Lesson 26

SQL Introduction

There are many uses for SQL in software development. We will focus on the parts of SQL used in data analysis.

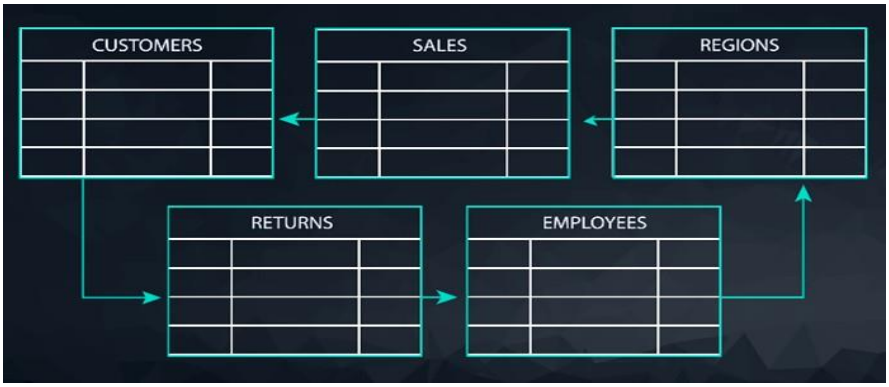
At the moment, companies are gathering a lot of data and expect staff in all fields to use it systematically in decision making. There are many ways to do this.

The Parch & Posey Database

For examples of this course, we will use data from Parch & Posey, a paper-selling company. It has 50 sales representatives spread across the United States in 4 regions. It sells 3 types of paper, plain, poster and glossy. And their clients are the largest of the 100 richest companies who attract them by advertising on Google, Facebook and Twitter.

By using SQL, we will be able to help Parch & Bosey answer the deceptive questions such as: Which of their product lines is worst performing ? Which of their they should make a greater marketing investment in?

There is one way to store data and is on a spreadsheet. Sometimes, we need a lot of spreadsheets to store data from different sources. We can represent relationships between spreadsheets by using Entity relationship diagram **"ERD"**.

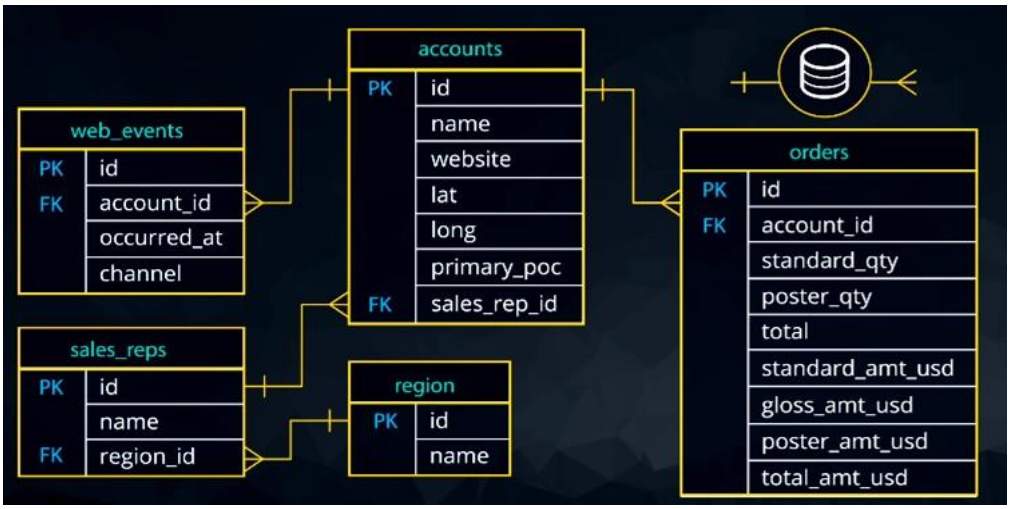


Entity Relationship Diagrams

An **entity relationship diagram** (ERD) is a common way to view data in a database. Below is the ERD for the database we will use from Parch & Posey. **These diagrams help you visualize the data you are analyzing including:**

1. The names of the tables.
2. The columns in each table.
3. The way the tables work together.

This is what an ERD looks like.



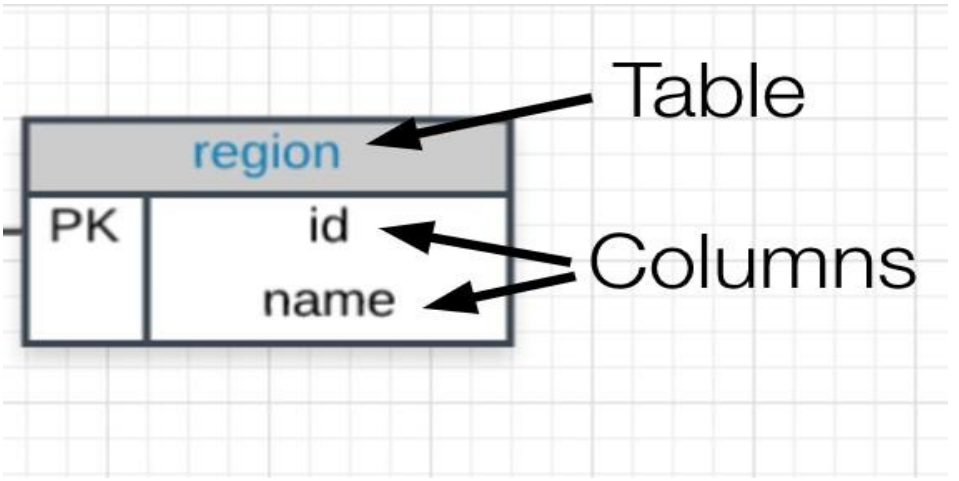
Each spreadsheet is represented on a table. At the top, you will see the name of the table, and then below, each column name is listed.

What to Notice

In the Parch & Posey database there are five tables (essentially 5 spreadsheets):

1. **web_events**
2. **accounts**
3. **orders**
4. **sales_reps**
5. **region**

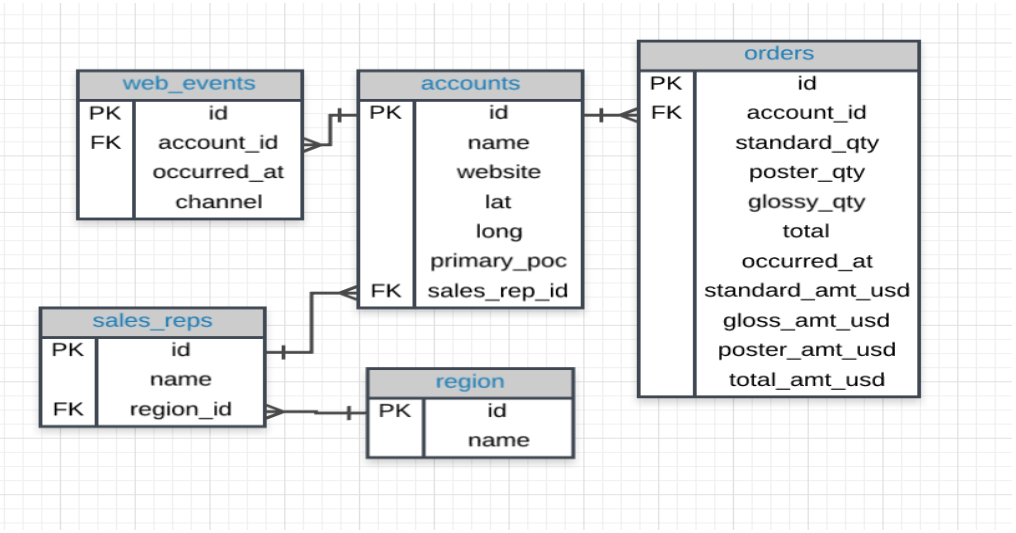
You can think of each of these tables as an individual spreadsheet. Then the columns in each spreadsheet are listed below the table name. For example, the **region** table has two columns: id and name. Alternatively the **web_events** table has four columns.



The "crow's foot" that connects the tables together shows us how the columns in one table relate to the columns in another table. In this first lesson, you will be learning the basics of how to work with SQL to interact with a single table. In the next lesson, you will learn more about why these connections are so important for working with SQL and relational databases.

- SQL is a language used to interact with a database.it can query one table, or it can query across multiple tables.
- Understanding these relationships will help you gain insights from your data faster than with other software.

Quiz: ERD Fundamentals



QUIZ QUESTION

Use the above image and your new knowledge of ERDs to match each Term to the appropriate Definition.

Submit to check your answer choices!

DEFINITION	TERM
A column name in the Parch & Posey database.	primary_poc
A table name in the Parch & Posey database.	web_events
A collection of tables that share connected data stored in a computer.	Database
A diagram that shows how data is structured in a database.	ERD
A language that allows us to access data stored in a database.	SQL

Map of SQL Content

Introduction

Throughout the next three lessons, you will be learning how to write **Structured Query Language (SQL)** to interact with a database here in the classroom. You will not need to download any software, and you will still be able to test your skills!

SQL is an extremely in demand skill. **Tons of jobs use SQL**, and in the next lessons you will be learning how to utilize SQL to analyze data and answer business questions.

Project

The skills you learn in the classroom are directly extendable to writing **SQL** in other environments outside this classroom. For the project at the end of these lessons, you will download a program that will allow you to write code on your local machine. You will then analyze and answer business questions using data associated with a music store by querying their database.

Lesson Outline

There are three lessons in this Nanodegree aimed at helping you understand how to write SQL queries. If you choose to take the **Business Analyst Nanodegree** or the **Data Analyst Nanodegree** programs, these three lessons will also be a part of these programs. However, there is also an additional lesson on Advanced SQL also taught by Derek!

The three lessons in this course aim at the following components of SQL:

- **SQL Basics** - Here you will get your first taste at how SQL works, and learn the basics of the SQL language. You will learn how to write code to interact with tables similar to the ones we analyzed in Excel earlier. Specifically, you will learn a little about databases, the basic syntax of SQL, and you will write your first queries!

- **SQL Joins** - In this lesson, you will learn the real power of SQL. You will learn about Entity Relationship Diagrams (ERDs), and how to join multiple tables together from a relational database. The power to join tables is what really moved companies to adopt this approach to holding data.

- **SQL Aggregations** - In this final lesson, you will learn some more advanced features of SQL. You will gain the ability to summarize data from multiple tables in a database.

At the end of these three lessons, you will be ready to tackle the project. The project aims to assure you have mastered these three topics, but you will also see some of the more advanced queries that were not covered in this course. These are just meant to introduce you to the advanced material, but don't feel discouraged if you didn't get these - they were beyond the scope of the class, and they are not required to pass the project!

Why SQL

Introduction

Before we dive into writing SQL queries, let's take a look at what makes SQL and the databases that utilize SQL so popular.

I think it is an important distinction to say that SQL is a **language**. Hence, the last word of SQ**L** being **language**. SQL is used all over the place beyond the databases we will utilize in this class. With that being said, SQL is most popular for its interaction with databases. For this class, you can think of a **database** as a bunch of excel spreadsheets all sitting in one place. Not all databases are a bunch of excel spreadsheets sitting in one place, but it is a reasonable idea for this class.

Why Do Data Analysts Use SQL?

There are some major advantages to using **traditional relational databases,** which we interact with using SQL. **The five most apparent are:**

- SQL is easy to understand.
- Traditional databases allow us to access data directly.
- Traditional databases allow us to audit and replicate our data.
- SQL is a great tool for analyzing multiple tables at once.
- SQL allows you to analyze more complex questions than dashboard tools like Google Analytics.

You will experience these advantages first hand, as we learn to write SQL to interact with data.

SQL vs. No.SQL

You may have heard of NoSQL, which stands for not only SQL. Databases using No.SQL allow for you to write code that interacts with the data a bit differently than what we will do in this course. These No.SQL environments tend to be particularly popular for web based data, but less popular for data that lives in spreadsheets the way we have been analyzing data up to this point. One of the most popular No.SQL languages is called **MongoDB**.

Why Do Businesses Choose SQL?

1. **Data integrity is ensured** - only the data you want entered is entered, and only certain users are able to enter data into the database.
2. **Data can be accessed quickly** - SQL allows you to obtain results very quickly from the data stored in a database. Code can be optimized to quickly pull results.
3. **Data is easily shared** - multiple individuals can access data stored in a database, and the data is the same for all users allowing for consistent results for anyone with access to your database.

Databases have a number of features that make them great at accessing data. More importantly, databases help to verify the integrity of data, and can ensure consistency of data entered.

For example, if we record the number of children in a person, the database will only be able to impose integers, because we can not have half a child.

In fact, they are also databases, and they can run very quickly across very large sets of data that are ideal for additional speed.

The databases represent common entities. Many people can access a database at the same time. More importantly, all these people will deal with the same data.

How Databases Store Data

A few key points about data stored in SQL databases:

1. **Data in databases is stored in tables that can be thought of just like Excel spreadsheets.**

   For the most part, you can think of a database as a bunch of Excel spreadsheets. Each spreadsheet has rows and columns. Where each row holds data on a transaction, a person, a company, etc., while each column holds data pertaining to a particular aspect of one of the rows you care about like a name, location, a unique id, etc.

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

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

Text + Quiz: Types of Databases

Types of Databases

SQL Databases

There are many different types of SQL databases designed for different purposes. In this course we will use **Postgres** within the classroom, which is a popular open-source database with a very complete library of analytical functions.
Some of the most popular databases include:

1. MySQL
2. Access
3. Oracle
4. Microsoft SQL Server
5. Postgres

You can also write SQL within other programming frameworks like Python, Scala, and HaDoop.

Small Differences

Each of these SQL databases may have subtle differences in syntax and available functions -- for example, MySQL doesn't have some of the functions for modifying dates as Postgres. **Most** of what you see with Postgres will be directly applicable to using SQL in other frameworks and database environments. For the differences that do exist, you should check the documentation. Most SQL environments have great documentation online that you can easily access with a quick Google search.
The article here compares three of the most common types of SQL: SQLite, Postgre SQL, and My SQL.
Though you will use Postgre SQL in the classroom, you will utilize SQLite for the project. Again, once you have learned how to write SQL in one environment, the skills are mostly transferable.
So with that, let's jump in!

QUIZ QUESTION

Check all of the below that are true about your learning experience here in the classroom.

- [x] The code you write in the classroom is exactly what you would write to analyze data in Postgre SQL.
- [ ] The code you write in the classroom is exactly what you would write to analyze data in MySQL.
- [x] The code you write in the classroom will allow you to easily pick up any SQL programming including Microsoft SQL Server, Oracle, or SQLite.
- [ ] When you leave the classroom, you will have to re-learn how to use SQL, as it works differently outside the classroom.

Types of Statements

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

1. **CREATE TABLE** is a statement that creates a new table in a database.
2. **DROP TABLE** is a statement that removes a table in a database.
3. **SELECT** allows you to read data and display it. This is called a **query**.

The **SELECT** statement is the common statement used by analysts, and you will be learning all about them throughout this course!

Quiz: Statements

QUESTION 1 OF 2

In SQL, you can think of a statement as (select all that apply):

- [x] A piece of correctly written SQL code.
- [x] A way to manipulate data stored in a database.
- [x] A sentence.
- [x] A way to read data stored in a database.

QUESTION 2 OF 2

In the following lessons we will focus on which type of query?

- [ ] DROP
- [ ] CREATE
- [ ] QUERY
- [ ] SELECT

SELECT & FROM

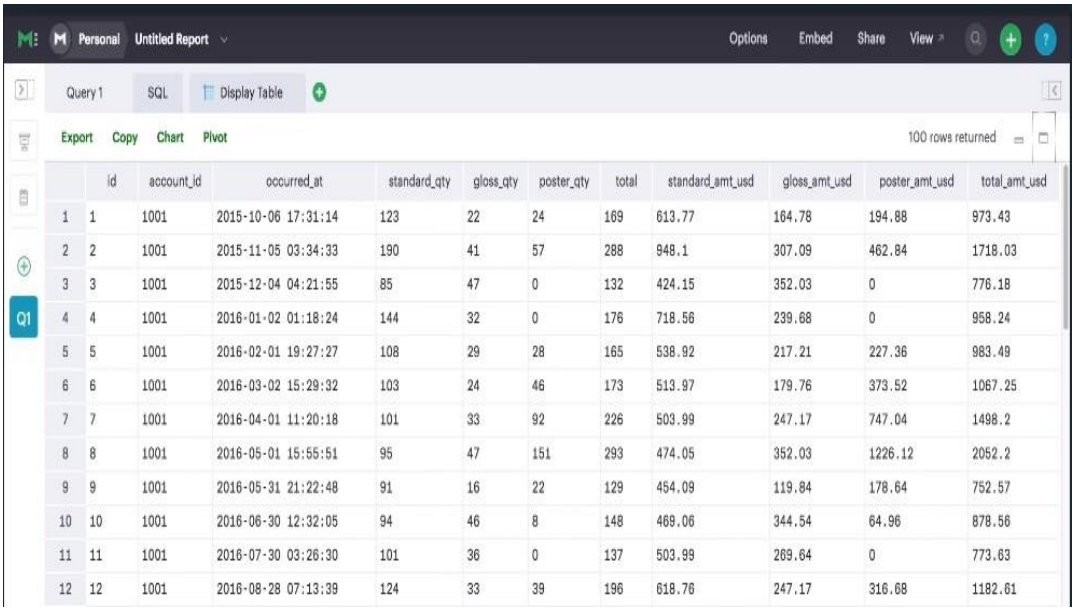
Here you were introduced to two statements that will be used in every query you write

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

2. **FROM** is where you tell the query what table you are querying from. Notice the columns need to exist in this table.

You will use these two statements in every query in this course, but you will be learning a few additional statements and operators that can be used along with them to ask more advanced questions of your data.

For example, **Parch & Posey** recorded their requests in a table called the Orders table. This table contains all the columns we see here , each corresponding to an attribute, or a request.



	id	account_id	occurred_at	standard_qty	gloss_qty	poster_qty	total	standard_amt_usd	gloss_amt_usd	poster_amt_usd	total_amt_usd
1	1	1001	2015-10-06 17:31:14	123	22	24	169	613.77	164.78	194.88	973.43
2	2	1001	2015-11-05 03:34:33	190	41	57	288	948.1	307.09	462.84	1718.03
3	3	1001	2015-12-04 04:21:55	85	47	0	132	424.15	352.03	0	776.18
4	4	1001	2016-01-02 01:18:24	144	32	0	176	718.56	239.68	0	958.24
5	5	1001	2016-02-01 19:27:27	108	29	28	165	538.92	217.21	227.36	983.49
6	6	1001	2016-03-02 15:29:32	103	24	46	173	513.97	179.76	373.52	1067.25
7	7	1001	2016-04-01 11:20:18	101	33	92	226	503.99	247.17	747.04	1498.2
8	8	1001	2016-05-01 15:55:51	95	47	151	293	474.05	352.03	1226.12	2052.2
9	9	1001	2016-05-31 21:22:48	91	16	22	129	454.09	119.84	178.64	752.57
10	10	1001	2016-06-30 12:32:05	94	46	8	148	469.06	344.54	64.96	878.56
11	11	1001	2016-07-30 03:26:30	101	36	0	137	503.99	269.64	0	773.63
12	12	1001	2016-08-28 07:13:39	124	33	39	196	618.76	247.17	316.68	1182.61

Each request has a unique identifier, so we can refer to it and a time stamp indicating when this request was made. It also contains the quantity sold, yield and output for each type of paper requested.

To create all the requests, we write the SELECT statement.

Consider the term "Select" when filling out a form to get the results we are trying to find.

The form contains a set of questions such as:

What data do you wont to pull from?

Which elements from the database do you wont to pull?

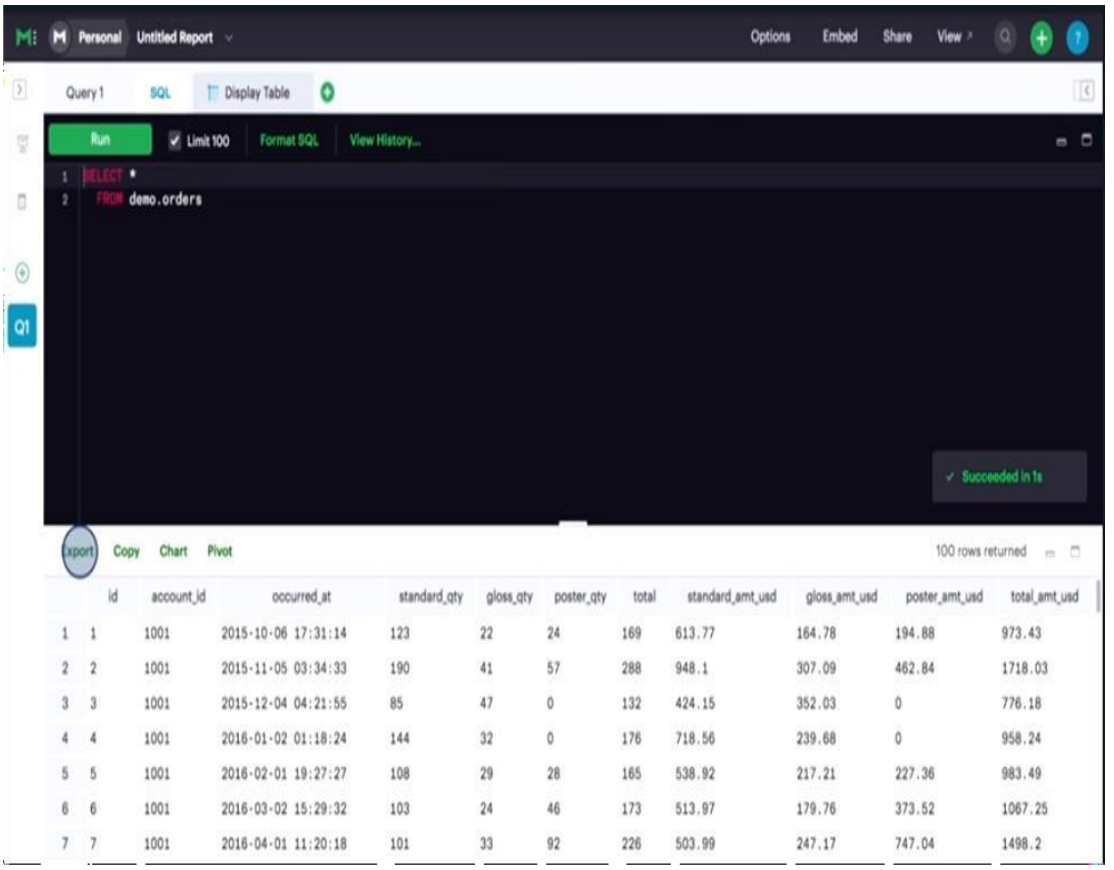
These questions are structured in the same order each time. Some of which are mandatory and the other optional.

The term "query" tells you what data to use from a table. The Select statement tells you which columns to read from that table.

Column names are separated by commas, with no comma after the last column name. We can also specify all column names using Select with the asterisk "star means all columns".

The terms Select and From are mandatory. They will be included in each query we write as part of this course. If we do not guarantee both, we will get an error message instead of the desired results.

