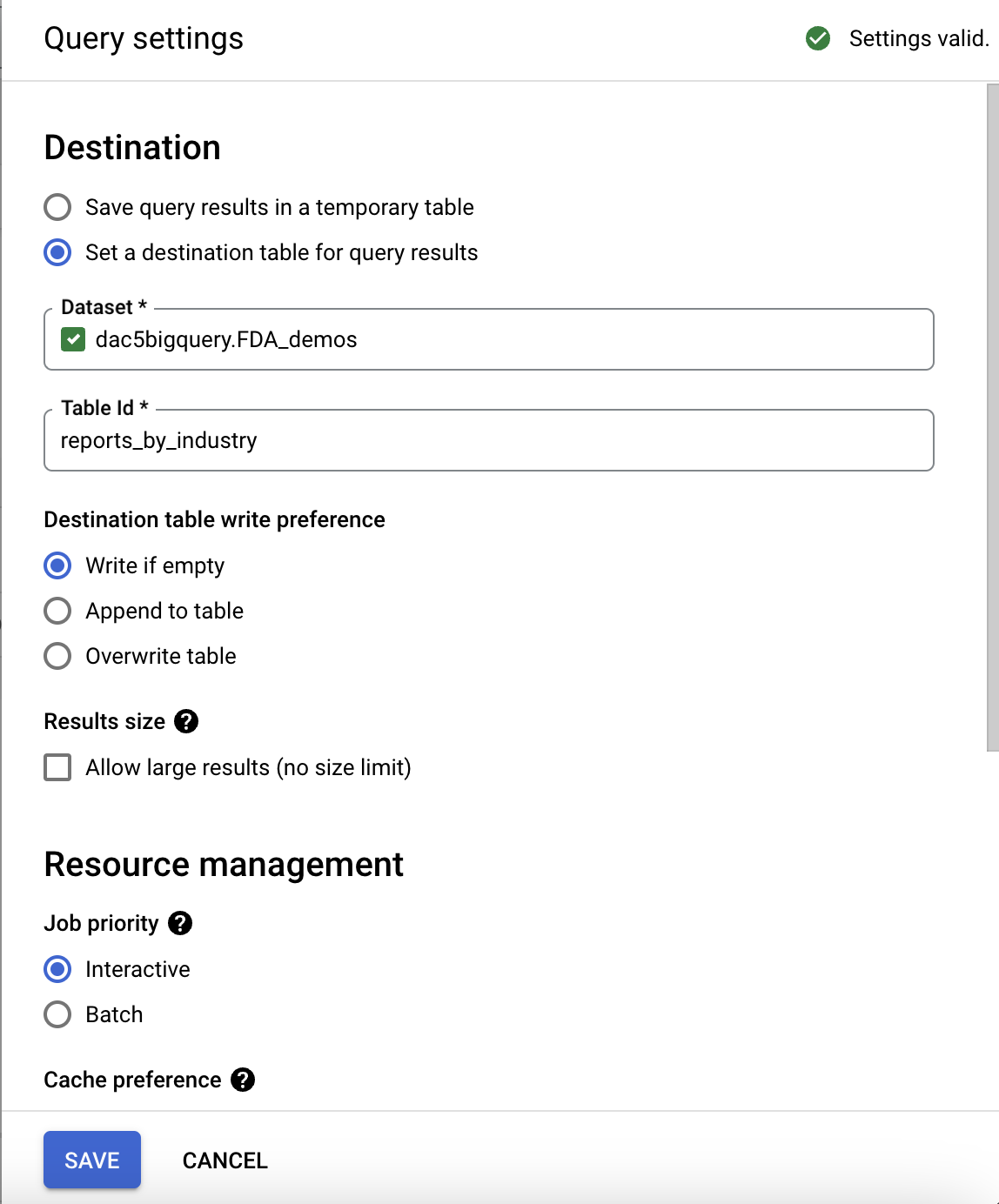
ORDER BY count\_reports DESC

LIMIT 10;

--The query will group the product industries by name so the report results will fall under each industry title. ORDER BY and DESC will tell the query to order the output by count\_reports in descending order so, the industry with the most amount of reports will appear at the top of the output table. LIMIT will limit the results to ten industries with the highest numbers of reports.

**Note:** Depending on the type of web browser you are using, you may have issues conducting a direct copy + paste of the above syntax directly from this assignment to your BigQuery workspace. If you are experiencing this issue, open two separate windows on your screen to manually enter the above information into the BigQuery workspace area.