When you click the play button, the database will execute this query and the results will be returned. Each row in the order table will show all available columns.



	id	account_id	occurred_at	standard_qty	gloss_qty	poster_qty	total	standard_amt_usd	gloss_amt_usd	poster_amt_usd	total_amt_usd
1	1	1001	2015-10-06 17:31:14	123	22	24	169	613.77	164.78	194.88	973.43
2	2	1001	2015-11-05 03:34:33	190	41	57	288	948.1	307.09	462.84	1718.03
3	3	1001	2015-12-04 04:21:55	85	47	0	132	424.15	352.03	0	776.18
4	4	1001	2016-01-02 01:18:24	144	32	0	176	718.56	239.68	0	958.24
5	5	1001	2016-02-01 19:27:27	108	29	28	165	538.92	217.21	227.36	983.49
6	6	1001	2016-03-02 15:29:32	103	24	46	173	513.97	179.76	373.52	1067.25
7	7	1001	2016-04-01 11:20:18	101	33	92	226	503.99	247.17	747.04	1498.2

Text + Quiz: Your First Query

First SQL Statement

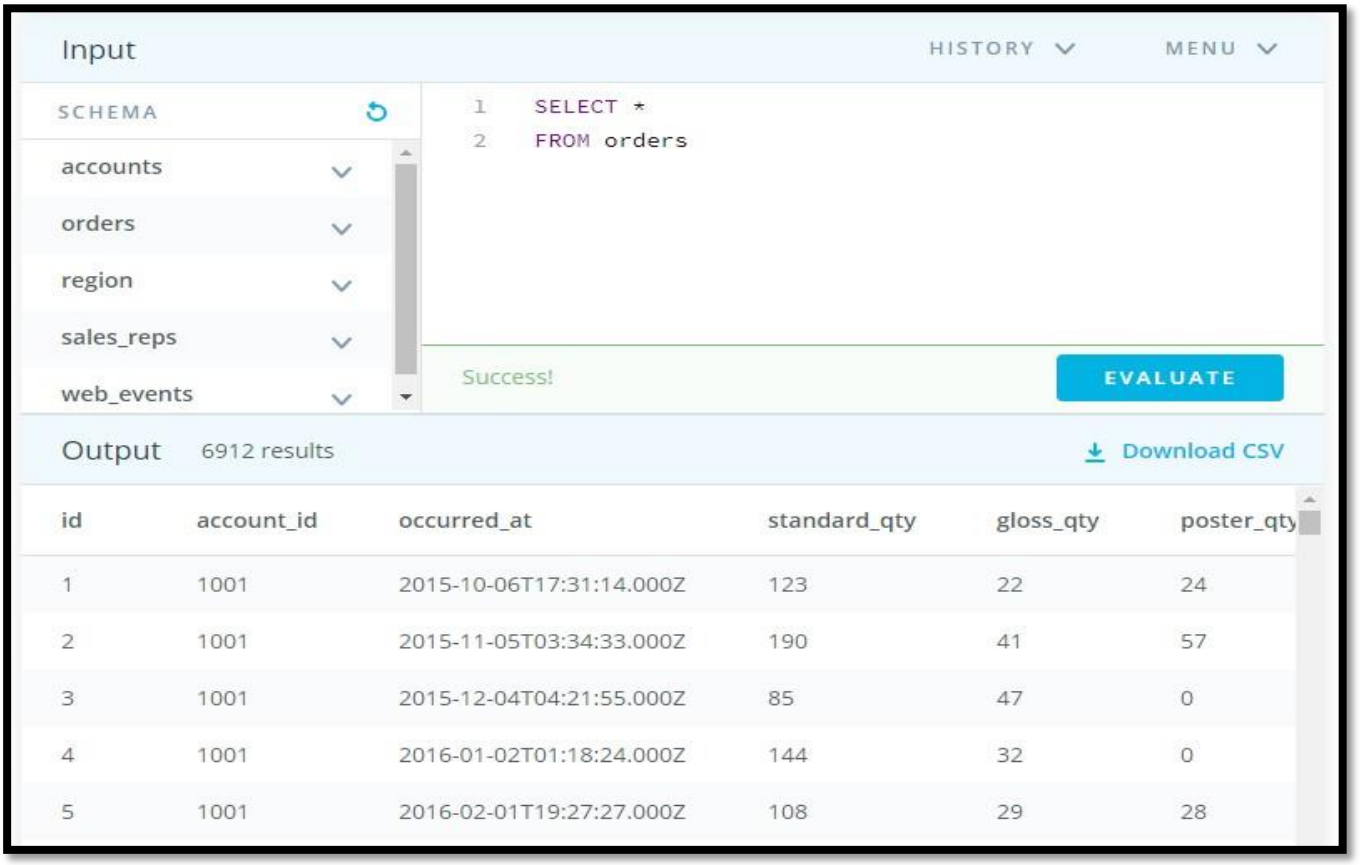
The tables from the Parch & Posey database are stored in the backend of the box below. You will notice on the left margin under **SCHEMA** is a list of tables that were shown in the ERD earlier. We'll write queries and run them to see the results using the table below in the same way you would in most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

The Udacity SQL Environment

In order to get started, try running the query you saw in the previous lesson ! In the environment at the bottom of the page, you have the ability to test your SQL code. In the left panel, you will find the tables that we saw earlier in the ERD. In the right panel you can write your SQL code, and we can click the **EVALUATE** button to run the query. This may take a moment to run.
The **HISTORY** menu will show your previously run queries. The **MENU** will allow you to remove the SCHEMA from the left panel, as well as reset the database.

```
SELECT *
FROM orders;
```

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



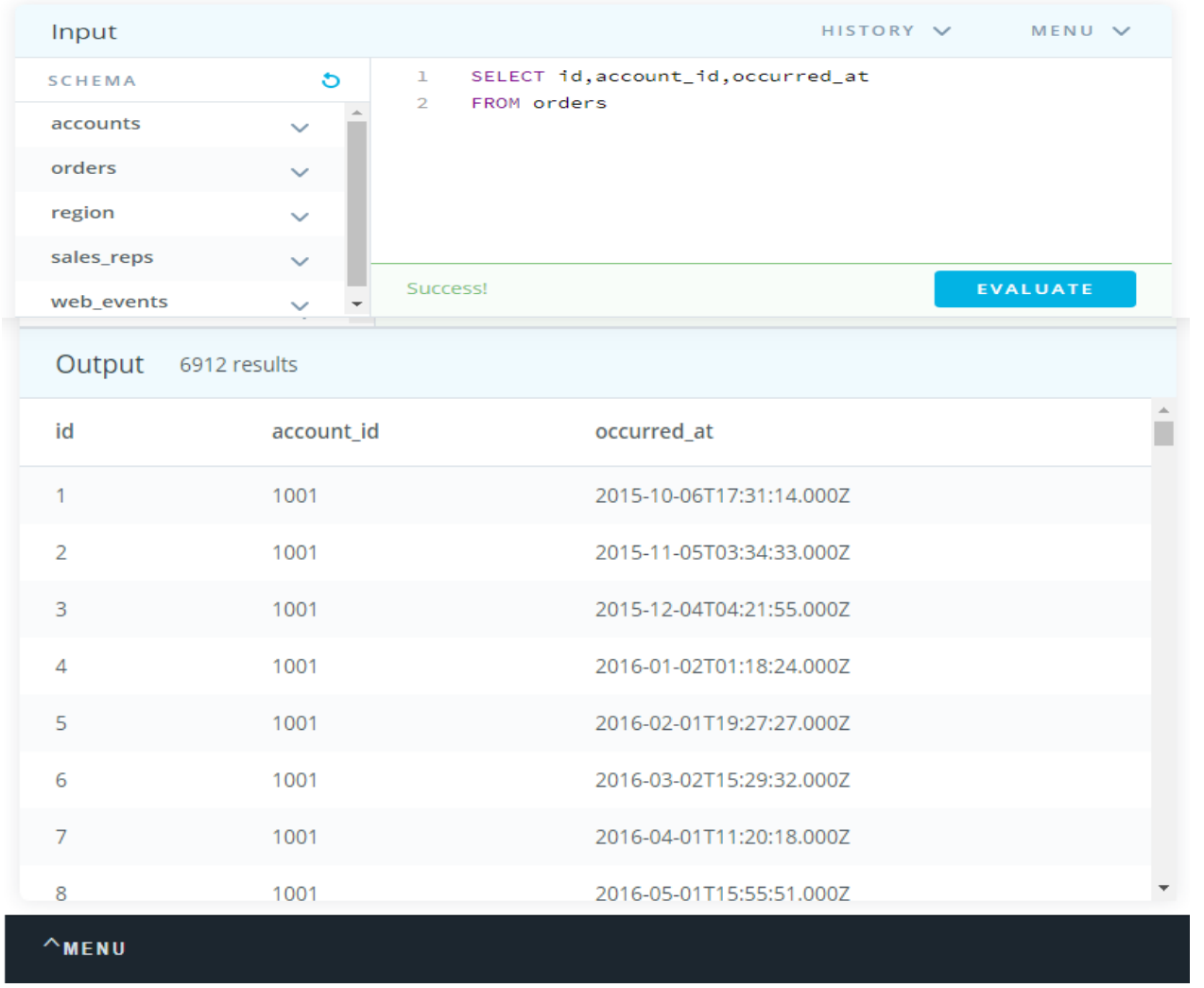
Every SQL Query

Every query will have at least a **SELECT** and **FROM** statement. The **SELECT** statement is where you put the **columns** for which you would like to show the data. The **FROM** statement is where you put the **tables** from which you would like to pull data.

Try writing your own query to select only the id, account_id, and occurred_at columns for all orders in the orders table.

Solution to Previous Concept

```
SELECT id, account_id, occurred_at
FROM orders
```



Output 6912 results

id	account_id	occurred_at
1	1001	2015-10-06T17:31:14.000Z
2	1001	2015-11-05T03:34:33.000Z
3	1001	2015-12-04T04:21:55.000Z
4	1001	2016-01-02T01:18:24.000Z
5	1001	2016-02-01T19:27:27.000Z
6	1001	2016-03-02T15:29:32.000Z
7	1001	2016-04-01T11:20:18.000Z
8	1001	2016-05-01T15:55:51.000Z

^MENU

Text: Formatting Best Practices

In case you want to test any of the ideas below, I have embedded the SQL environment at the bottom of this page.

Formatting Your Queries

Capitalization

You may have noticed that we have been capitalizing SELECT and FROM, while we leave table and column names lowercase. This is a common formatting convention. It is common practice to capitalize commands (SELECT, FROM), and keep everything else in your query lowercase. This makes queries easier to read, which will matter more as you write more complex queries. For now, it is just a good habit to start getting into.

Avoid Spaces in Table and Variable Names

It is common to use underscores and avoid spaces in column names. It is a bit annoying to work with spaces in SQL. In Postgres if you have spaces in column or table names, you need to refer to these columns/tables with double quotes around them (Ex: FROM "Table Name" as opposed to FROM table_name). In other environments, you might see this as square brackets instead (Ex: FROM [Table Name]).

Use White Space in Queries

SQL queries ignore spaces, so you can add as many spaces and blank lines between code as you want, and the queries are the same. This query

```
SELECT account_id FROM orders
```

is equivalent to this query:

```
SELECT account_id
FROM orders
```

and this query (but please don't ever write queries like this):

```
SELECT account_id
FROM     orders
```

SQL isn't Case Sensitive

If you have programmed in other languages, you might be familiar with programming languages that get very upset if you do not type the correct characters in terms of lower and uppercase. SQL is not case sensitive. The following query:

```
SELECT account_id
FROM orders
```

is the same as:

```
select  account_id
from  orders
```

which is also the same as:

```
SeLect  account_id
From orders
```

However, I would again urge you to follow the conventions outlined earlier in terms of fully capitalizing the commands, while leaving other pieces of your code in lowercase.

Semicolons

Depending on your SQL environment, your query may need a semicolon at the end to execute. Other environments are more flexible in terms of this being a "requirement." It is considered best practices to put a semicolon at the end of each statement, which also allows you to run multiple commands at once if your environment is able to show multiple results at once.

Best practice:

```
SELECT account_id
FROM  orders;
```

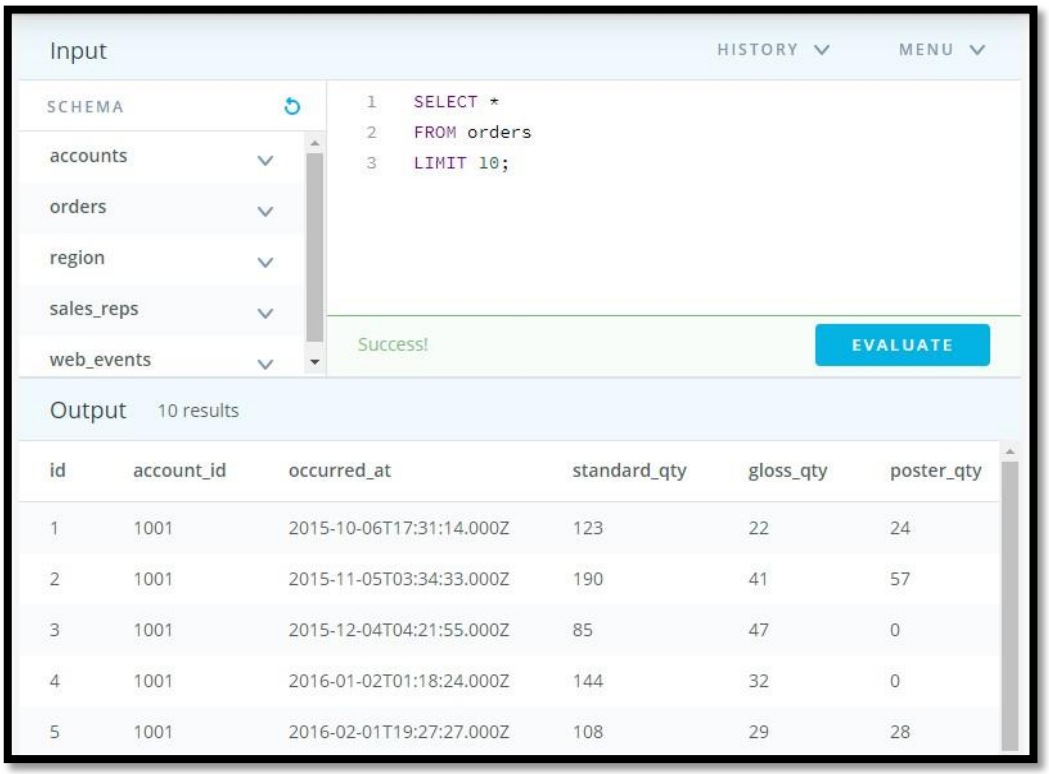
Since, our environment here doesn't require it, you will see solutions written without the semicolon:

```
SELECT account_id
FROM  orders
```

LIMIT

We have already seen the **SELECT** (to choose columns) and **FROM** (to choose tables) statements. The **LIMIT** statement is useful when you want to see just the first few rows of a table. This can be much faster for loading than if we load the entire dataset.

The **LIMIT** command is always the very last part of a query. **An example of showing just the first 10 rows of the orders table with all of the columns might look like the following:**



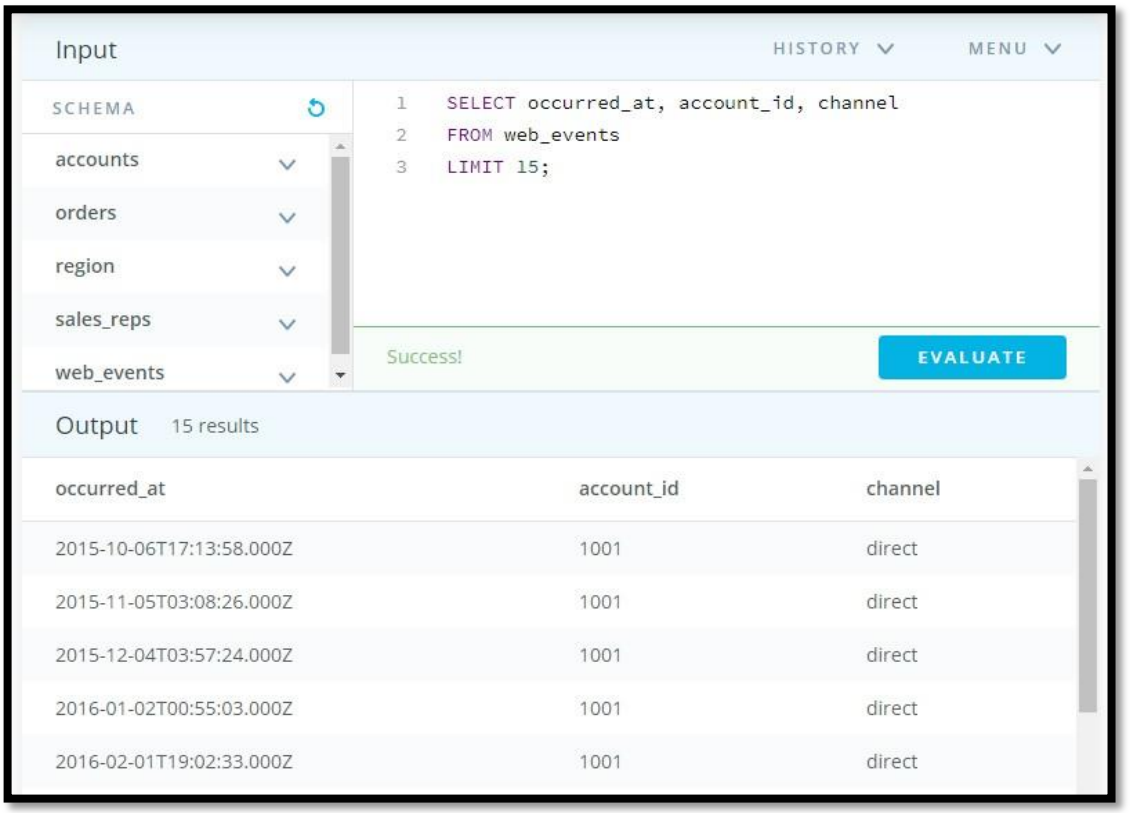
id	account_id	occurred_at	standard_qty	gloss_qty	poster_qty
1	1001	2015-10-06T17:31:14.000Z	123	22	24
2	1001	2015-11-05T03:34:33.000Z	190	41	57
3	1001	2015-12-04T04:21:55.000Z	85	47	0
4	1001	2016-01-02T01:18:24.000Z	144	32	0
5	1001	2016-02-01T19:27:27.000Z	108	29	28

- We could also change the number of rows by changing the 10 to any other number of rows.
- If there are not many rows in the table, simply reduce the rows in the table.

Quiz: LIMIT

Can You Use LIMIT?

Try it yourself below by writing a query that limits the response to only the first 15 rows and includes the occurred_at, account_id, and channel fields in the **web_events** table.



ORDER BY

The **ORDER BY** statement allows us to order our table by any row. If you are familiar with Excel, this is similar to the sorting you can do with filters.

The **ORDER BY** statement is always after the **SELECT** and **FROM** statements, but it is before the **LIMIT** statement. As you learn additional commands, the order of these statements will matter more. If we are using the **LIMIT** statement, it will always appear last.

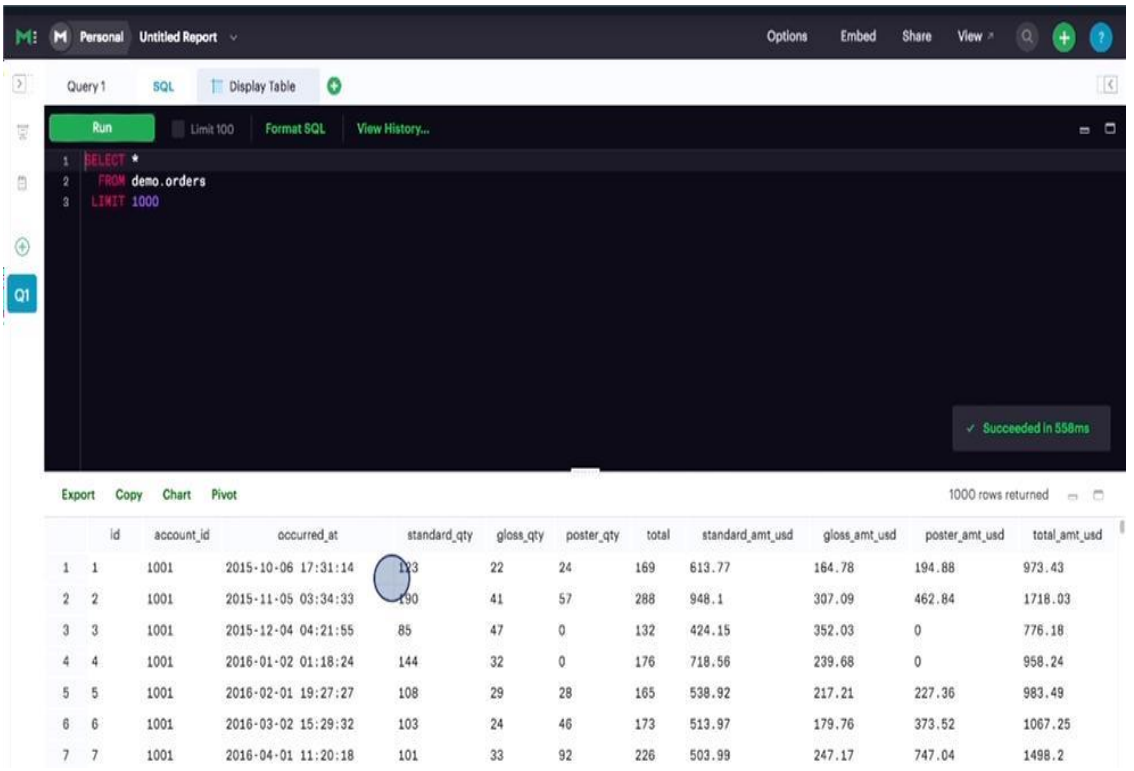
Pro Tip

Remember DESC can be added after the column in your **ORDER BY** statement to sort in descending order, as the default is to sort in ascending order.

For example, Suppose we are in the finance department in Burch and Bosey. We want to find the most recent orders so that we can send invoices to customers.

The words **ORDER BY** will help us To complete this, by allowing us to sort orders by date.

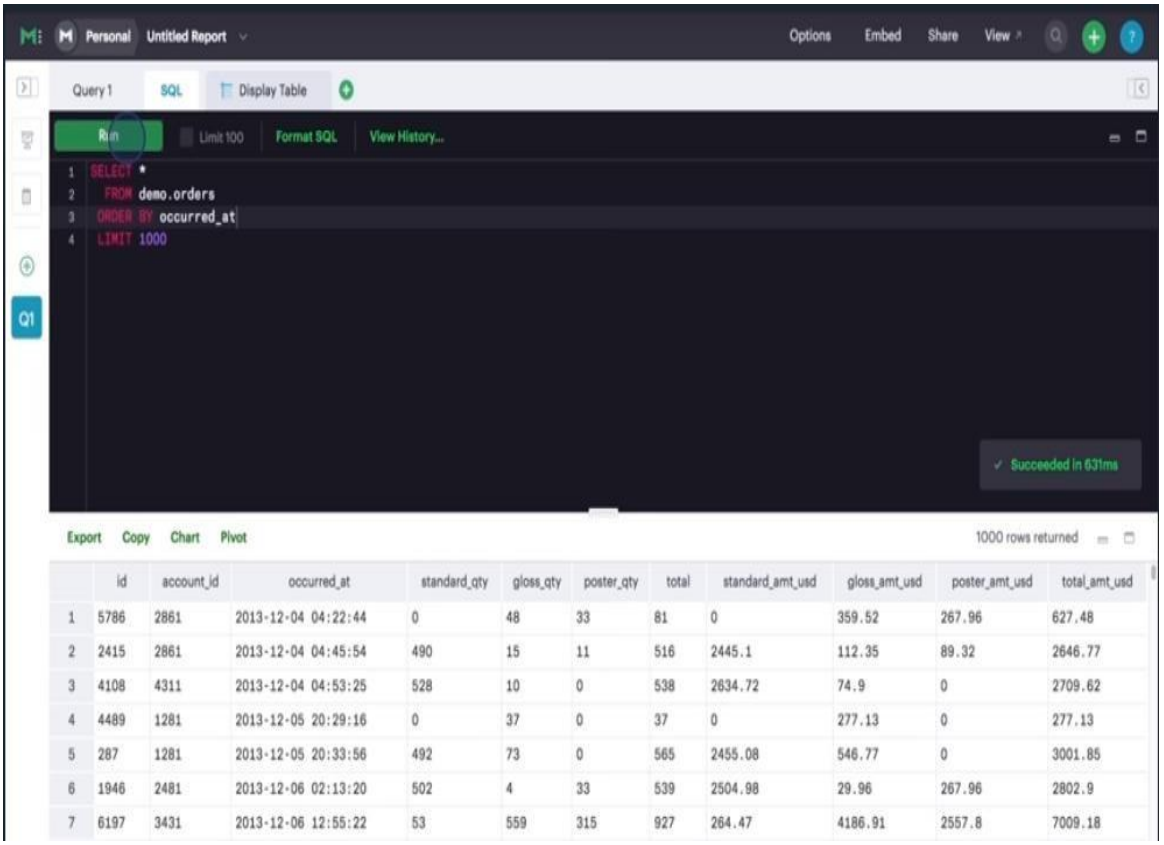
Let's look at how data is arranged by default. We'll notice that this table appears as if it was sorted by account ID by default. Starts from 1001 and starts to grow from here.



	id	account_id	occurred_at	standard_qty	gloss_qty	poster_qty	total	standard_amt_usd	gloss_amt_usd	poster_amt_usd	total_amt_usd
1	1	1001	2015-10-06 17:31:14	123	22	24	169	613.77	164.78	194.88	973.43
2	2	1001	2015-11-05 03:34:33	190	41	57	288	948.1	307.09	462.84	1718.03
3	3	1001	2015-12-04 04:21:55	85	47	0	132	424.15	352.03	0	776.18
4	4	1001	2016-01-02 01:18:24	144	32	0	176	718.56	239.68	0	958.24
5	5	1001	2016-02-01 19:27:27	108	29	28	165	538.92	217.21	227.36	983.49
6	6	1001	2016-03-02 15:29:32	103	24	46	173	513.97	179.76	373.52	1067.25
7	7	1001	2016-04-01 11:20:18	101	33	92	226	503.99	247.17	747.04	1498.2

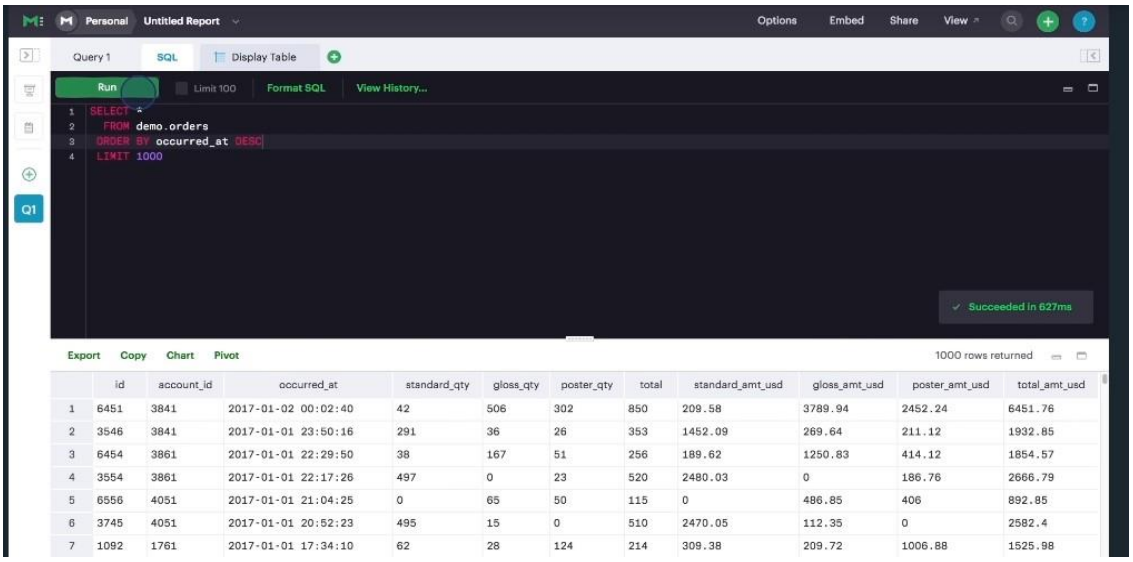
Let's ORDER BY clause to reorder the results based on the data the order was placed, which you can see in the occurred_at column. You have to write the clauses in this order, or the query will not run.

By default, order by goes from A – Z, lowest to highest, or earliest to latest, if working with dates.



	id	account_id	occurred_at	standard_qty	gloss_qty	poster_qty	total	standard_amt_usd	gloss_amt_usd	poster_amt_usd	total_amt_usd
1	5786	2861	2013-12-04 04:22:44	0	48	33	81	0	359.52	267.96	627.48
2	2415	2861	2013-12-04 04:45:54	490	15	11	516	2445.1	112.35	89.32	2646.77
3	4108	4311	2013-12-04 04:53:25	528	10	0	538	2634.72	74.9	0	2709.62
4	4489	1281	2013-12-05 20:29:16	0	37	0	37	0	277.13	0	277.13
5	287	1281	2013-12-05 20:33:56	492	73	0	565	2455.08	546.77	0	3001.85
6	1946	2481	2013-12-06 02:13:20	502	4	33	539	2504.98	29.96	267.96	2802.9
7	6197	3431	2013-12-06 12:55:22	53	559	315	927	264.47	4186.91	2557.8	7009.18

If you wont to order the other way, you can add desc, short for descending, to the end of the ORDER BY clause. This will get us to the data set that we're after, which shows the 10 most recent orders, with the most recent at the top.



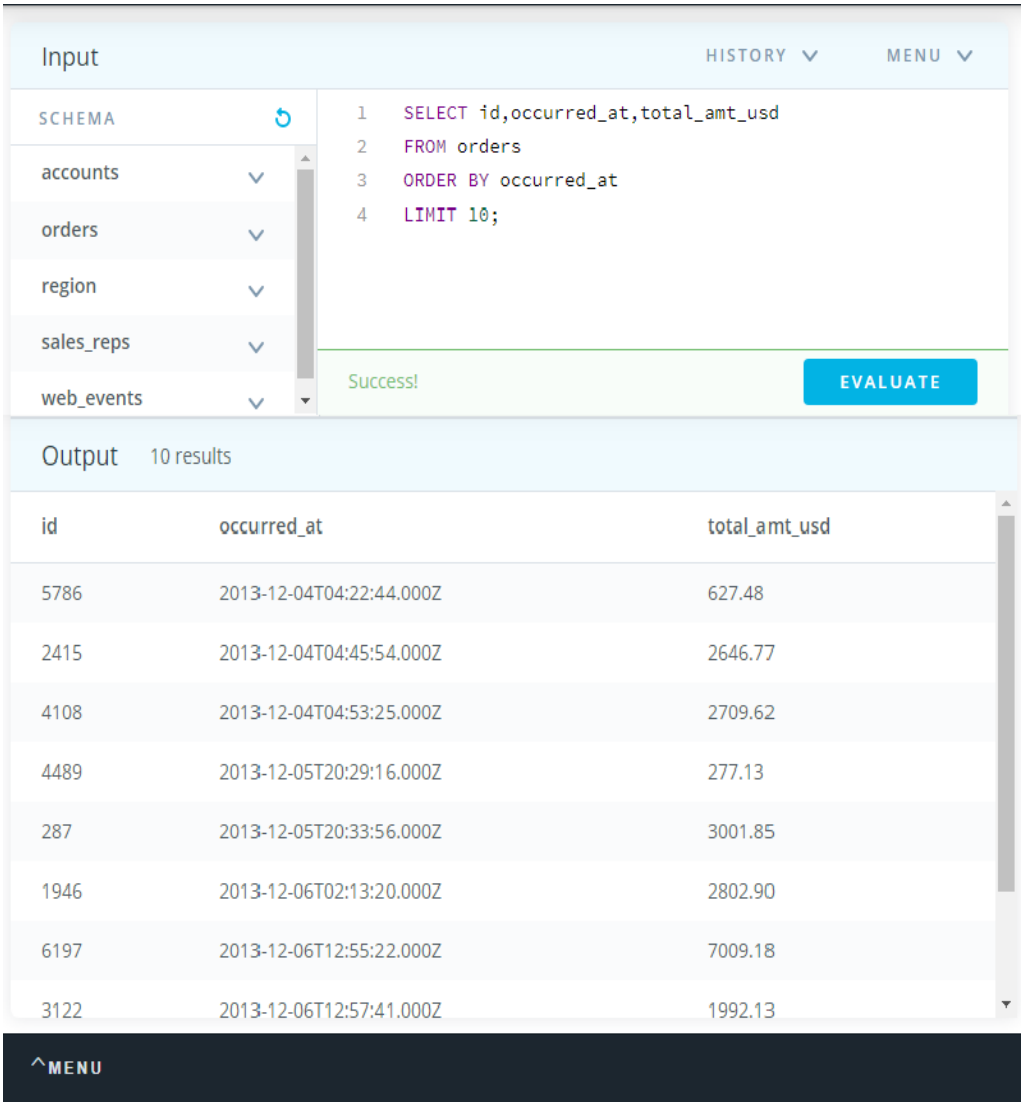
	id	account_id	occurred_at	standard_qty	gloss_qty	poster_qty	total	standard_amt_usd	gloss_amt_usd	poster_amt_usd	total_amt_usd
1	6451	3841	2017-01-02 00:02:40	42	506	302	850	209.58	3789.94	2452.24	6451.76
2	3546	3841	2017-01-01 23:50:16	291	36	26	353	1452.09	269.64	211.12	1932.85
3	6454	3861	2017-01-01 22:29:50	38	167	51	256	189.62	1250.83	414.12	1854.57
4	3554	3861	2017-01-01 22:17:26	497	0	23	520	2480.03	0	186.76	2666.79
5	6556	4051	2017-01-01 21:04:25	0	65	50	115	0	486.85	406	892.85
6	3745	4051	2017-01-01 20:52:23	495	15	0	510	2470.05	112.35	0	2582.4
7	1092	1761	2017-01-01 17:34:10	62	28	124	214	309.38	209.72	1006.88	1525.98

Quiz: ORDER BY

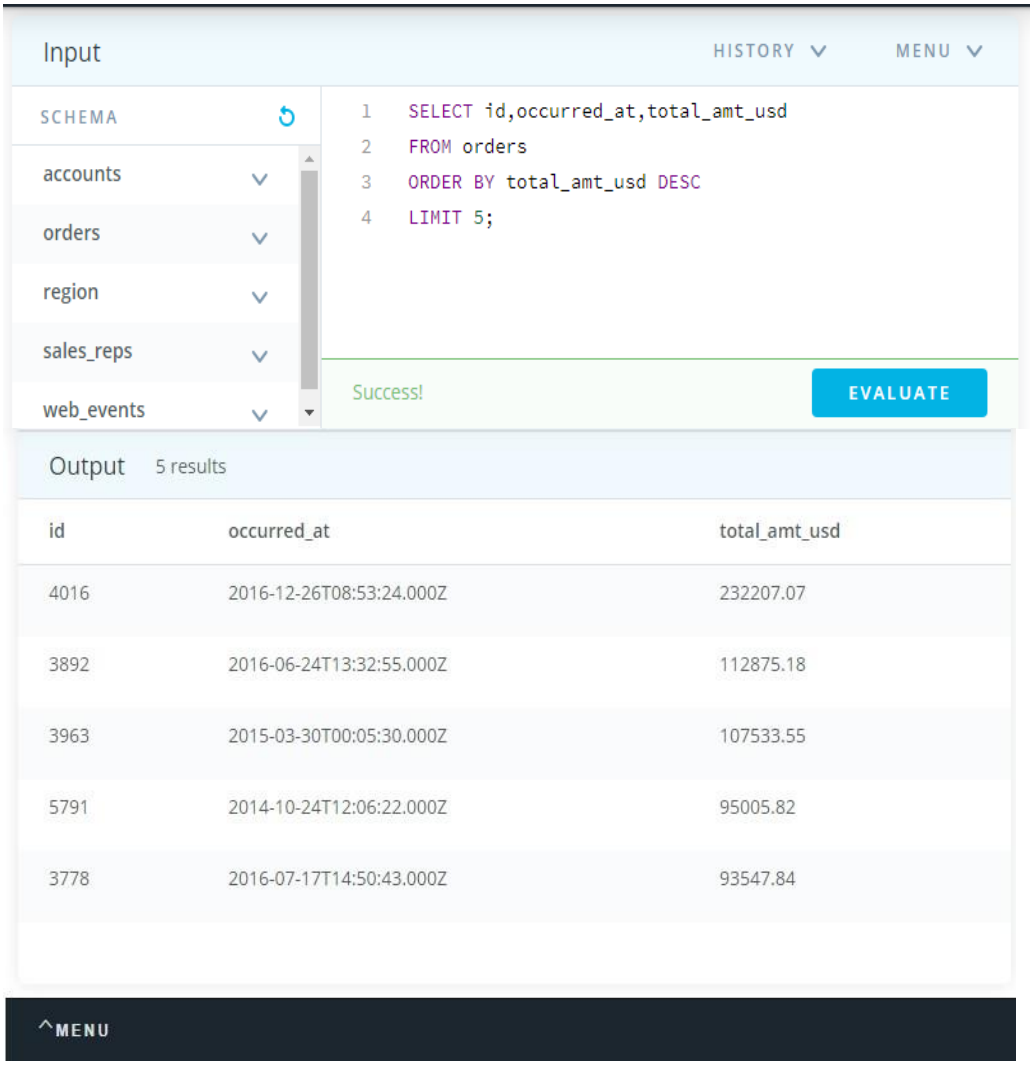
Practice

In order to gain some practice using **ORDER BY**:

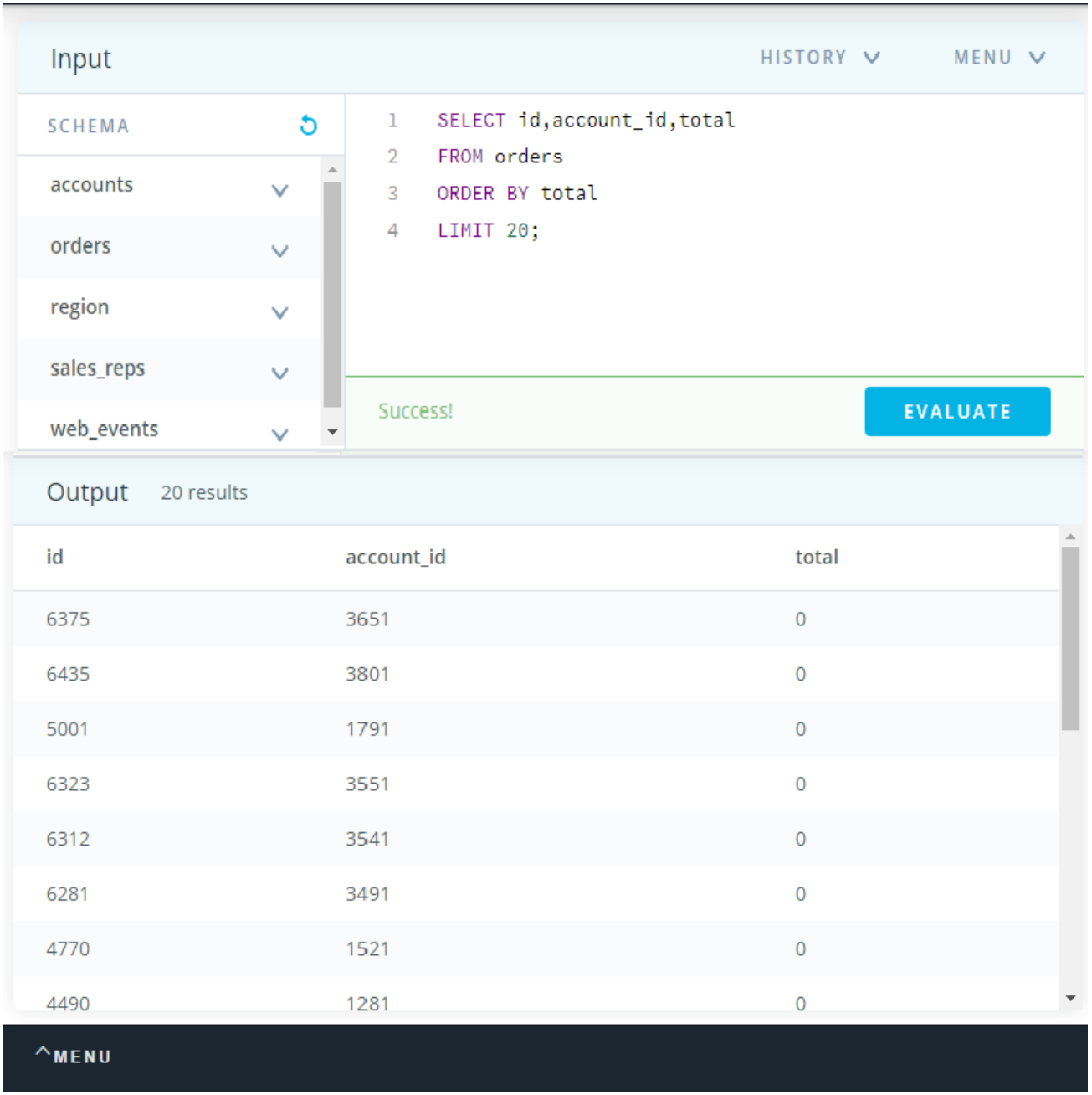
1. Write a query to return the 10 earliest orders in the **orders** table. Include the id, occurred_at, and total_amt_usd.



2. Write a query to return the top 5 **orders** in terms of largest total_amt_usd.



3. Write a query to return the bottom 20 **orders** in terms of least total. Include the id, account_id, and total.



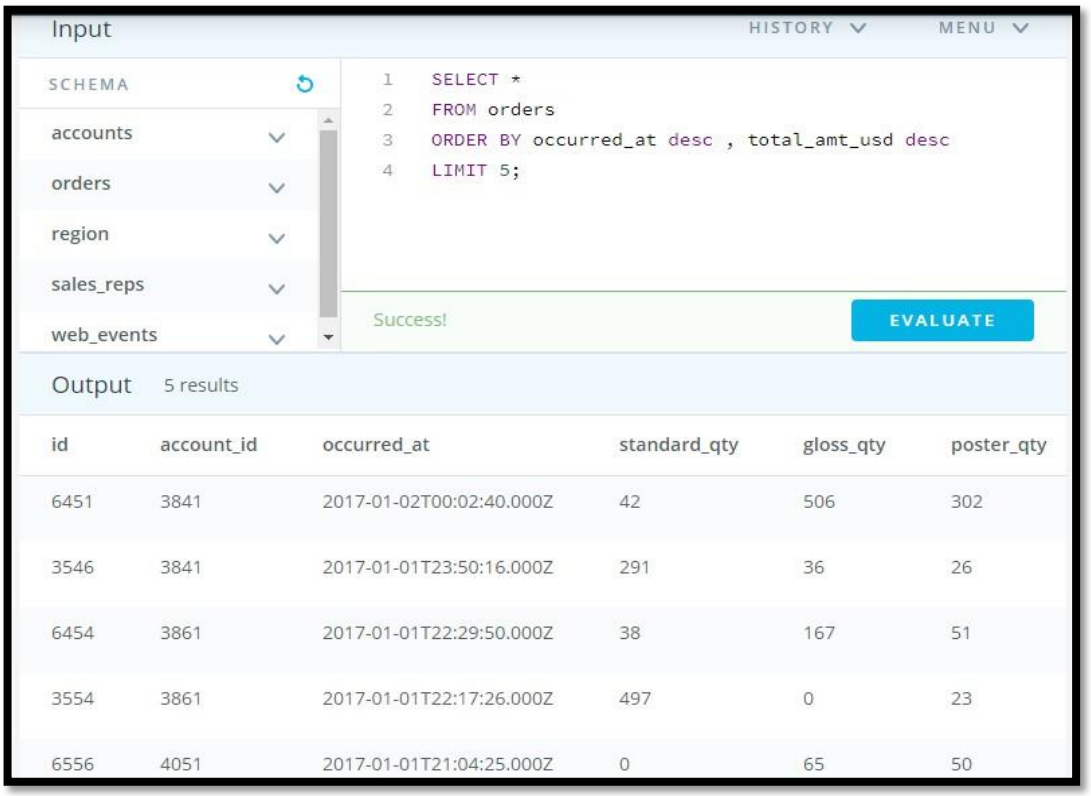
Quiz: ORDER BY Part II

You will notice that many of the queries that you write in this first lesson might not produce exactly what you would hope, but by the end of all the lessons, you should have the skills to better tackle these problems.

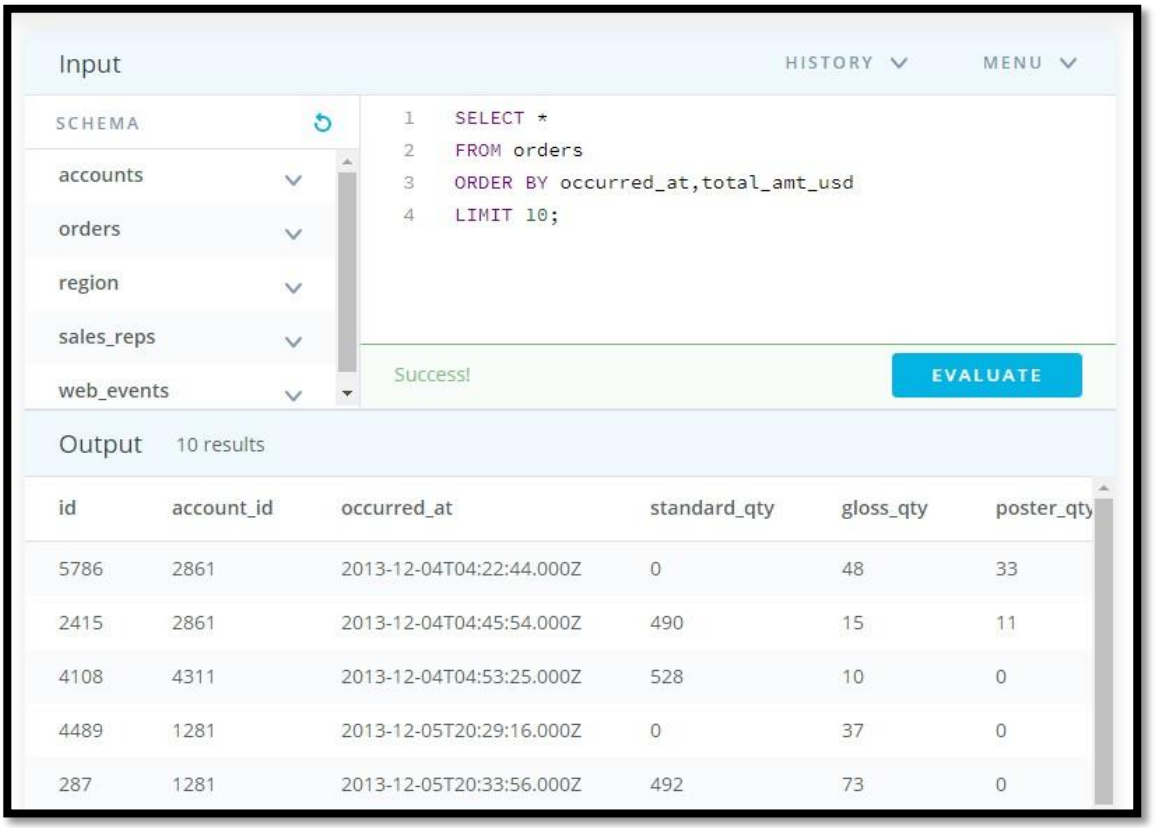
Notice that the **ORDERS** table is quite large, so additional columns are in the right of the output you immediately see.

Questions

1. Write a query that returns the top 5 rows from **orders** ordered according to newest to oldest, but with the largest total_amt_usd for each date listed first for each date. **You will notice each of these dates shows up as unique because of the time element. When you learn about truncating dates in a later lesson, you will better be able to tackle this question on a day, month, or yearly basis.**



2. Write a query that returns the top 10 rows from **orders** ordered according to oldest to newest, but with the smallest total_amt_usd for each date listed first for each date. **You will notice each of these dates shows up as unique because of the time element. When you learn about truncating dates in a later lesson, you will better be able to tackle this question on a day, month, or yearly basis.**



id	account_id	occurred_at	standard_qty	gloss_qty	poster_qty
5786	2861	2013-12-04T04:22:44.000Z	0	48	33
2415	2861	2013-12-04T04:45:54.000Z	490	15	11
4108	4311	2013-12-04T04:53:25.000Z	528	10	0
4489	1281	2013-12-05T20:29:16.000Z	0	37	0
287	1281	2013-12-05T20:33:56.000Z	492	73	0

WHERE

Using the **WHERE** statement, we can subset out tables based on conditions that must be met. In the next sections, you will learn some of the common operators that are used with the **WHERE** statement.

Common symbols used within WHERE statements include:

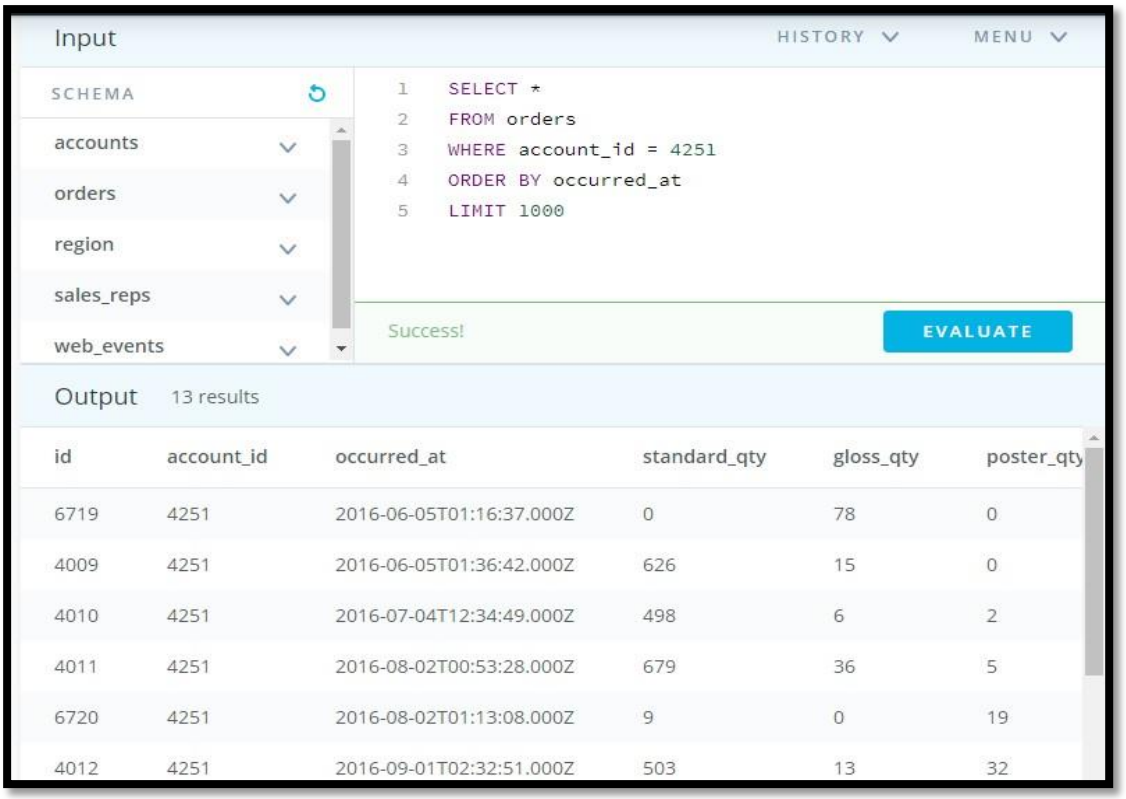
1. > (greater than)
2. < (less than)
3. >= (greater than or equal to)
4. <= (less than or equal to)
5. = (equal to)
6. != (not equal to)

Example :

Assume that an account manager at parch & posey is about to go to one of the most important clients.

- We can use WHERE to create a list of all purchases made by that particular customer. It also allows us to filter a set of results based on certain criteria.
- WHERE comes after FROM but before ORDER BY & LIMIT.
- Phrases must be in the correct order and the query will be returned by mistake.

we'll write a query to show only orders from our top customer, which is represented here by account ID 4251.we'll write this simple equation where account ID is equal to 4251. This will produce a result set that includes all rows for which the value in account ID is equal to 4251.