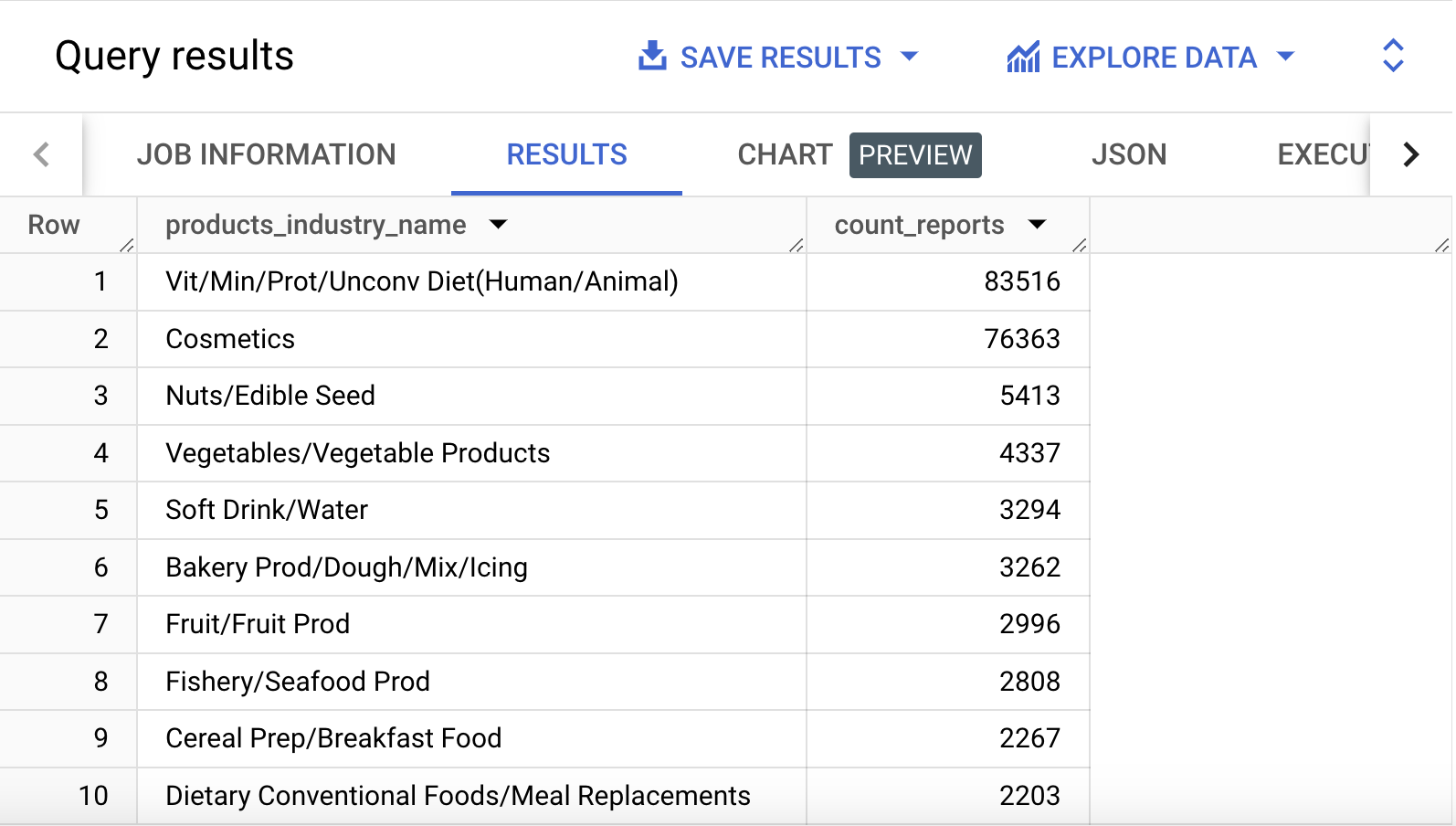
3. You have the option to save your query as a table. To do this, select the MORE menu from the Query Editor and open the Query Settings menu. In the Query Settings menu, select Set a destination table for query results. Set the dataset option to demos and name the table reports\_by\_industry.



4. Select RUN. The query will save as a new table in your demos dataset.

The Query results window will populate with a table containing two columns and 10 rows. The first column is **products\_industry\_name**, and the second is **count\_reports**. The 10 different industries are listed in descending order in the rows.

Take some time to check the different industries and the differences in report numbers. You can modify your search to examine the different symptoms that consumers experienced (**reactions**) based on products (**products\_brand\_name**) as well as the **outcomes**, or consequences, from each food event.



### 

### **Step 3: Determine the number of hospitalizations**

With a list of the industries receiving the most complaints, your next step is to find out which of these complaints led to hospitalizations.   
  
**Filter the initial list accordingly and focus solely on complaints that resulted in hospital visits.**

**To do this:**

**1. Create a new query.**

**2. Enter the following into the editor:**

SELECT

products\_industry\_name,

COUNT(report\_number) AS count\_hospitalizations

FROM

bigquery-public-data.fda\_food.food\_events

WHERE products\_industry\_name IN

(SELECT

products\_industry\_name

FROM

bigquery-public-data.fda\_food.food\_events

GROUP BY products\_industry\_name

ORDER BY COUNT(report\_number) DESC LIMIT 10)

AND outcomes LIKE '%Hospitalization%'

--The AND operator displays a record if all the conditions are TRUE.

--The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.