When using SQL entire rows of data are preserved intact. If you write a WHERE clause filters based on values in one column as we have done here you'll limit the result in all columns to rows that satisfy the condition.

The idea is that each row is one data point or observation and all the information contained in that row belongs together.

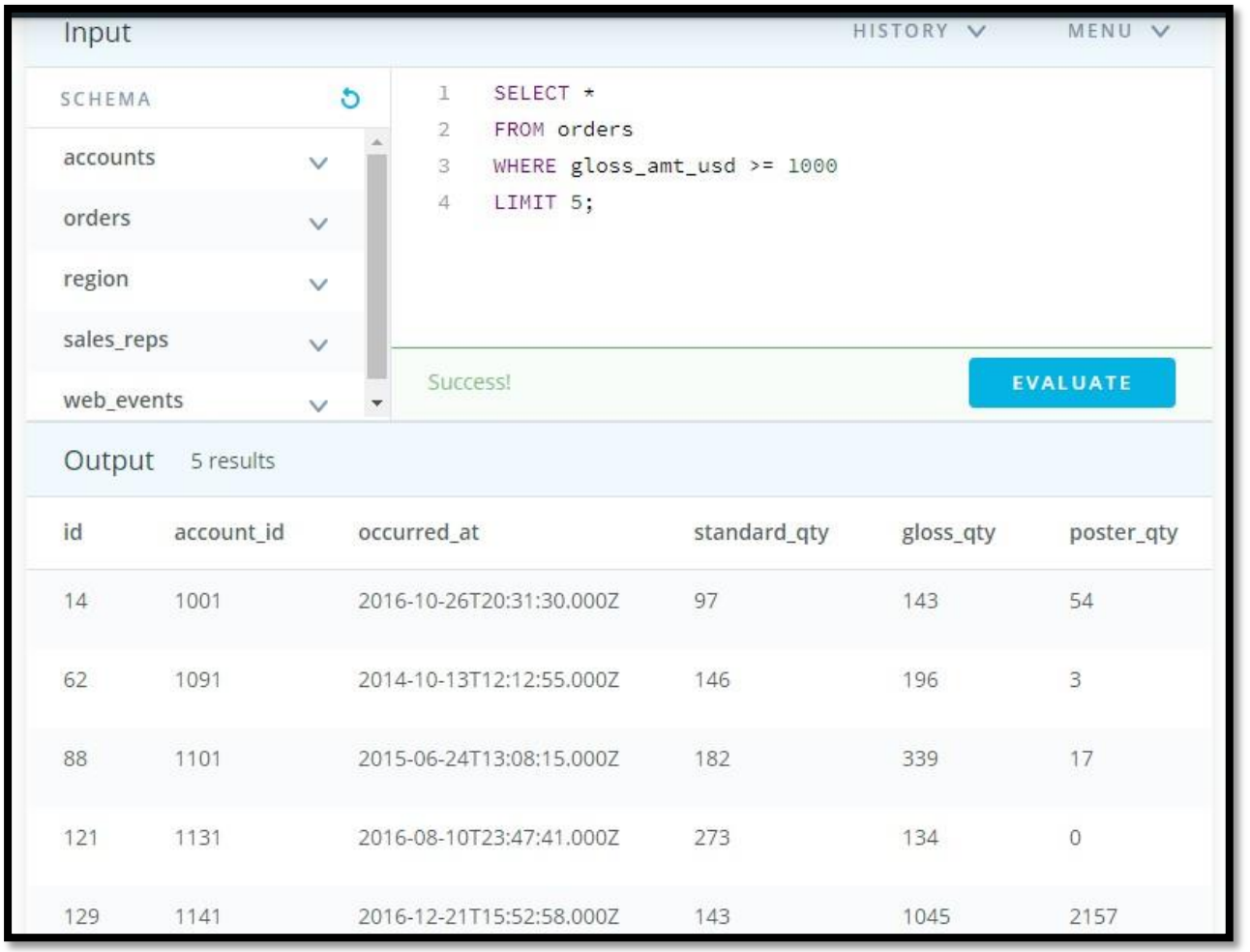
All the information in one row is about one order of paper.

Quiz: WHERE

Questions

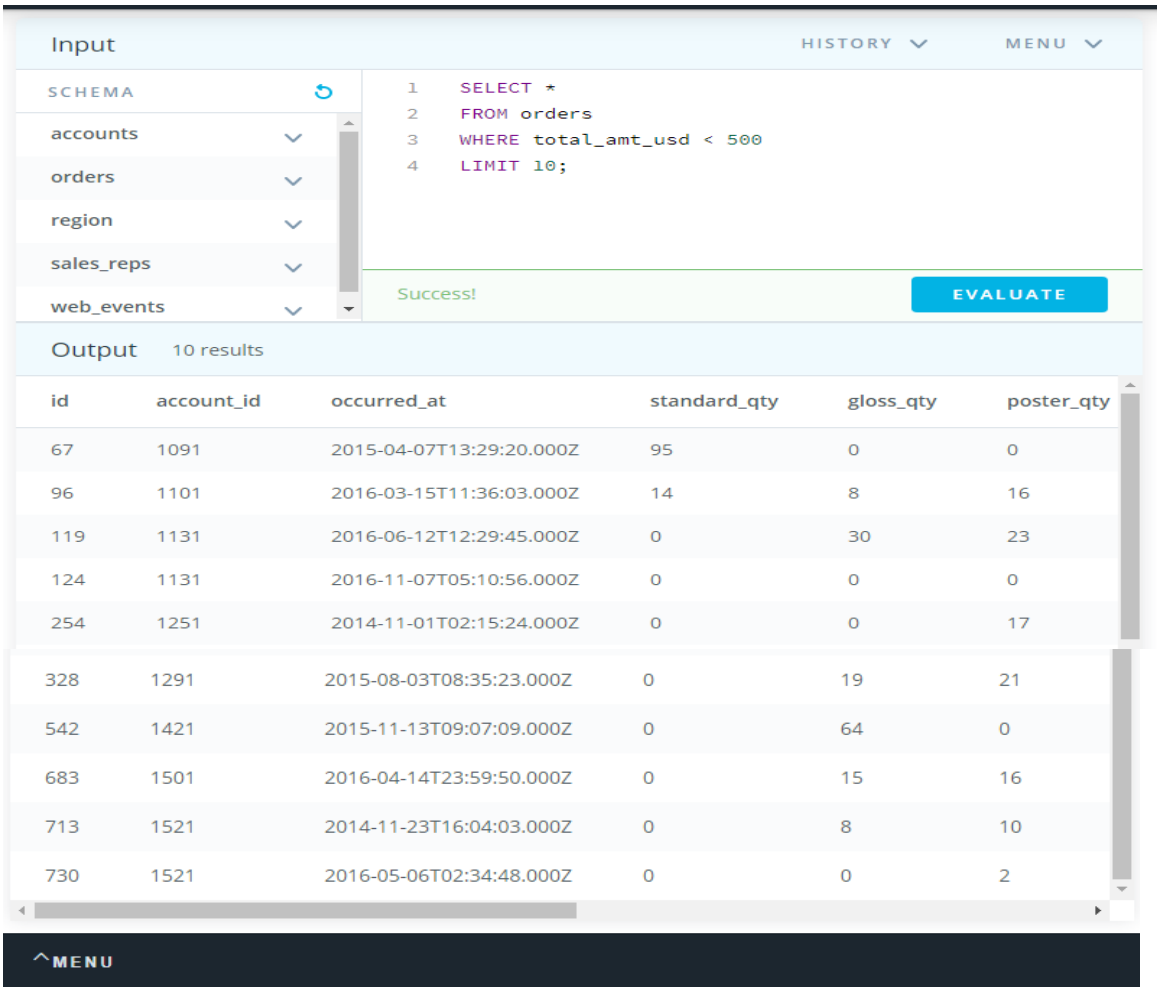
Write a query that

1. Pull the first 5 rows and all columns from the **orders** table that have a dollar amount of gloss_amt_usd greater than or equal to 1000.



id	account_id	occurred_at	standard_qty	gloss_qty	poster_qty
14	1001	2016-10-26T20:31:30.000Z	97	143	54
62	1091	2014-10-13T12:12:55.000Z	146	196	3
88	1101	2015-06-24T13:08:15.000Z	182	339	17
121	1131	2016-08-10T23:47:41.000Z	273	134	0
129	1141	2016-12-21T15:52:58.000Z	143	1045	2157

2. Pull the first 10 rows and all columns from the **orders** table that have a total_amt_usd less than 500.



You will notice when using these **WHERE** statements, we do not need to **ORDER BY** unless we want to actually order our data. Our condition will work without having to do any sorting of the data.

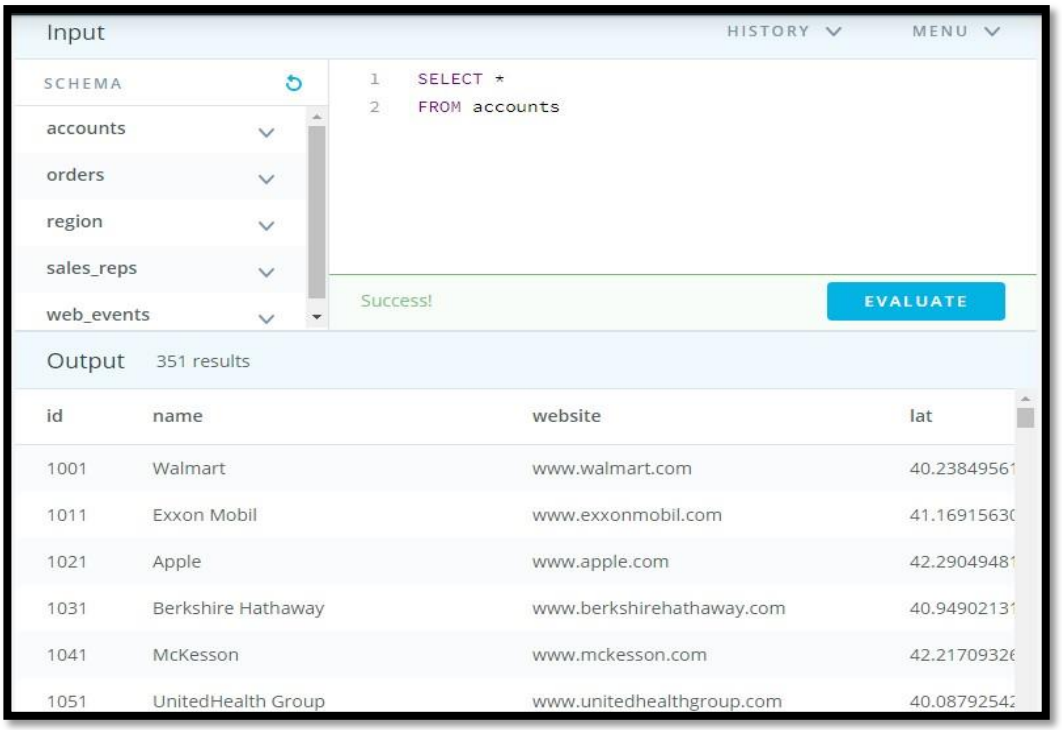
WHERE with Non-Numeric Data

The **WHERE** statement can also be used with non-numerical data. We can use the = and != operators here. You also need to be sure to use single quotes (just be careful if you have quotes in the original text) with the text data.

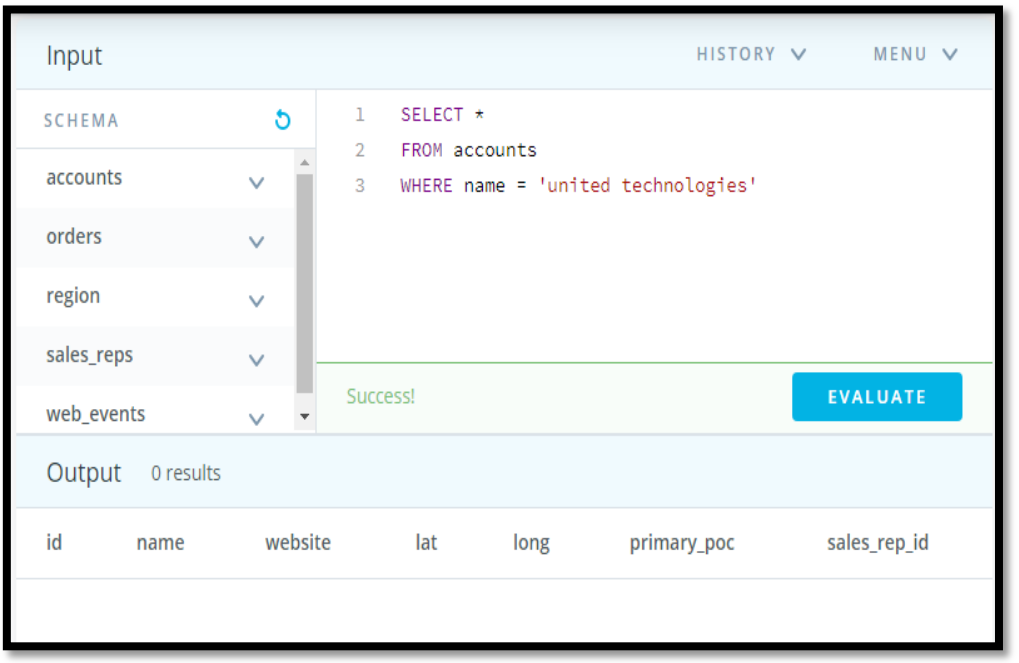
Commonly when we are using **WHERE** with non-numeric data fields, we use the **LIKE**, **NOT**, or **IN** operators. We will see those before the end of this lesson!

Comparison operators can work with non-numerical data as well. Equals and not equal to make perfect sets. They allow you to select rows that match or don't match any value.

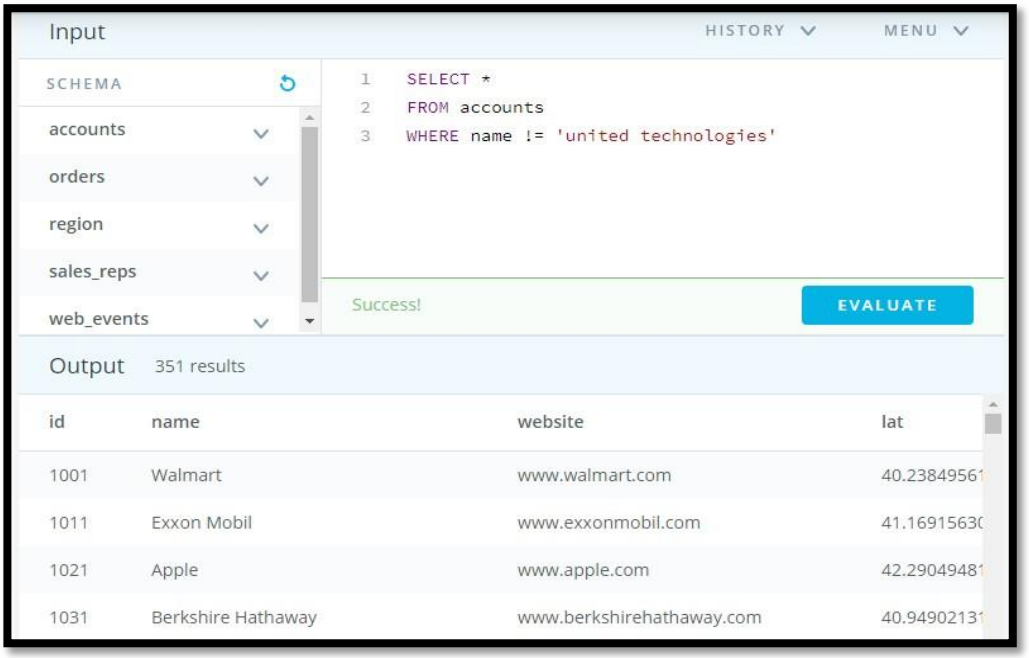
Let's look at an example from Parch & Posey accounts table which simply lists all of their customers and tell attributes about them as you see here.



There are some important rules when using these operators. If you're using an operator with values that are non-numeric, you need to put the value in single quotes. As you can see, this data set has been filtered down to the single row that matches the condition in our WHERE clause.



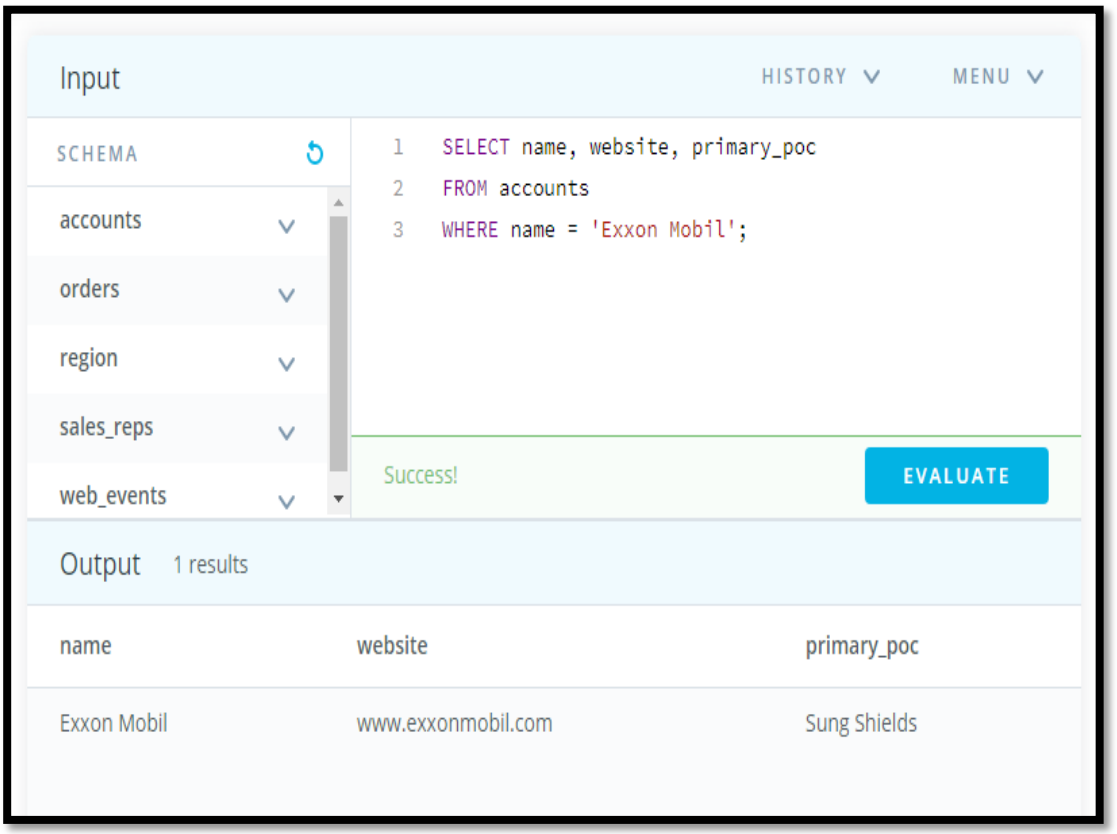
If you change the operator to not equals, the results will show every row except the one for united technologies



Quiz: WHERE with Non-Numeric

Practice Question Using WHERE with Non-Numeric Data

1. Filter the accounts table to include the company name, website, and the primary point of contact (primary_poc) for **Exxon Mobil** in the **accounts** table.



Arithmetic Operators

Derived Columns

Creating a new column that is a combination of existing columns is known as a **derived** column.

Common operators include:

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

Order of Operations

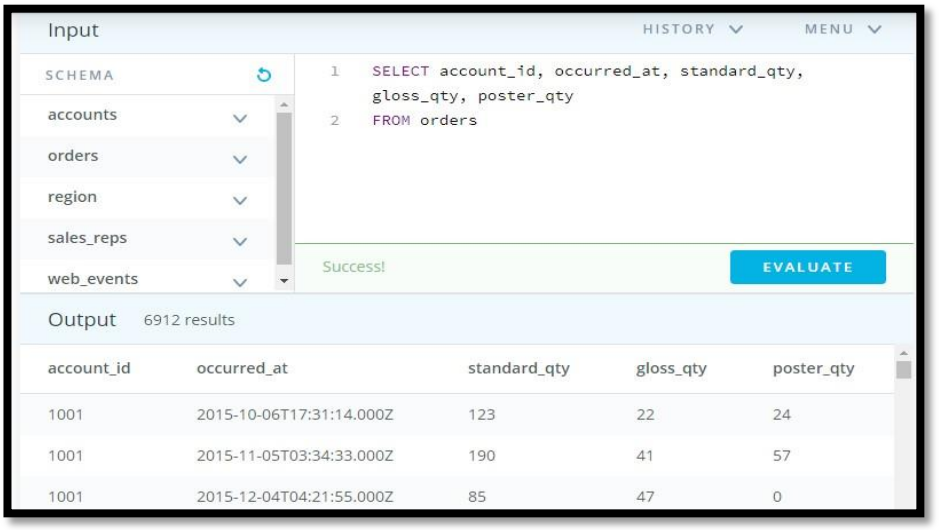
The following two statements have very different end results:

1. **Standard_qty / standard_qty + gloss_qty + poster_qty**
2. **standard_qty / (standard_qty + gloss_qty + poster_qty)**

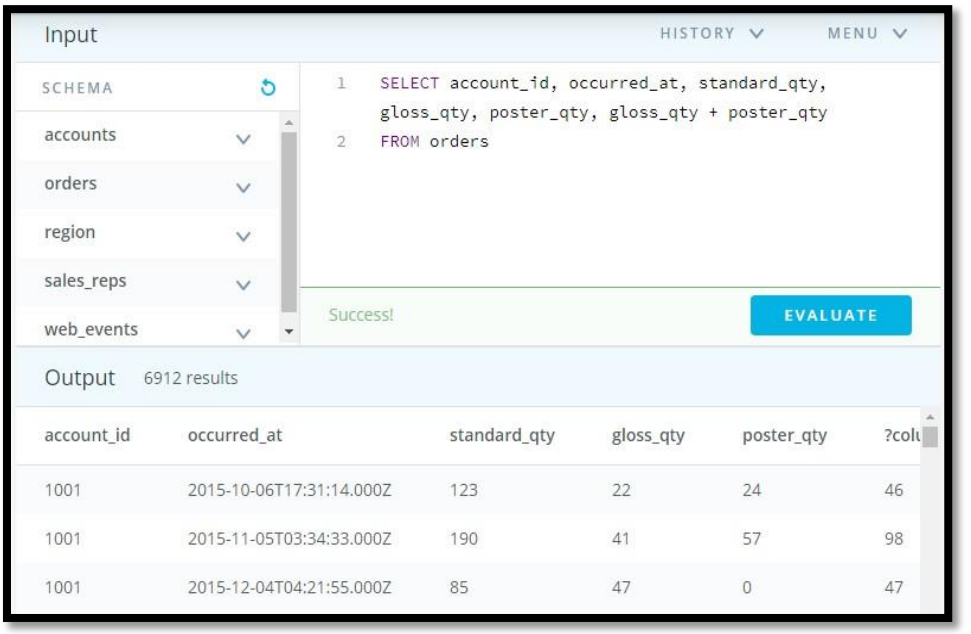
It is likely the case you mean to calculate the statement in part 2.

Example:

Before we calculate how much non standard paper was sold, let's check out the order quantities in each column of the orders table.



Now let's create a new column that adds poster and gloss paper together to create a filed for non standard paper. to do this, we'll create a new line in the query and use an arithmetic operator.

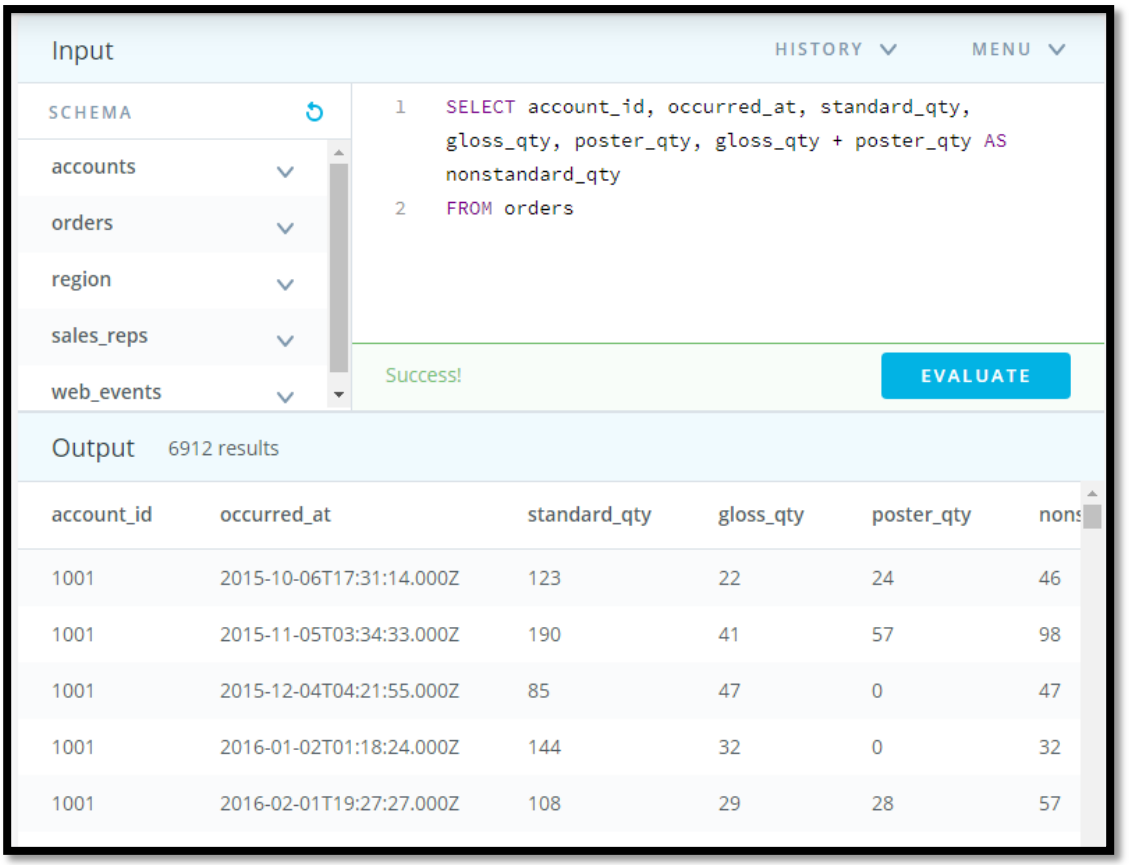


You'll notice that the values in this new column are equal to the sum of the gloss and poster quantities on a row-by-row basis.