GROUP BY products\_industry\_name

ORDER BY count\_hospitalizations DESC;

The outer query used here is just like the previous query. **SELECT** calls the product industry names, and **COUNT** counts the number of reports and labels them as count\_hospitalizations. The query is told to pull this data from the **food\_events** table.

The subquery begins with a **WHERE** clause and an **IN** operator—these will allow the subquery to specify multiple values. The second **SELECT** statement tells the subquery to pull the product industry names **FROM** the **food\_events** table in the **FDA\_food** dataset. Results are then grouped by product industry name and ordered in descending order. **LIMIT** tells the query to limit the number of results to 10. **AND** and **LIKE** are used to indicate if hospitalizations are present within the **outcomes** field.

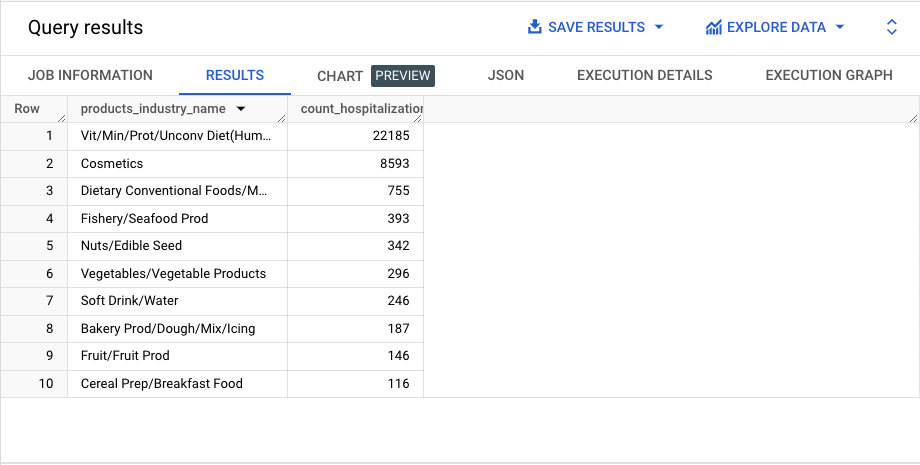
The outer query is then closed by grouping results by **product\_industry\_name** and ordered in descending order by the number of hospitalizations recorded.

3. You can also save this query as a table. Before running the query, select the MORE menu from the Query Editor and open the Query Settings menu. In the Query Settings menu, select Set a destination table for query results. Set the dataset option to

4. Select RUN. The query will save as a new table in your demos dataset.

The Query results window will populate with a table containing two columns and ten rows. The first column is **products\_industry\_name**, and the second is **count\_hospitalizations**. The 10 different industries are listed in descending order in the rows.

Take some time to review the different industries and the differences in reported hospitalizations. You can modify your search to examine the different symptoms that consumers experienced (**reactions**) based on products (**products\_brand\_name**); the dates in which the food events occurred (**date\_started**); and the **outcomes**, or consequences, from each food event. These modifications will allow you to find and analyze any links among fields within the datasets for your research.



Great work! Through this process, you discovered which industries receive an unusual number of complaints and, more importantly, which are connected to serious health issues. This knowledge allows you to allocate your resources more effectively to safeguard public health. By completing this activity, you have created subqueries using **SELECT** statements with **FROM, GROUP BY, WHERE, IN, AS, COUNT**, and **ORDER BY** clauses, as well as **AND** and **LIKE** operators to search and analyze the query results relating to reports of food events by industry.

### [TEST YOUR KNOWLEDGE ON WORKING WITH SUBQUERIES](https://www.coursera.org/learn/analyze-data/quiz/NzpAZ/test-your-knowledge-on-working-with-subqueries)

## MODULE 3 CHALLENGE

### [GLOSSARY TERMS FROM MODULE 3](https://www.coursera.org/learn/analyze-data/supplement/eXxLS/glossary-terms-from-module-3)

## **Terms and definitions for Course 5, Module 3**

**Absolute reference:** A reference within a function that is locked so that rows and columns won’t change if the function is copied

**Aggregation:** The process of collecting or gathering many separate pieces into a whole

**Aliasing:** Temporarily naming a table or column in a query to make it easier to read and write

**COUNT DISTINCT:** A SQL function that only returns the distinct values in a specified range

**Data aggregation:** The process of gathering data from multiple sources and combining it into a single, summarized collection

**INNER JOIN :** A SQL function that returns records with matching values in both tables

**JOIN:** A SQL function that is used to combine rows from two or more tables based on a related column

**LEFT JOIN:** A SQL function that will return all the records from the left table and only the matching records from the right table

**LIMIT:** A SQL clause that specifies the maximum number of records returned in a query

**MATCH:** A spreadsheet function used to locate the position of a specific lookup value

**OUTER JOIN:** A SQL function that combines RIGHT and LEFT JOIN to return all matching records in both tables

**RIGHT JOIN:** A SQL function that will return all records from the right table and only the matching records from the left.

**Subquery:** A SQL query that is nested inside a larger query

**VALUE:** A spreadsheet function that converts a text string that represents a number to a numeric value