In the first row, 22 gloss and 24 poster sum to 46 nonstandard in this new column we've created.

There's a special name for this. It's a derived column. Derived columns can include simple arithmetic or any number of advance calculations. What makes them special is the fact that they're not just directly selected from the underlying data, they're manipulated in some way.

Rather than stick with this ugly column name, let's add something more descriptive using an Alias. We can do this by adding AS to the end of the line that produces the derived column and then giving it a name.



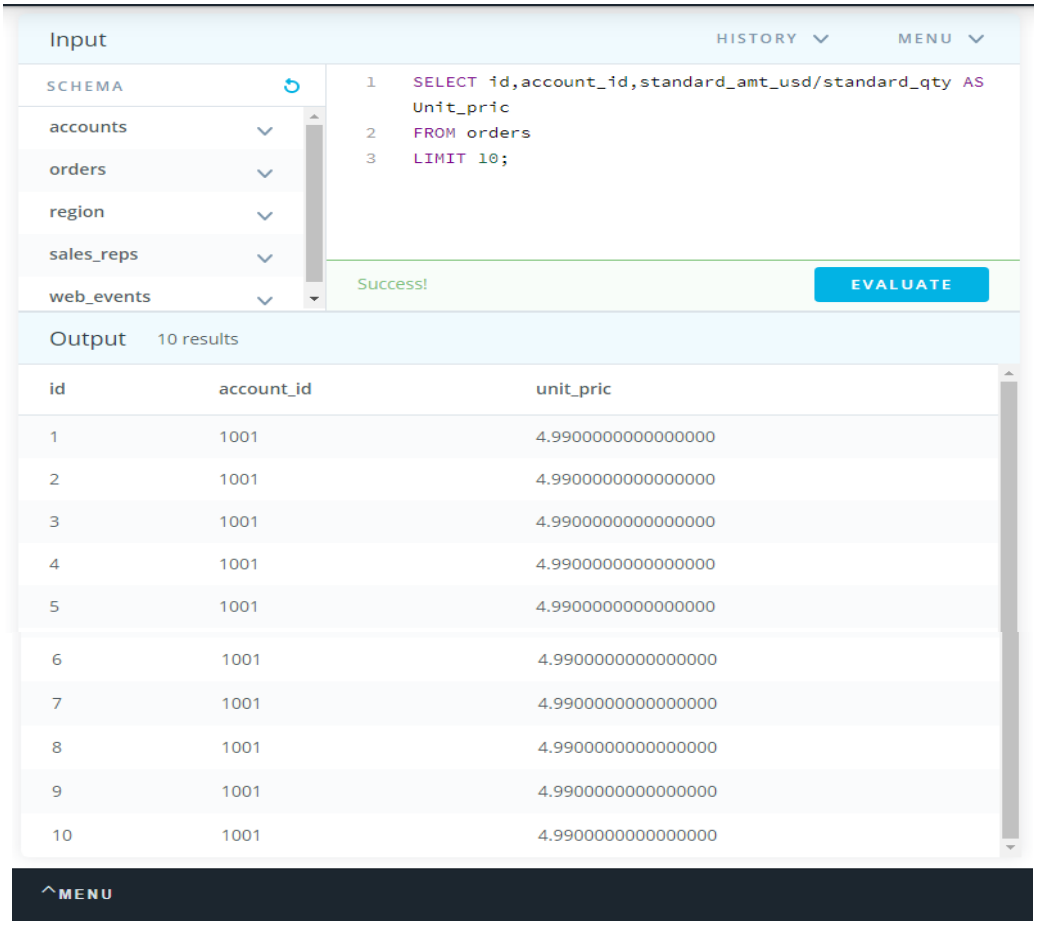
The best practice here is to use names that are clearly descriptive, follow existing conventions, and don't include capital letters or space. We'll use nonstandard quantity.

Quiz: Arithmetic Operators

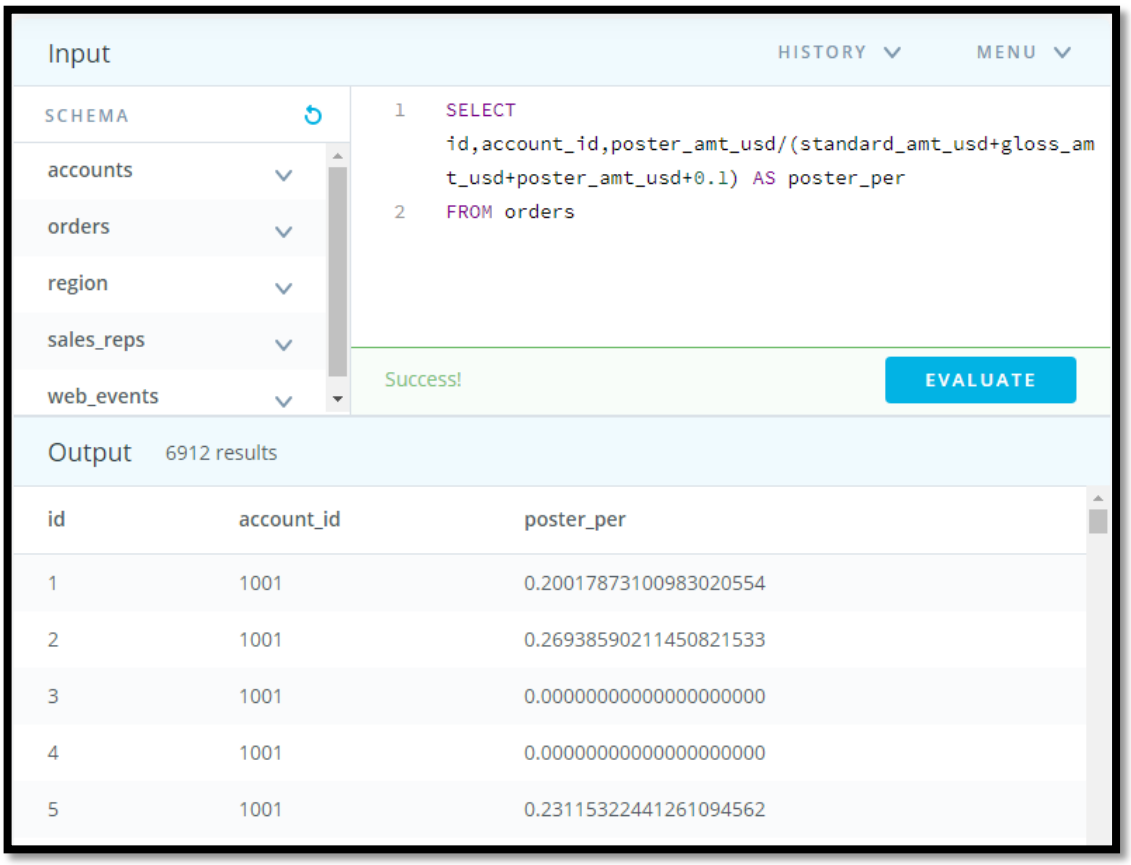
Questions using Arithmetic Operations

Using the **orders** table:

1. Create a column that divides the standard_amt_usd by the standard_qty to find the unit price for standard paper for each order. Limit the results to the first 10 orders, and include the id and account_id fields.



2. Write a query that finds the percentage of revenue that comes from poster paper for each order. You will need to use only the columns that end with _usd. (Try to do this without using the total column). Include the id and account_id fields. **NOTE - you will be thrown an error with the correct solution to this question. This is for a division by zero. You will learn how to get a solution without an error to this query when you learn about CASE statements in a later section. For now, you might just add some very small value to your denominator as a work around.**



Notice, the above operators combine information across columns for the same row. If you want to combine values of a particular column, across multiple rows, we will do this with aggregations. We will get to that before the end of the course!

Introduction to Logical Operators

Introduction to Logical Operators

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

Logical Operators include:

1. **LIKE**

   This allows you to perform operations similar to using **WHERE** and **=**, but for cases when you might **not** know **exactly** what you are looking for.

2. **IN**

   This allows you to perform operations similar to using **WHERE** and **=**, but for more than one condition.

3. **NOT**

   This is used with **IN** and **LIKE** to select all of the rows **NOT LIKE** or **NOT IN** a certain condition.

4. **AND & BETWEEN**

   These allow you to combine operations where all combined conditions must be true.

5. **OR**

   This allow you to combine operations where at least one of the combined conditions must be true.

Don't worry if this doesn't make total sense right now. You will get practice with each in the next sections.

LIKE

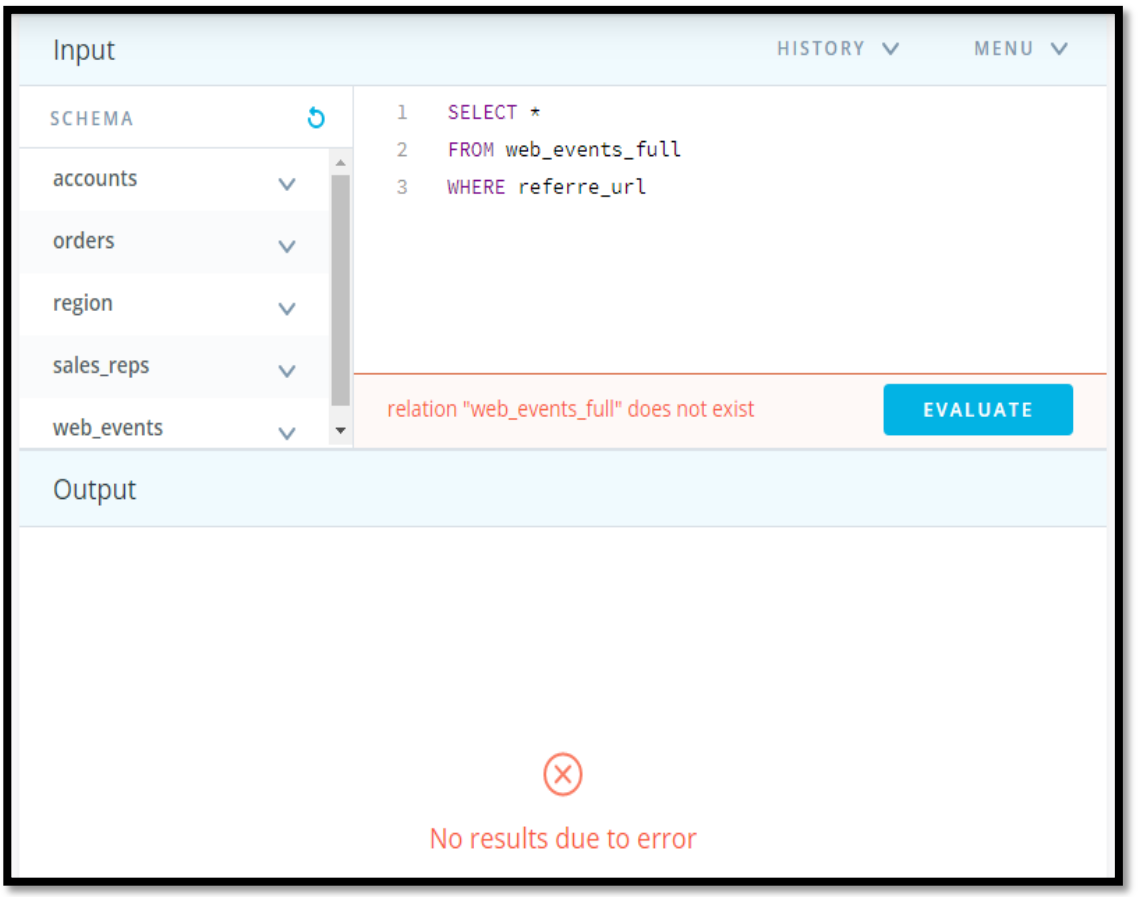
The **LIKE** operator is extremely useful for working with text. You will use **LIKE** within a **WHERE** clause. The **LIKE** operator is frequently used with %. The % tells us that we might want any number of characters leading up to a particular set of characters or following a certain set of characters. Remember you will need to use single quotes for the text you pass to the **LIKE** operator, because of this lower and uppercase letters are not the same within the string. Searching for **'T'** is not the same as searching for **'t'**. In other SQL environments (outside the classroom), you can use either single or double quotes.

Hopefully you are starting to get more comfortable with SQL, as we are starting to move toward operations that have more applications, but this also means we can't show you every use case. Hopefully, you can start to think about how you might use these types of applications to identify phone numbers from a certain region, or an individual where you can't quite remember the full name.

Example:

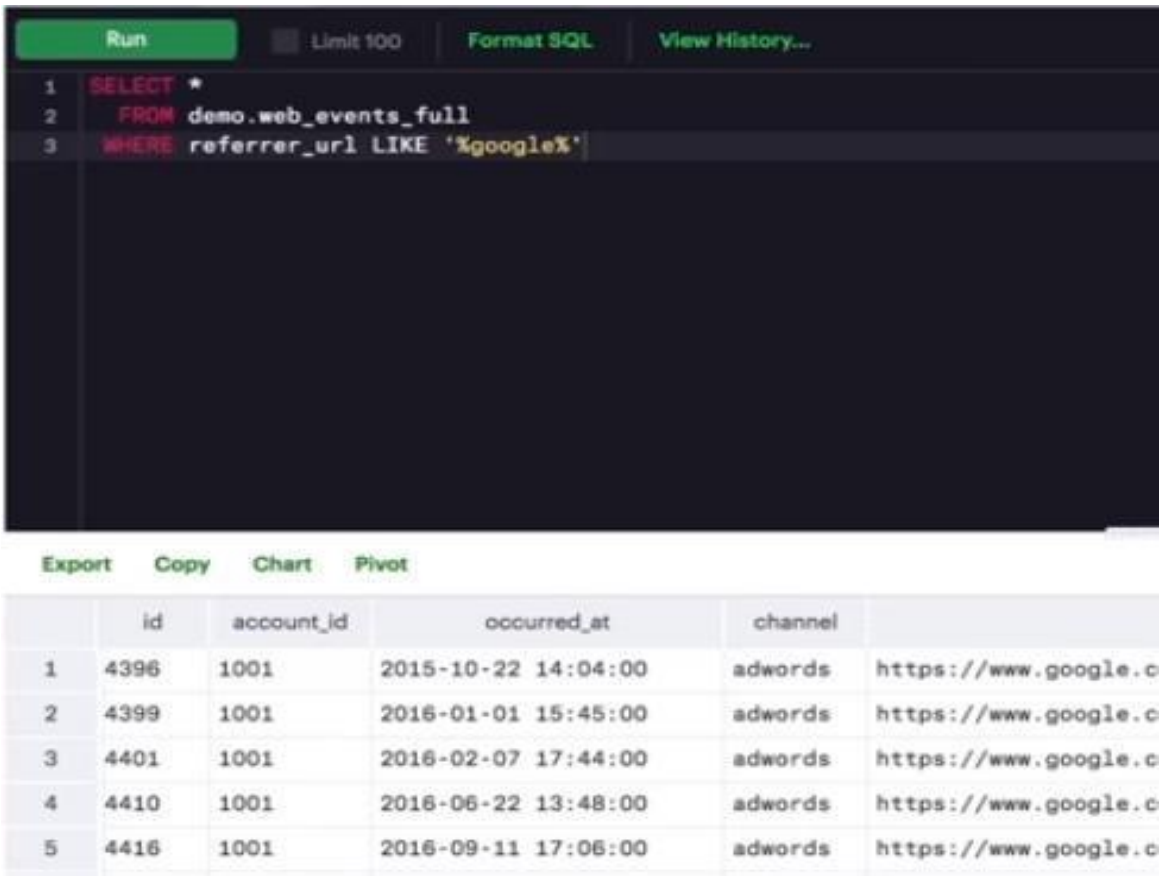
Suppose a digital marketing manager at Parch & Posey wants to determine the web-based traffic that came from any page in Google. There is a problem though.

As you can see, the referrer URLs from Google all have the same domain but often have a bunch of extra stuff tacked on. In order to write a filter that capture all web traffic from Google regardless of the stuff that's tacked onto the end, you can use the LIKE operator. But first, let's see what happens if I just use an equal sign in the WHERE clause. As you can see, we get back zero results.



Now I'll use the LIKE operator to find all rows for which the referrer URL is similar to Google but with any number of characters on either side.

You'll notice that the LIKE function requires the use of wildcards. So in this case, the wildcard we're using is a percent sign.



	id	account_id	occurred_at	channel	
1	4396	1001	2015-10-22 14:04:00	adwords	https://www.google.c
2	4399	1001	2016-01-01 15:45:00	adwords	https://www.google.c
3	4401	1001	2016-02-07 17:44:00	adwords	https://www.google.c
4	4410	1001	2016-06-22 13:48:00	adwords	https://www.google.c
5	4416	1001	2016-09-11 17:06:00	adwords	https://www.google.c

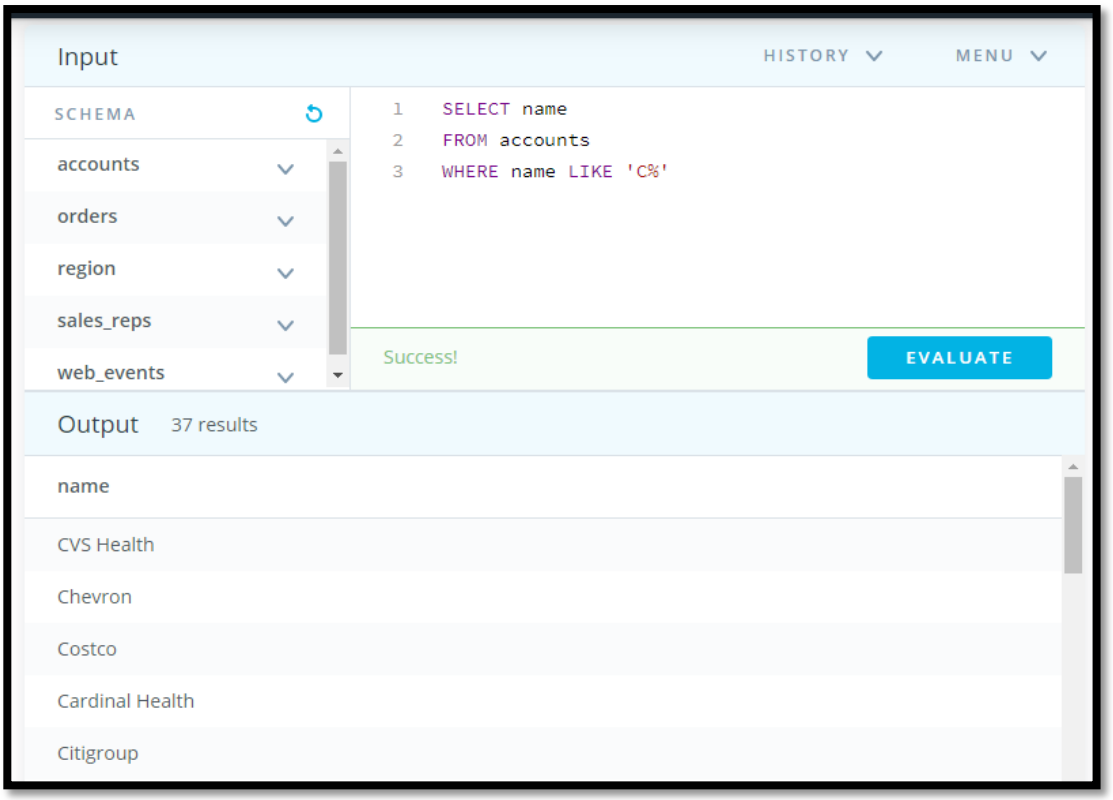
Each of these percent signs is used to represent a character or any number of characters, whatever those characters may be. So in this case, we're looking to match referrer URL for values that start with a character or any number of characters then contain the text Google, then have any number of characters on the end.

Quiz: LIKE

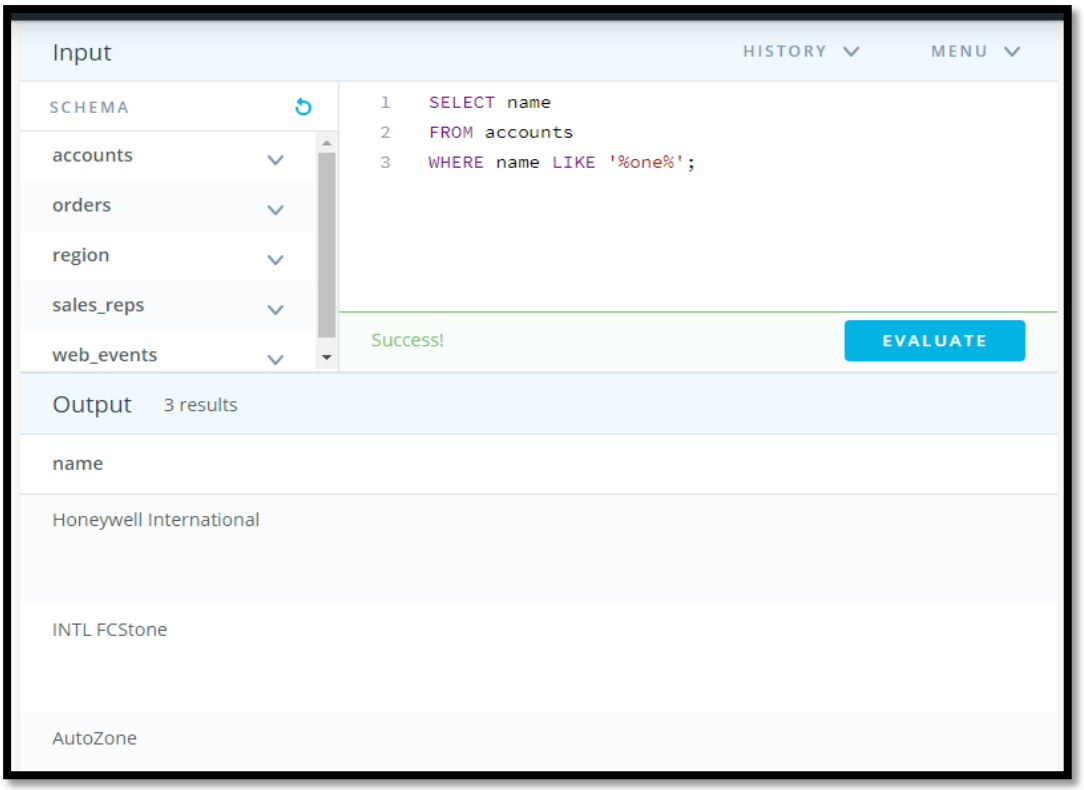
Questions using the LIKE operator

Use the **accounts** table to find

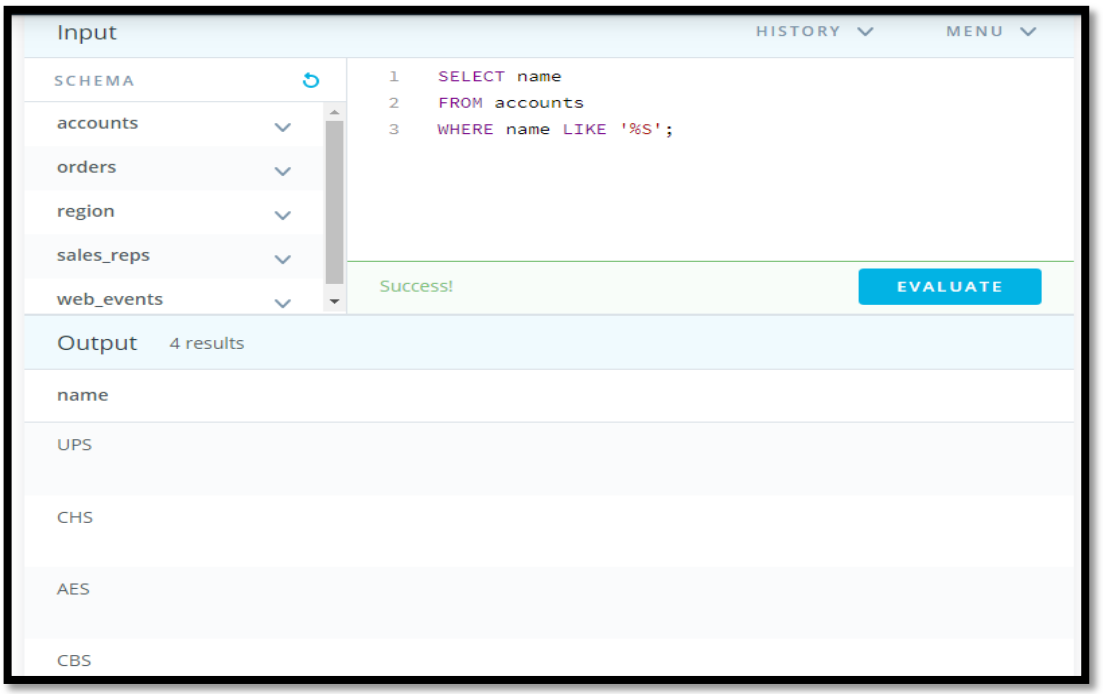
1. All the companies whose names start with 'C'.



2. All companies whose names contain the string 'one' somewhere in the name.



3. All companies whose names end with 's'.



IN

The **IN** operator is useful for working with both numeric and text columns. This operator allows you to use an =, but for more than one item of that particular column. We can check one, two or many column values for which we want to pull data, but all within the same query. In the upcoming concepts, you will see the **OR** operator that would also allow us to perform these tasks, but the **IN** operator is a cleaner way to write these queries.

Expert Tip

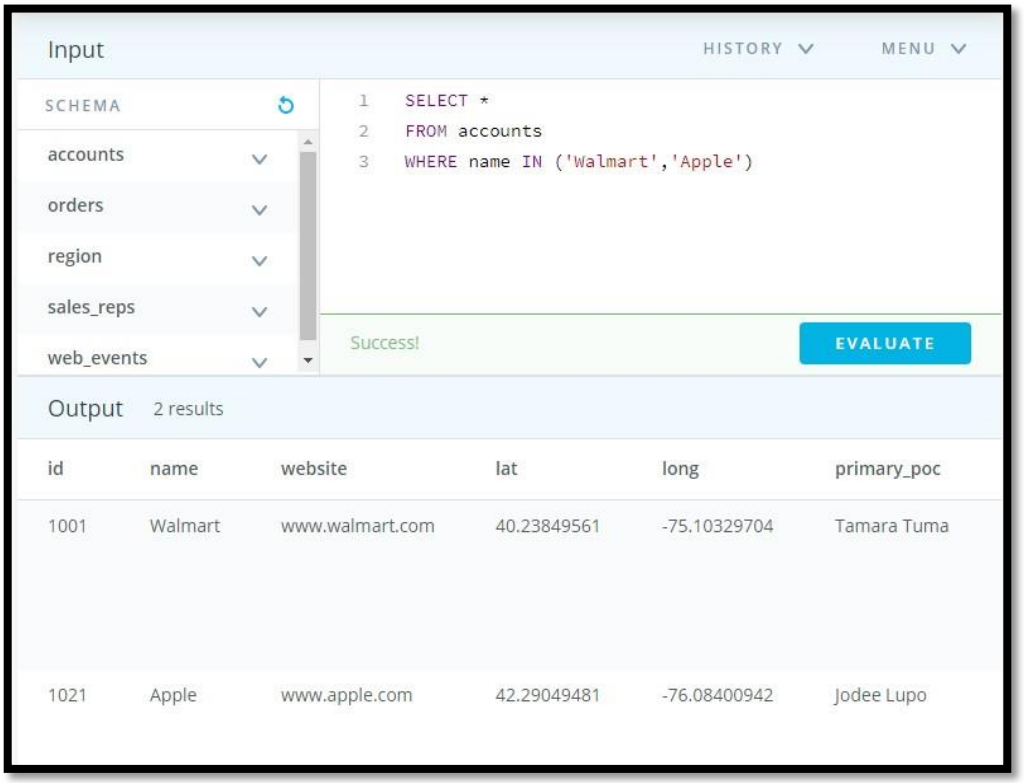
In most SQL environments, you can use single or double quotation marks - and you may NEED to use double quotation marks if you have an apostrophe within the text you are attempting to pull.

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

Example:

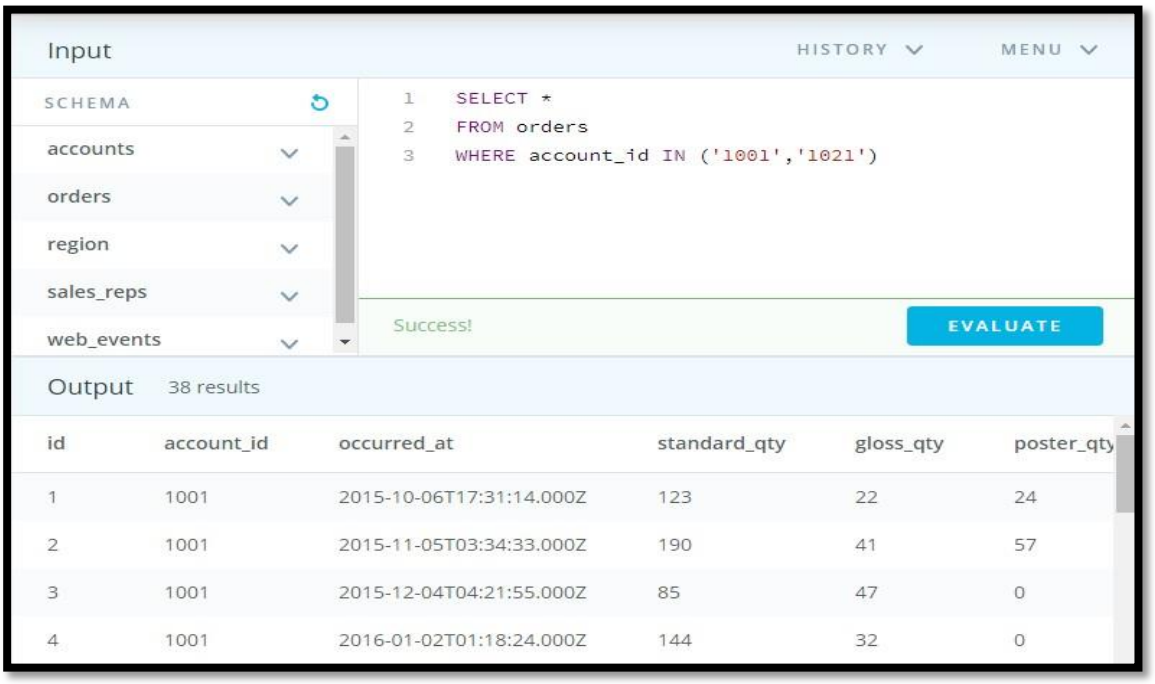
Suppose a digital marketing manager at Parch & Posey wants to determine the web-based traffic that came from any page in Google. There is a problem though.

The IN function will allow you to filter data based on several possible values. So, if you only want to see information for the Walmart & Apple accounts, here's how you do it.



As with comparison operators, IN requires single quotation marks around non-numerical data can be entered directly. Let's look at the actual orders associated with those accounts.

First, make note of the account IDs here, 1001 for Walmart and 1021 for Apple. Now we see of all the individual orders that come us for account 1001&1021.



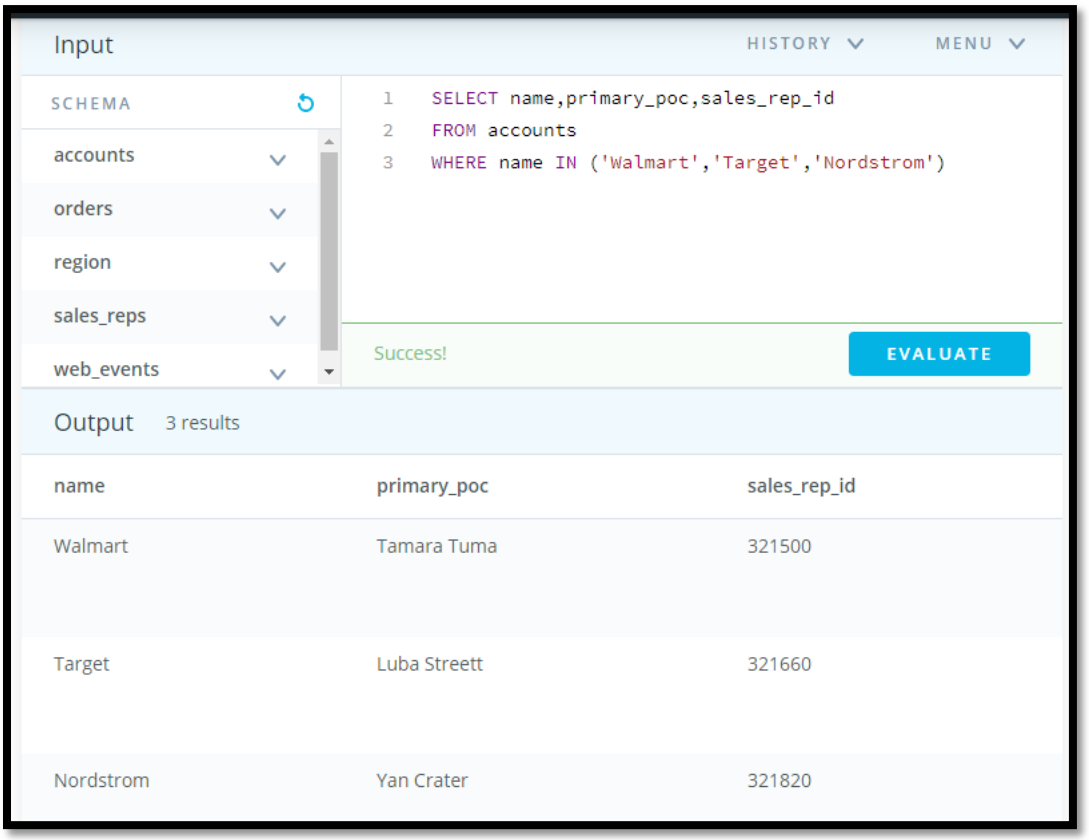
And as you can see, they're all jumbled together because there is no ORDER BY clause in the query. One thing to note here is the comma between these two values.

You can put as many values as you want inside these parentheses, but you have to put a comma in between each pair of distinct values.

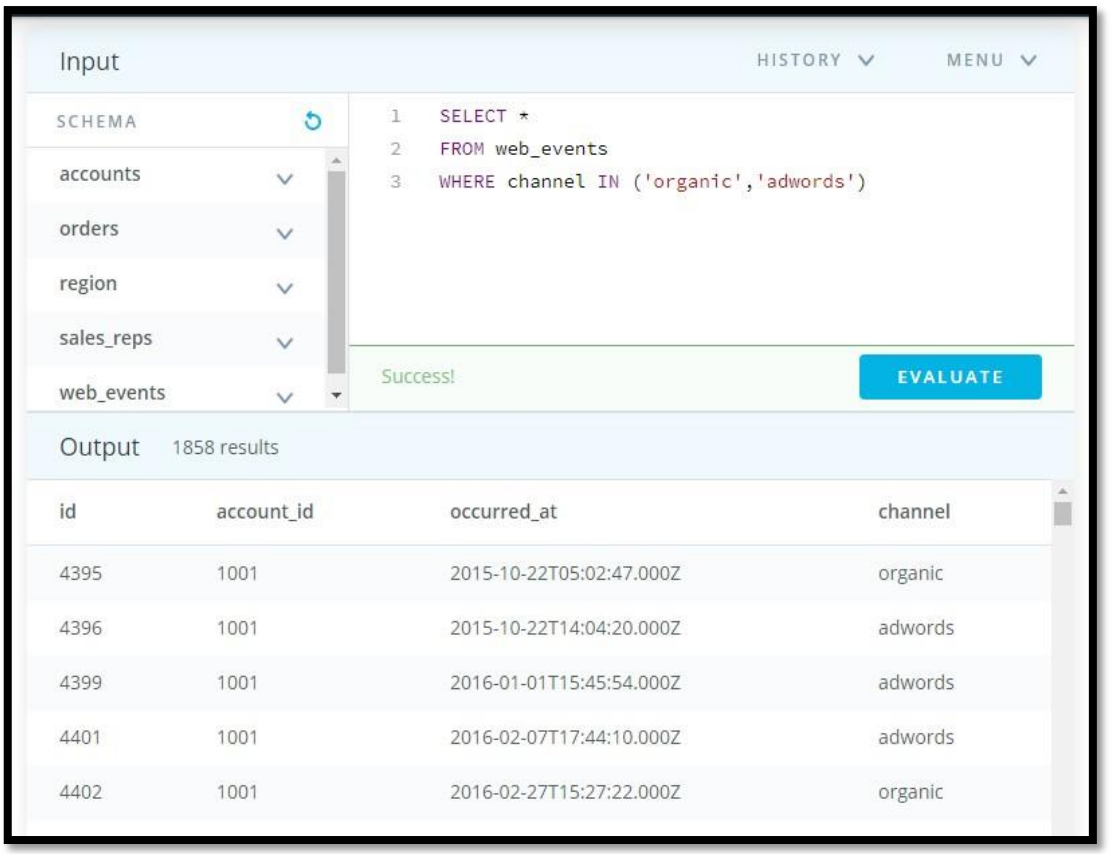
Quiz: IN

Questions using IN operator

1. Use the **accounts** table to find the account name, primary_poc, and sales_rep_id for Walmart, Target, and Nordstrom.



2. Use the **web_events** table to find all information regarding individuals who were contacted via the **channel** of organic or adwords.

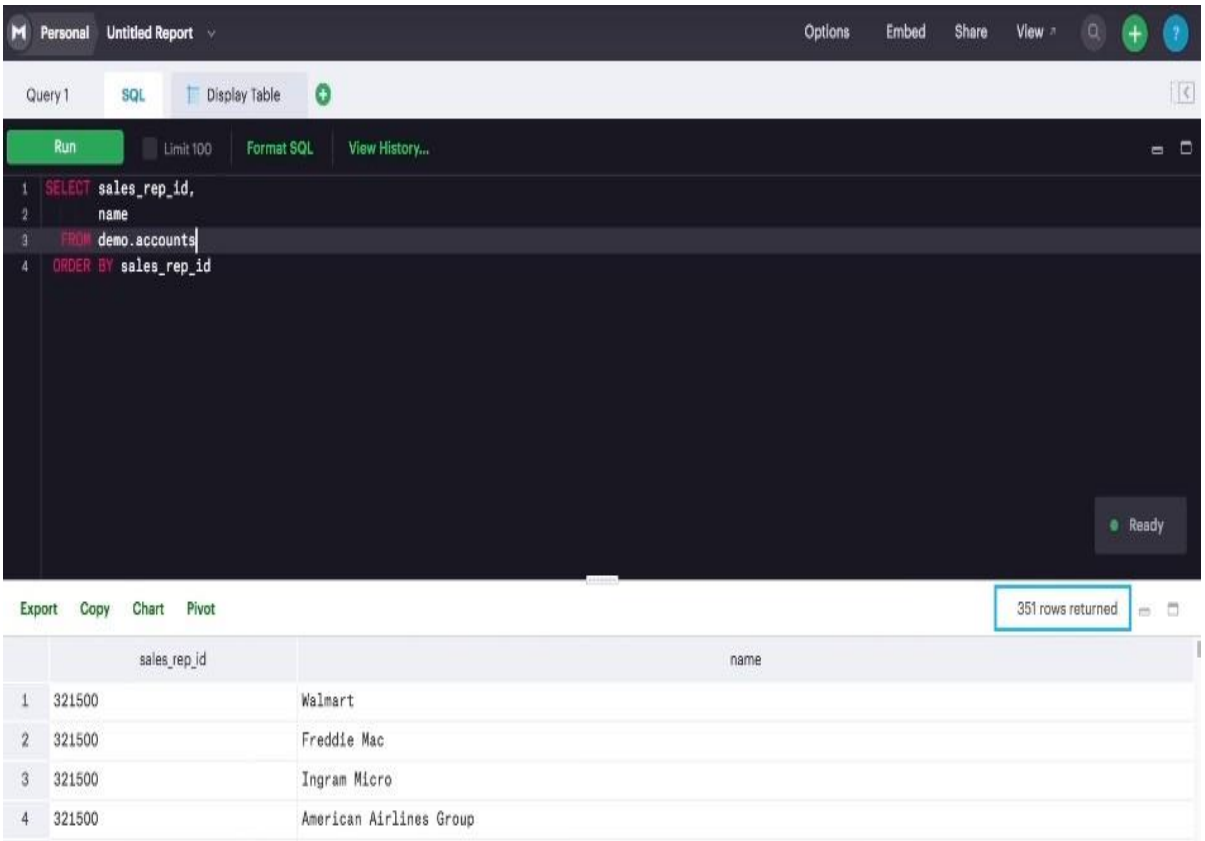


NOT

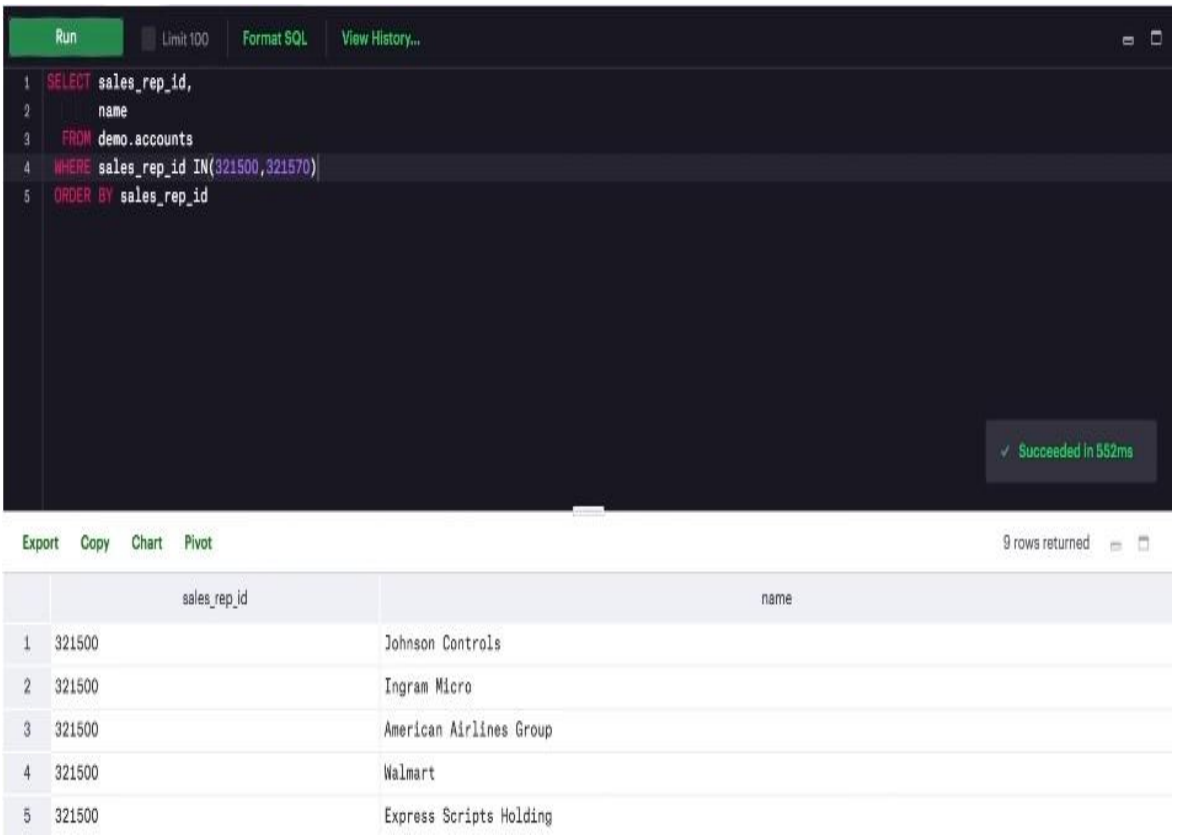
The **NOT** operator is an extremely useful operator for working with the previous two operators we introduced: **IN** and **LIKE**. By specifying **NOT LIKE** or **NOT IN**, we can grab all of the rows that do not meet a particular criteria.

Example:

Imagine you're a sales manager at Parch & Posey. You're considering promoting two of your top sales reps into management, but you need to figure out how you're going to divvy up all of their accounts among the other sales rep. to do this, let's first take a look at all of Parch & Posey accounts. You can see here that there are 351 in total.



Now let's look at the accounts that these two high performing sales reps are working. You can see here we filtered the list of accounts to just the two reps in question, nine accounts in total.

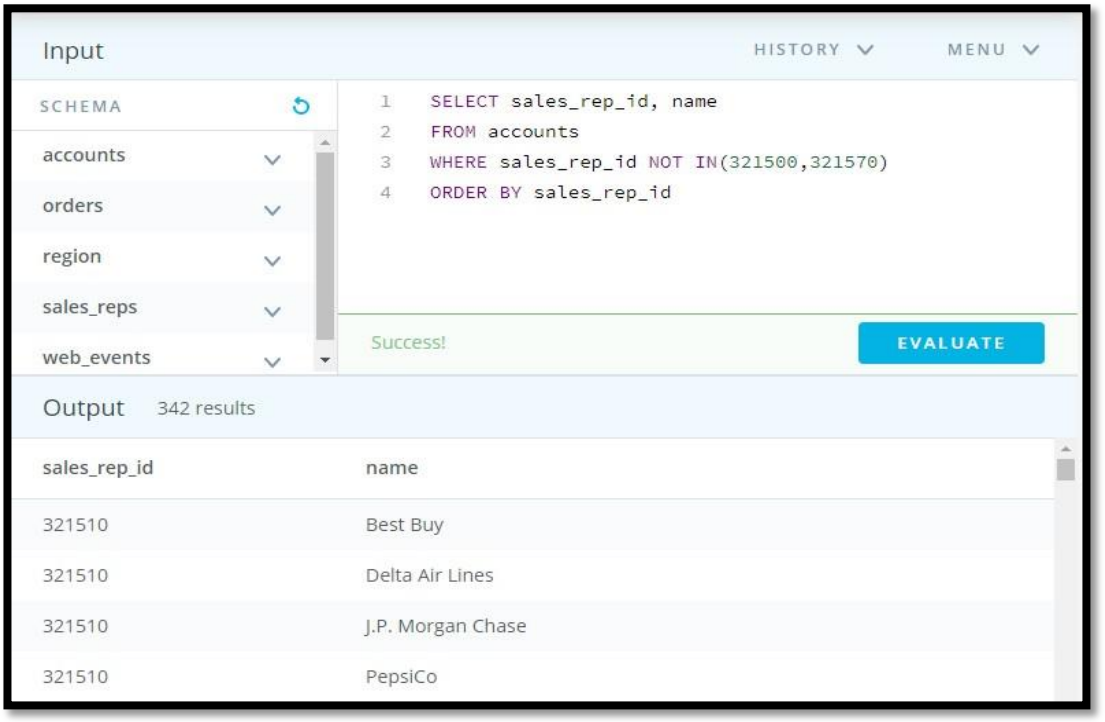


	sales_rep_id	name
1	321500	Johnson Controls
2	321500	Ingram Micro
3	321500	American Airlines Group
4	321500	Walmart
5	321500	Express Scripts Holding

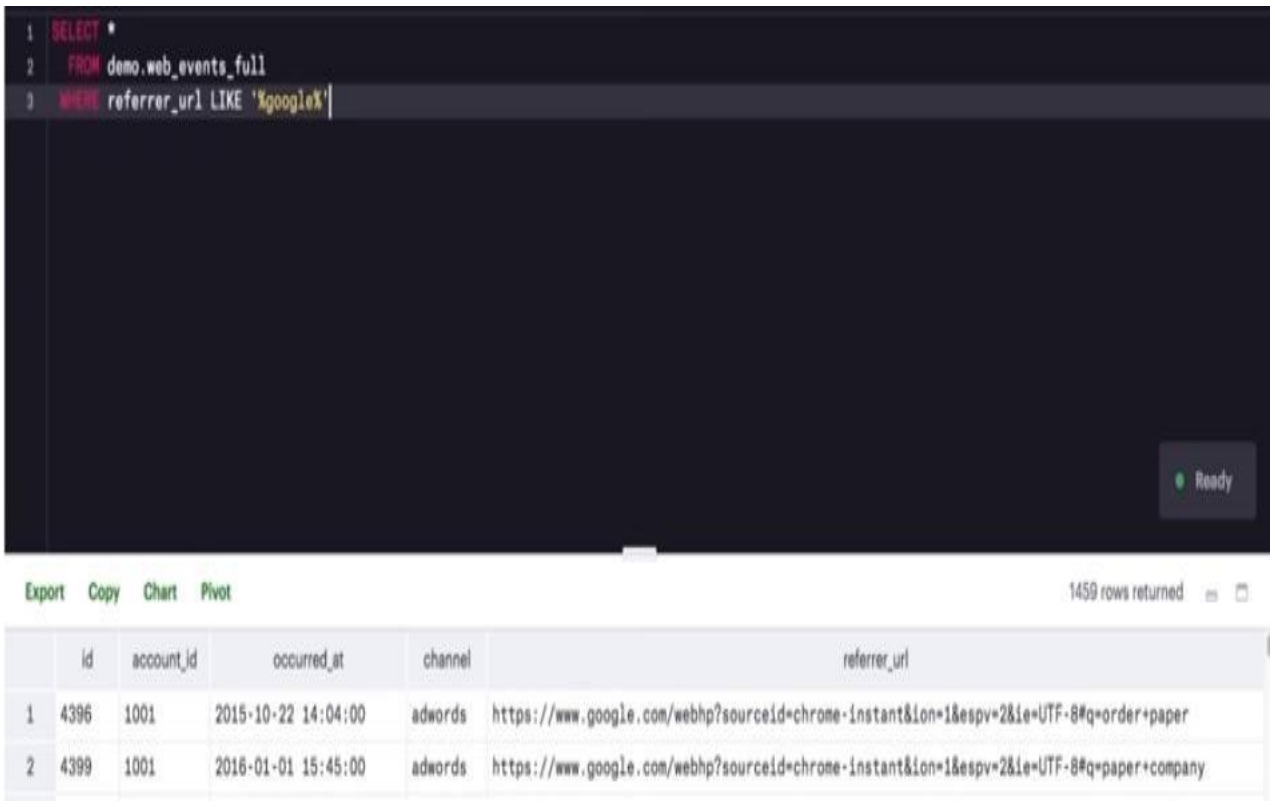
In deciding who to assign these accounts to, you might also want to consider which accounts the other sales reps are currently working. In another wards, you'd like to look at all the accounts that are not listed in this query we're looking at right now.

The NOT operator will allow you to do exactly this. By simply typing NOT before this logical statement, we can turn it into the inverse.

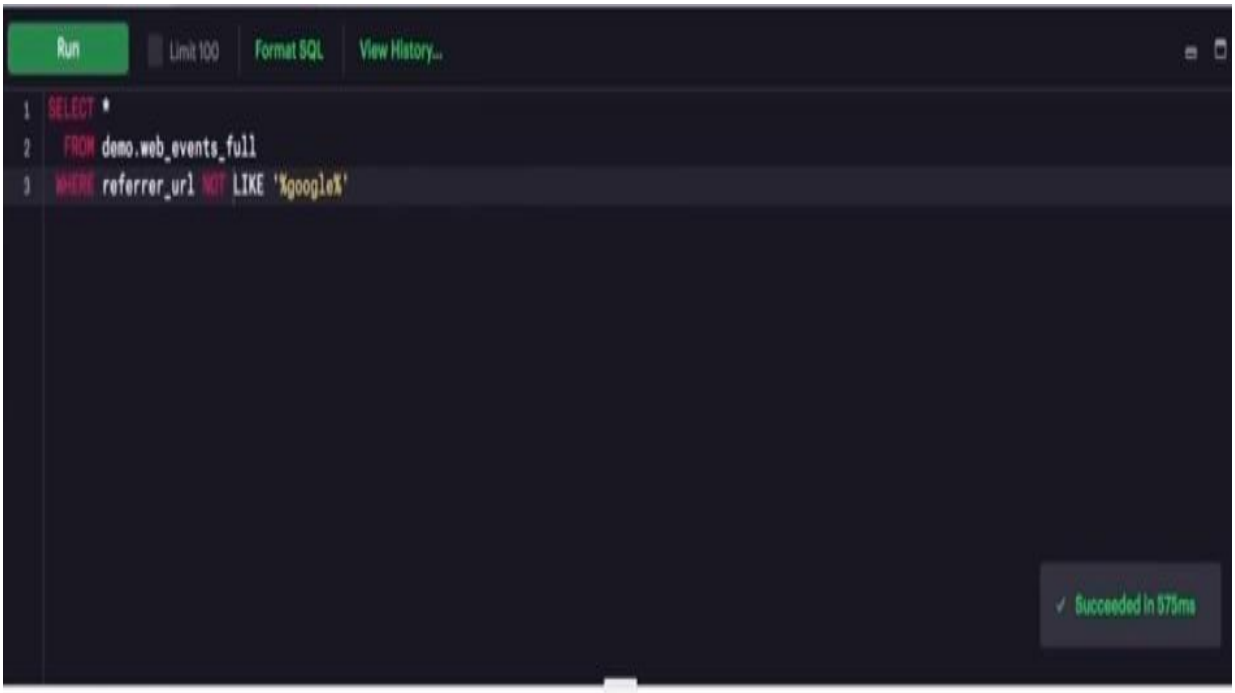
You can see that in this new result set, none of accounts that were previously shown are appearing. Of the 351 accounts that Parch & Posey sells to, the nine in the previous query are absent from this result set, so we see 342 results.



NOT also works with LIKE, the same way it works with IN. you can see here, we're looking at web traffic from Google in order to illustrate this.



By adding NOT, we can look at the inverse, all traffic from sets other than Google



Export Copy Chart Pivot 2316 rows returned

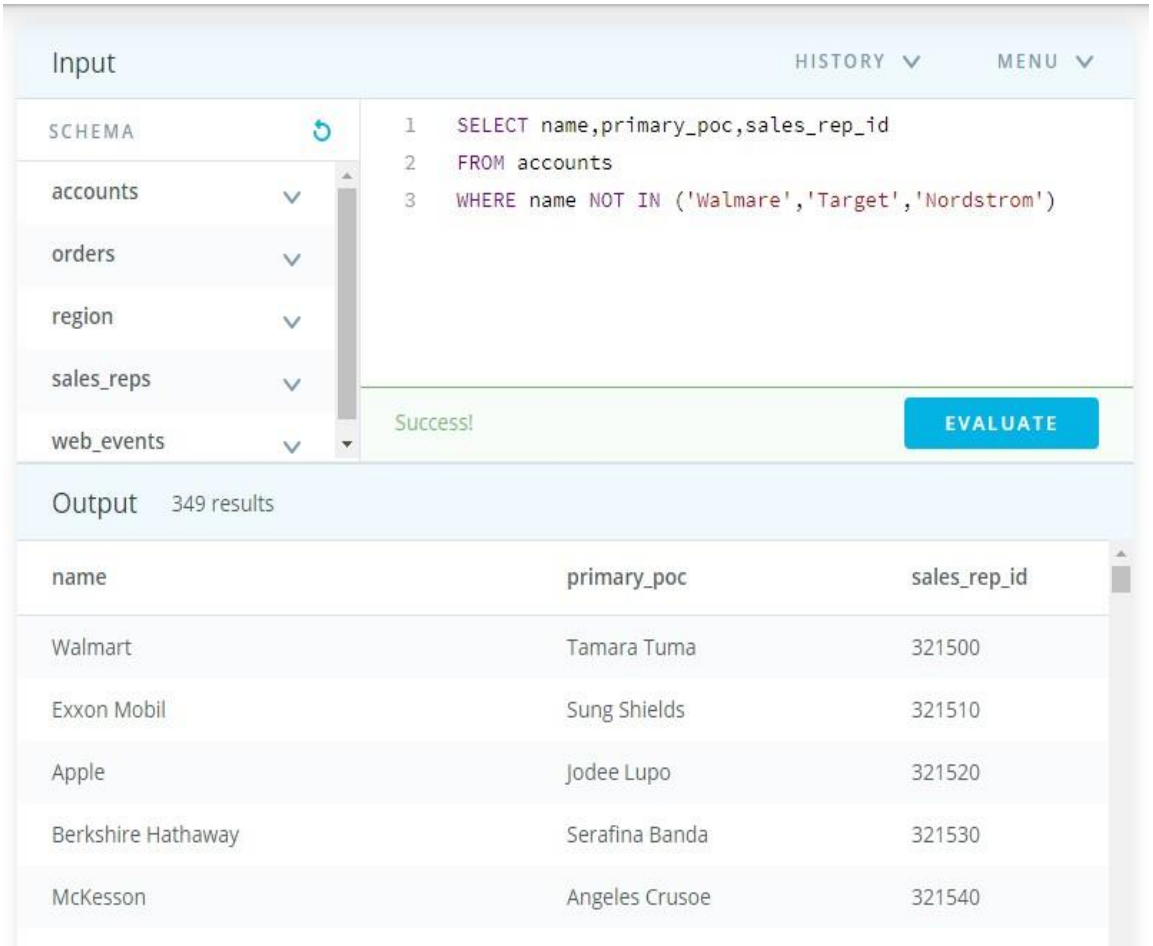
	id	account_id	occurred_at	channel	referrer_url
1	4407	1001	2016-05-07 02:28:00	banner	http://www.espn.com
2	4411	1001	2016-07-12 22:43:00	banner	http://www.espn.com
3	4413	1001	2016-07-26 21:08:00	banner	https://en.lichess.org
4	4423	1021	2015-12-26 02:15:00	banner	http://www.wellappointeddesk.com

Quiz: NOT

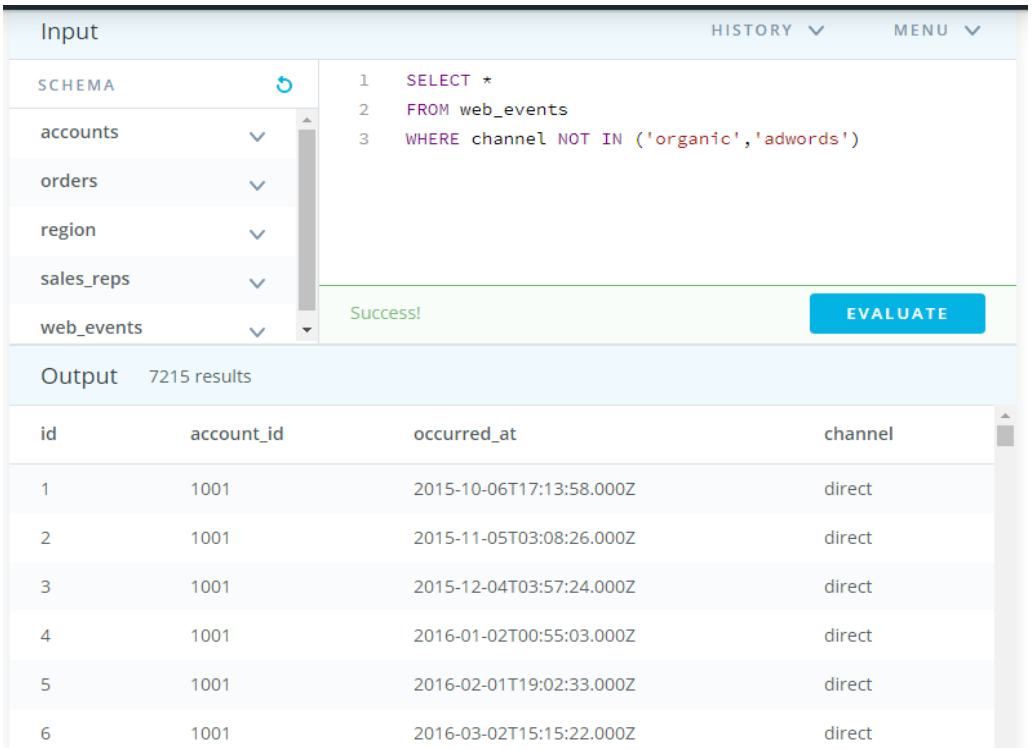
Questions using the NOT operator

We can pull all of the rows that were excluded from the queries in the previous two concepts with our new operator.

1. Use the **accounts** table to find the account name, primary poc, and sales rep id for all stores except Walmart, Target, and Nordstrom.

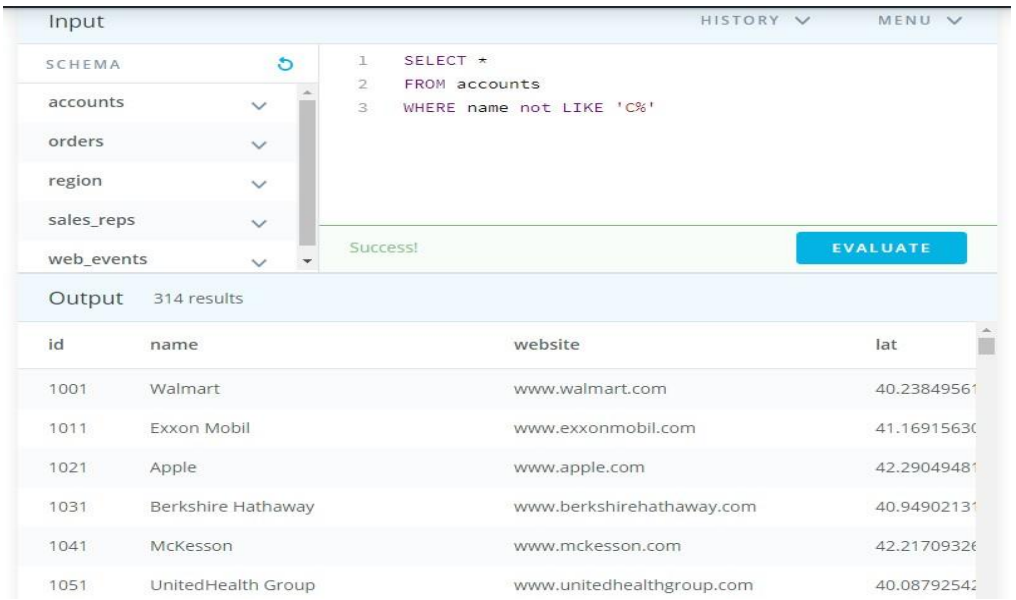


2. Use the **web_events** table to find all information regarding individuals who were contacted via any method except using organic or adwords methods.

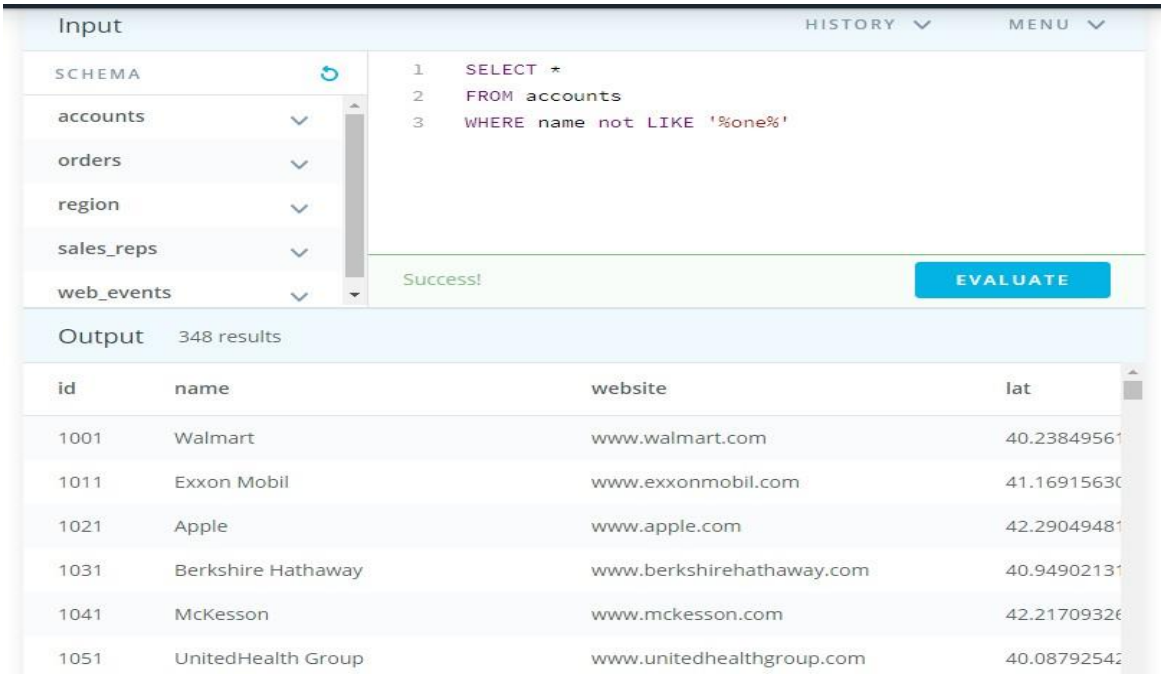


Use the **accounts** table to find:

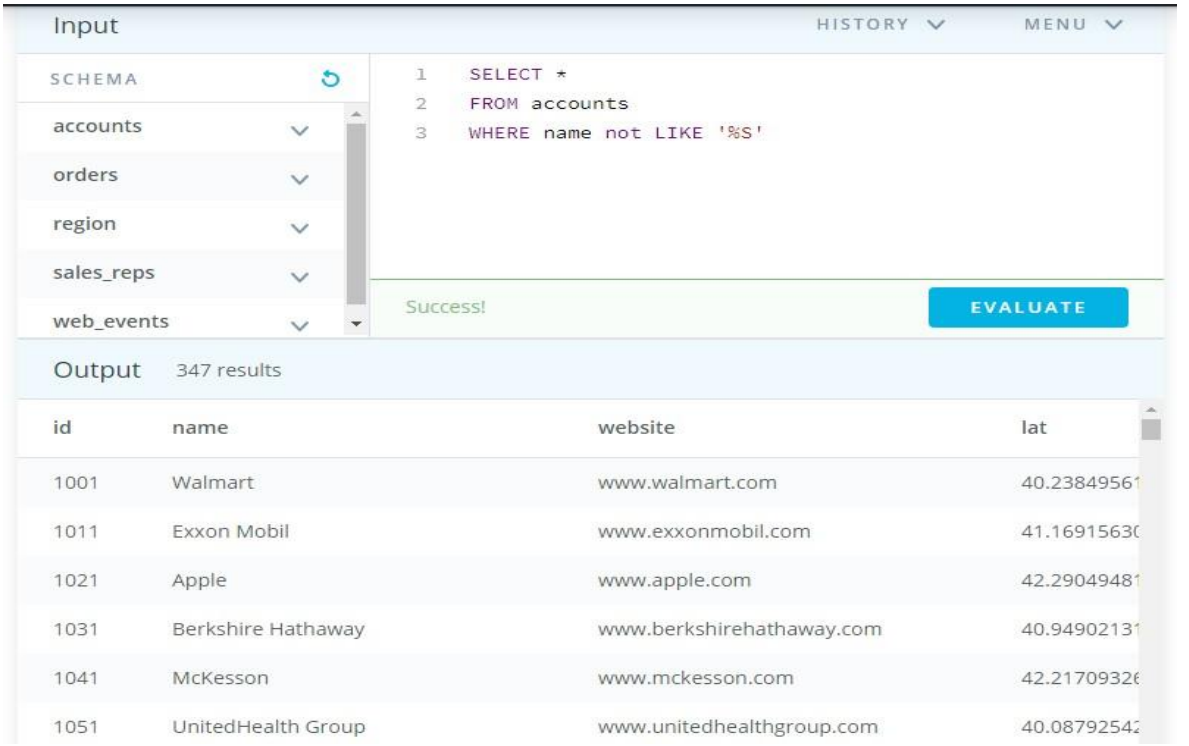
1. All the companies whose names do not start with 'C'.



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



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



AND and BETWEEN

The **AND** operator is used within a **WHERE** statement to consider more than one logical clause at a time. Each time you link a new statement with an **AND**, you will need to specify the column you are interested in looking at. You may link as many statements as you would like to consider at the same time. This operator works with all of the operations we have seen so far including arithmetic operators (+, *, -, /). **LIKE**, **IN**, and **NOT** logic can also be linked together using the **AND** operator.

BETWEEN Operator

Sometimes we can make a cleaner statement using **BETWEEN** than we can using **AND**. Particularly this is true when we are using the same column for different parts of our **AND** statement. In the previous lesson, we probably should have used **BETWEEN**.

Instead of writing :

```
WHERE column >= 6 AND column <= 10
```

we can instead write, equivalently:

```
WHERE column BETWEEN 6 AND 10
```

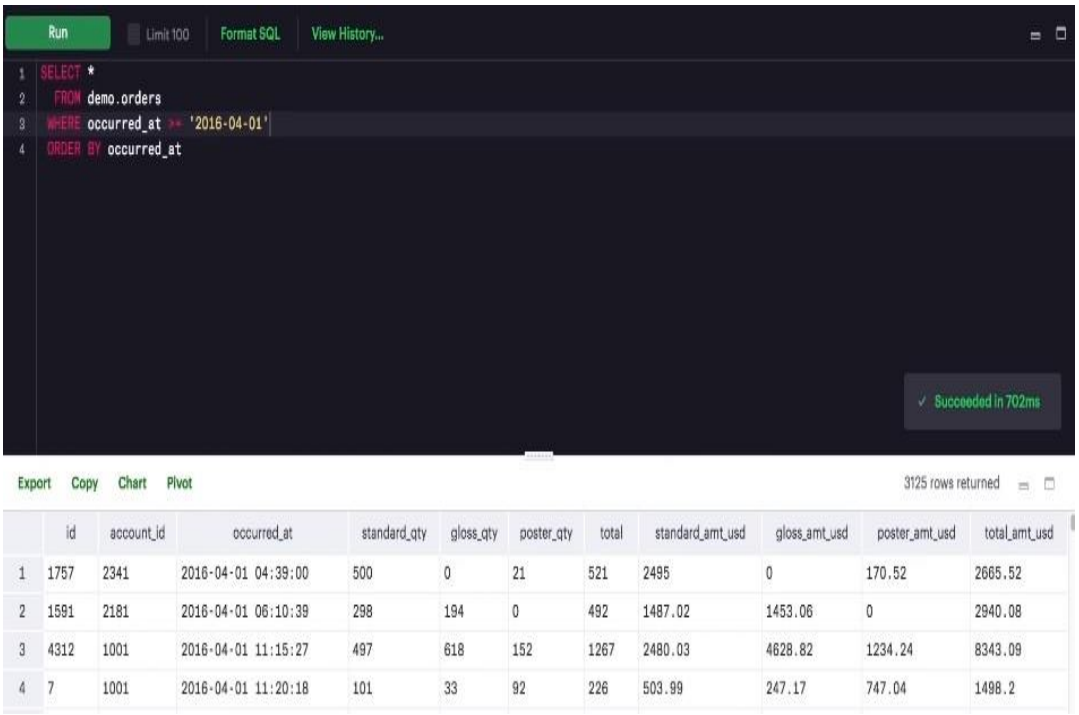
Example:

Imagine yourself as a sales manager at Parch & Posey, you want to see which of your customers bought paper a while ago and might be due to make a purchase soon.

Let's assume it's January 2017. You'd probably want to look up customers who made purchases somewhere between three &nine months prior, or between April and October of 2016.

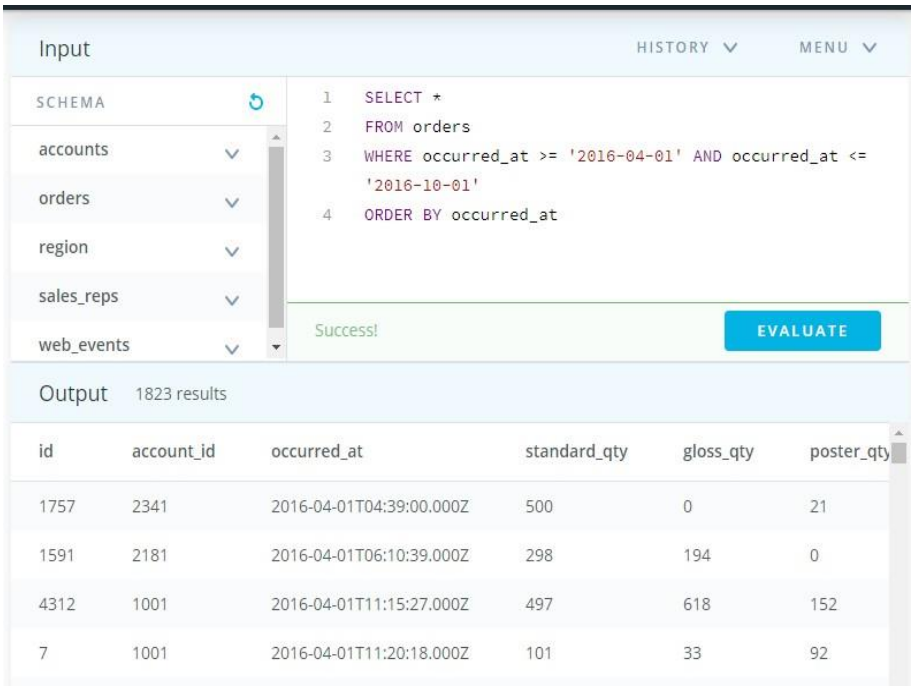
In order to do this, you'll need to filter based on multiple criteria using AND.

Let's dig into this example by pulling orders that took place after April 1st 2016.



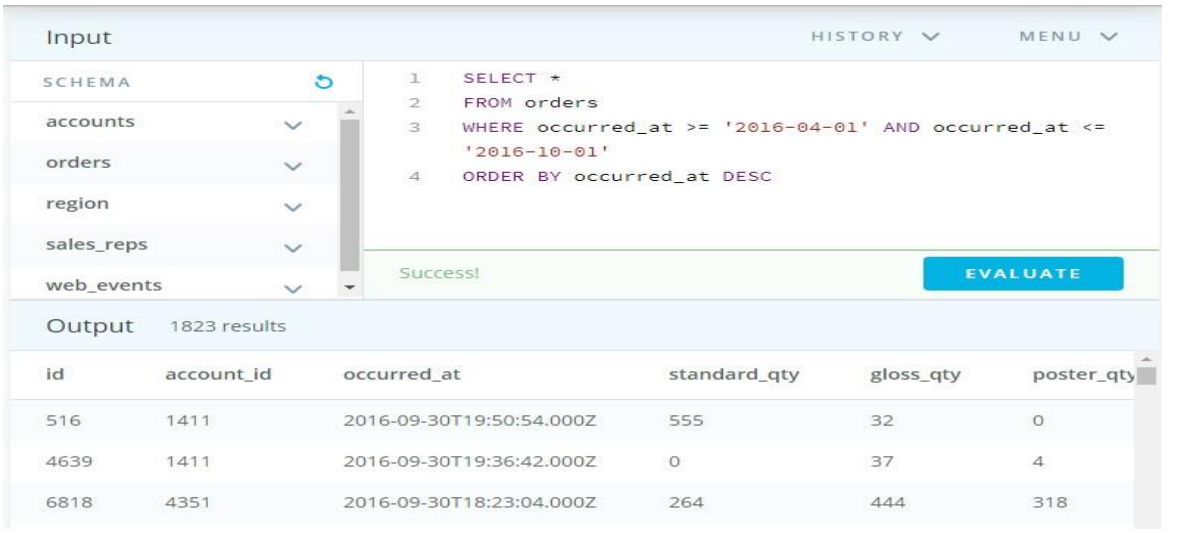
	id	account_id	occurred_at	standard_qty	gloss_qty	poster_qty	total	standard_amt_usd	gloss_amt_usd	poster_amt_usd	total_amt_usd
1	1757	2341	2016-04-01 04:39:00	500	0	21	521	2495	0	170.52	2665.52
2	1591	2181	2016-04-01 06:10:39	298	194	0	492	1487.02	1453.06	0	2940.08
3	4312	1001	2016-04-01 11:15:27	497	618	152	1267	2480.03	4628.82	1234.24	8343.09
4	7	1001	2016-04-01 11:20:18	101	33	92	226	503.99	247.17	747.04	1498.2

Next, let's exclude orders more recent than October 1st 2016. We can add this to our WHERE clause using AND.



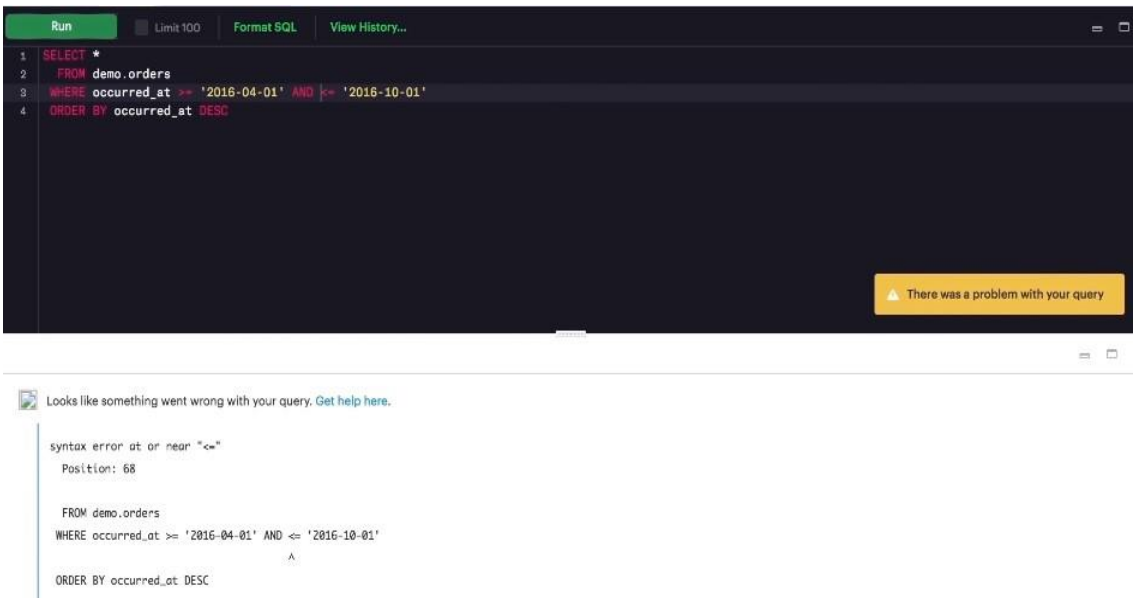
id	account_id	occurred_at	standard_qty	gloss_qty	poster_qty
1757	2341	2016-04-01T04:39:00.000Z	500	0	21
1591	2181	2016-04-01T06:10:39.000Z	298	194	0
4312	1001	2016-04-01T11:15:27.000Z	497	618	152
7	1001	2016-04-01T11:20:18.000Z	101	33	92

Let's wrap this up by reordering these result to confirm that orders after October 1st were excluded.



You can see that all orders after October 1st , 2016, have not been filtered out. One important thing to note here is that the AND operator allows you to run two complete logical statements. When this query runs, it evaluates a row to see if occurred_at is greater than or equal to April 1st, 2016. Then evaluates the same row to see of occurred at is less than or equal to October 1st 2016. These need to be independent logical statements, even though they're both operating on the same column name.

We can't just type where occurred_at is greater than or equal to April 1st and less than or equal to October 1st . less than or equal to October 1st isn't a logical statement that can be evaluated as

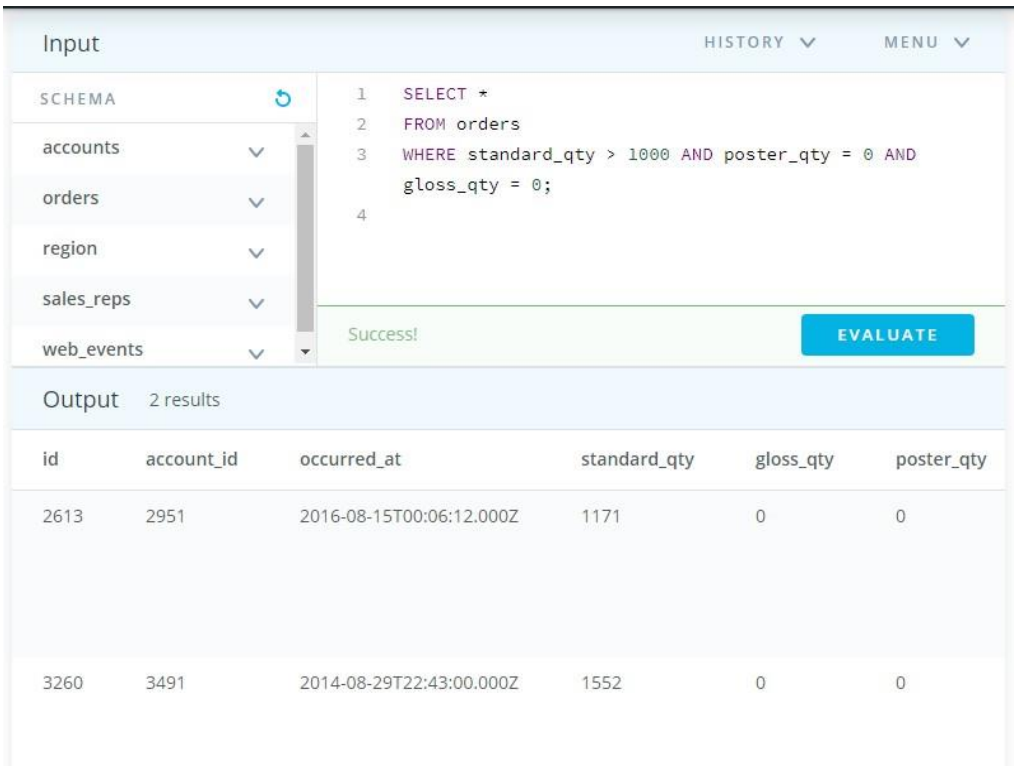


true or false, so the query will return an error.

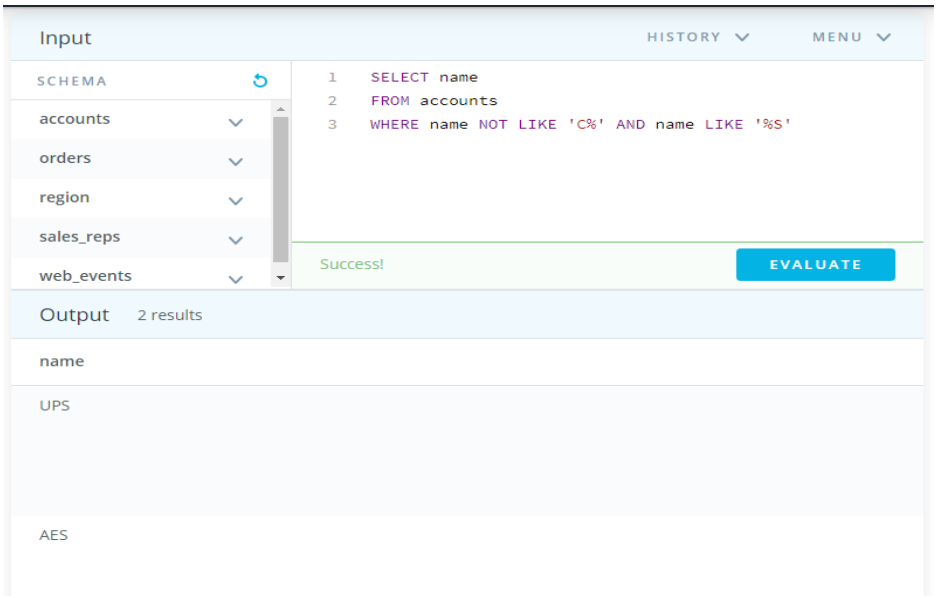
Quiz: AND and BETWEEN

Questions using AND and BETWEEN operators

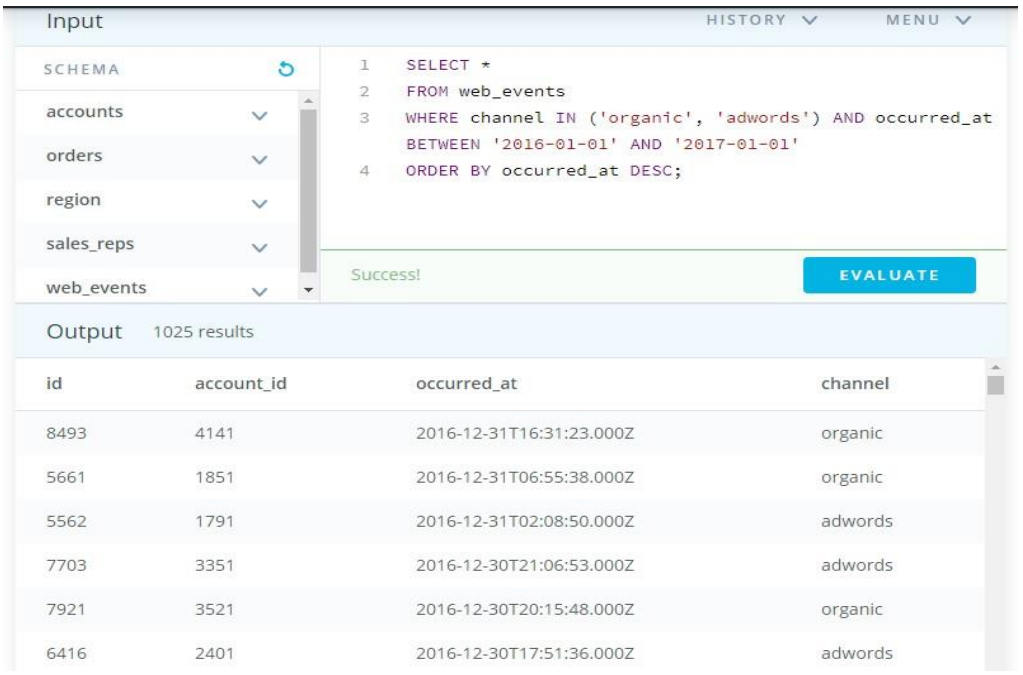
1. Write a query that returns all the **orders** where the standard_qty is over 1000, the poster_qty is 0, and the gloss_qty is 0.



2. Using the **accounts** table find all the companies whose names do not start with 'C' and end with 's'.



3. Use the **web_events** table to find all information regarding individuals who were contacted via organic or adwords and started their account at any point in 2016 sorted from newest to oldest.



OR

Similar to the **AND** operator, the **OR** operator can combine multiple statements. Each time you link a new statement with an **OR**, you will need to specify the column you are interested in looking at. You may link as many statements as you would like to consider at the same time. This operator works with all of the operations we have seen so far including arithmetic operators (+, *, -, /), **LIKE**, **IN**, **NOT**, **AND**, and **BETWEEN** logic can all be linked together using the **OR** operator.

When combining multiple of these operations, we frequently might need to use parentheses to assure that logic we want to perform is being executed correctly. The example below shows of these situations.

Example :

Imagine yourself a sales manager at Porch & Posey. You've determined that another great way to expand revenue is to sell new types of paper to existing to customers. In order to figure out the best prospects, you'd like to see existing customers whose orders omitted some type of paper.

Orders different types of paper are marked in three separate columns in the orders table.

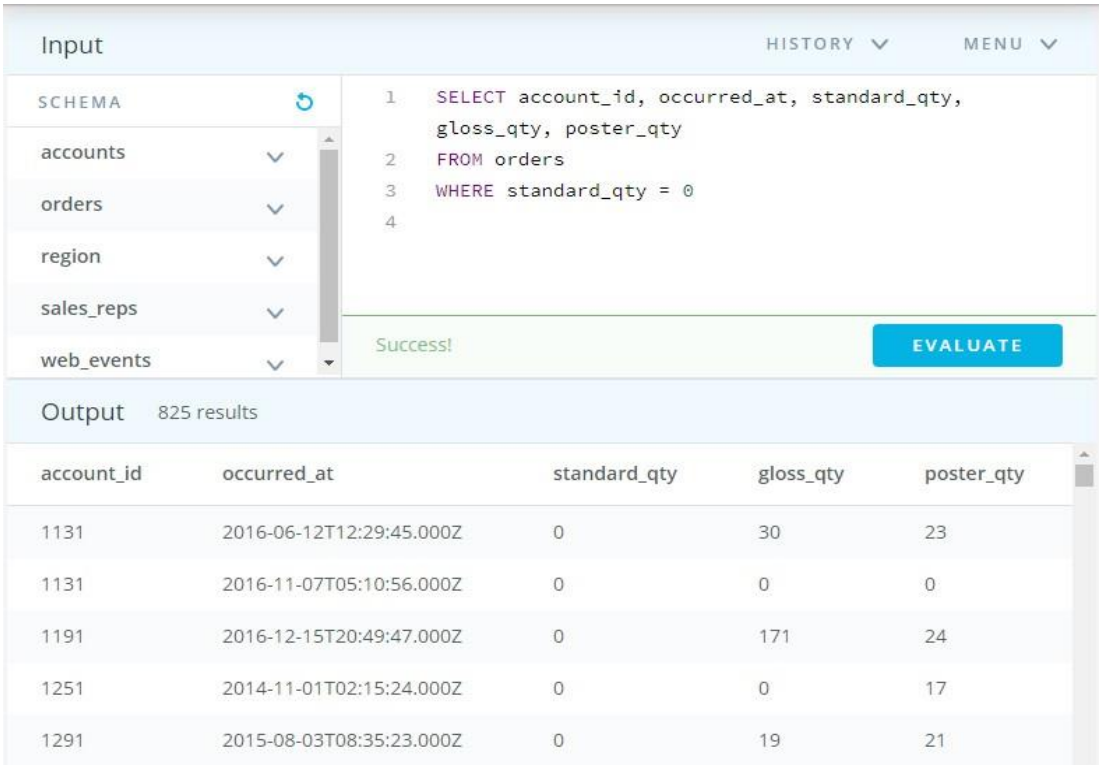
So you'll need to write a query that makes logical comparisons on each of those columns. You can see that most orders are for all three types of paper.

Now let's start to build in the logic to fined orders where one paper type might not have been ordered. You can do that using OR.

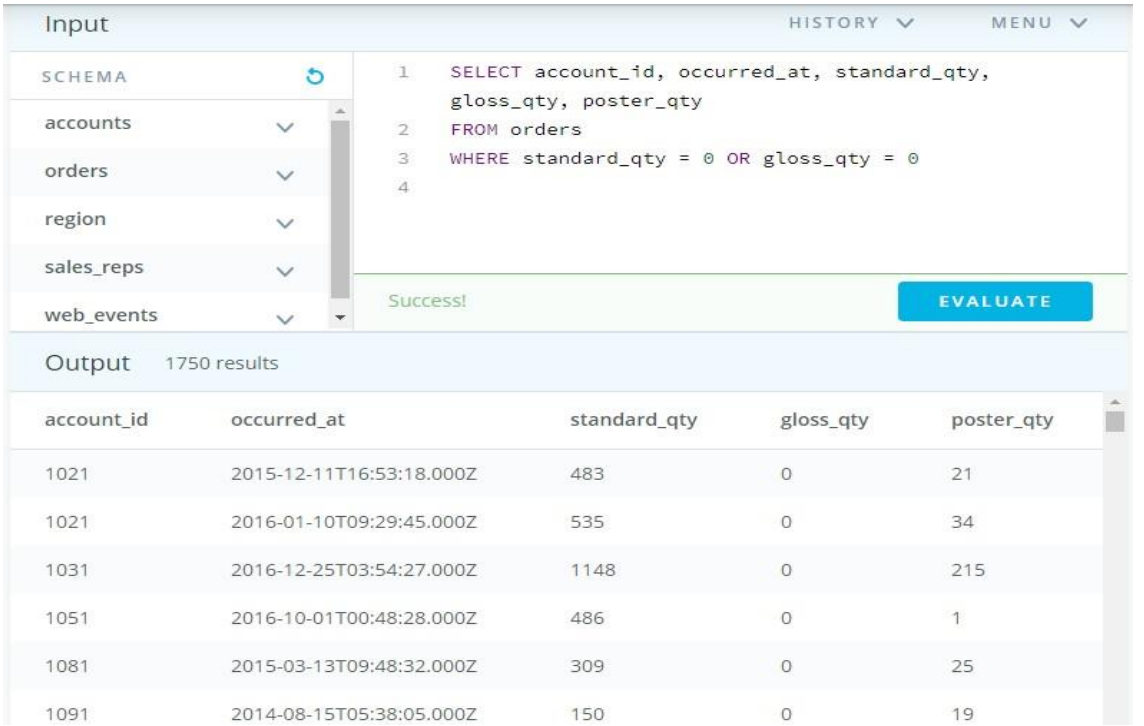
OR is a logical operator in SQL that allows you to select rows that satisfy either of two conditions.

It work similarly to AND which selects the rows that satisfy both of two conditions.

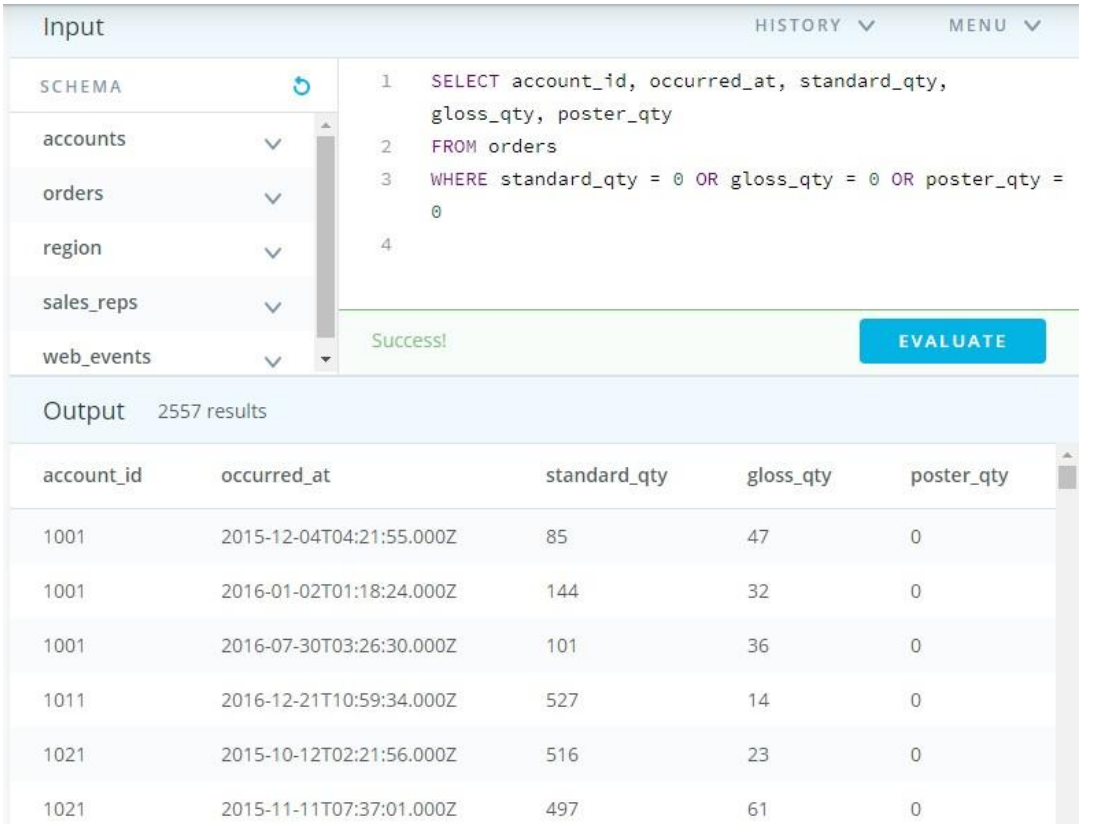
Let's build this out by first selecting only orders for which standard paper was not included.



You can see that broduces 825 results. Now let's add another paper type using OR.

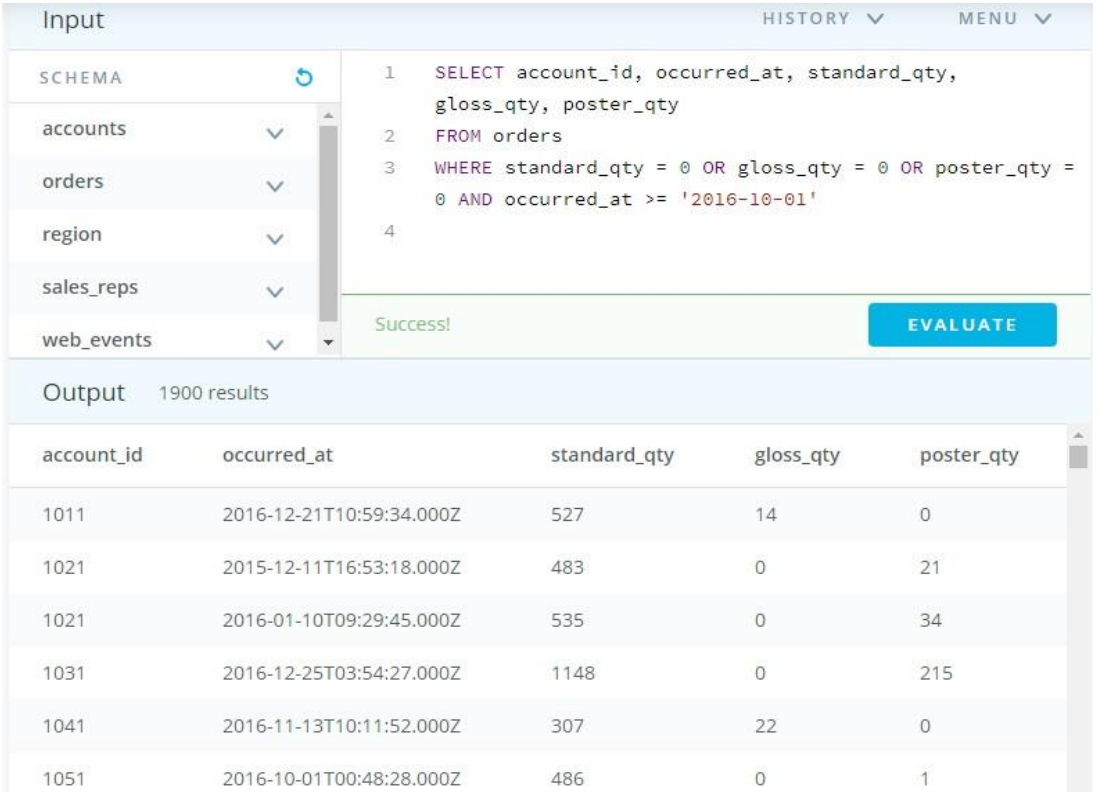


You can see that this result set is largest than the previous one. That's because we've made this filter more inclusive. It will now return results where either one of these two logical statements resolves is true. By adding more logical statement, will arrive at a query that returns all orders where some type of paper was omitted.



account_id	occurred_at	standard_qty	gloss_qty	poster_qty
1001	2015-12-04T04:21:55.000Z	85	47	0
1001	2016-01-02T01:18:24.000Z	144	32	0
1001	2016-07-30T03:26:30.000Z	101	36	0
1011	2016-12-21T10:59:34.000Z	527	14	0
1021	2015-10-12T02:21:56.000Z	516	23	0
1021	2015-11-11T07:37:01.000Z	497	61	0

OR can be combined with other operators for using parentheses. Going back to our example where we were looking for orders that did not contain all three types of paper, we can make that even more targeted. The best prospects might be folks who omitted some type of paper, and also ordered recently. Let's keep our filters from that previous query, but also require that the order have occurred after October 1st 2016.

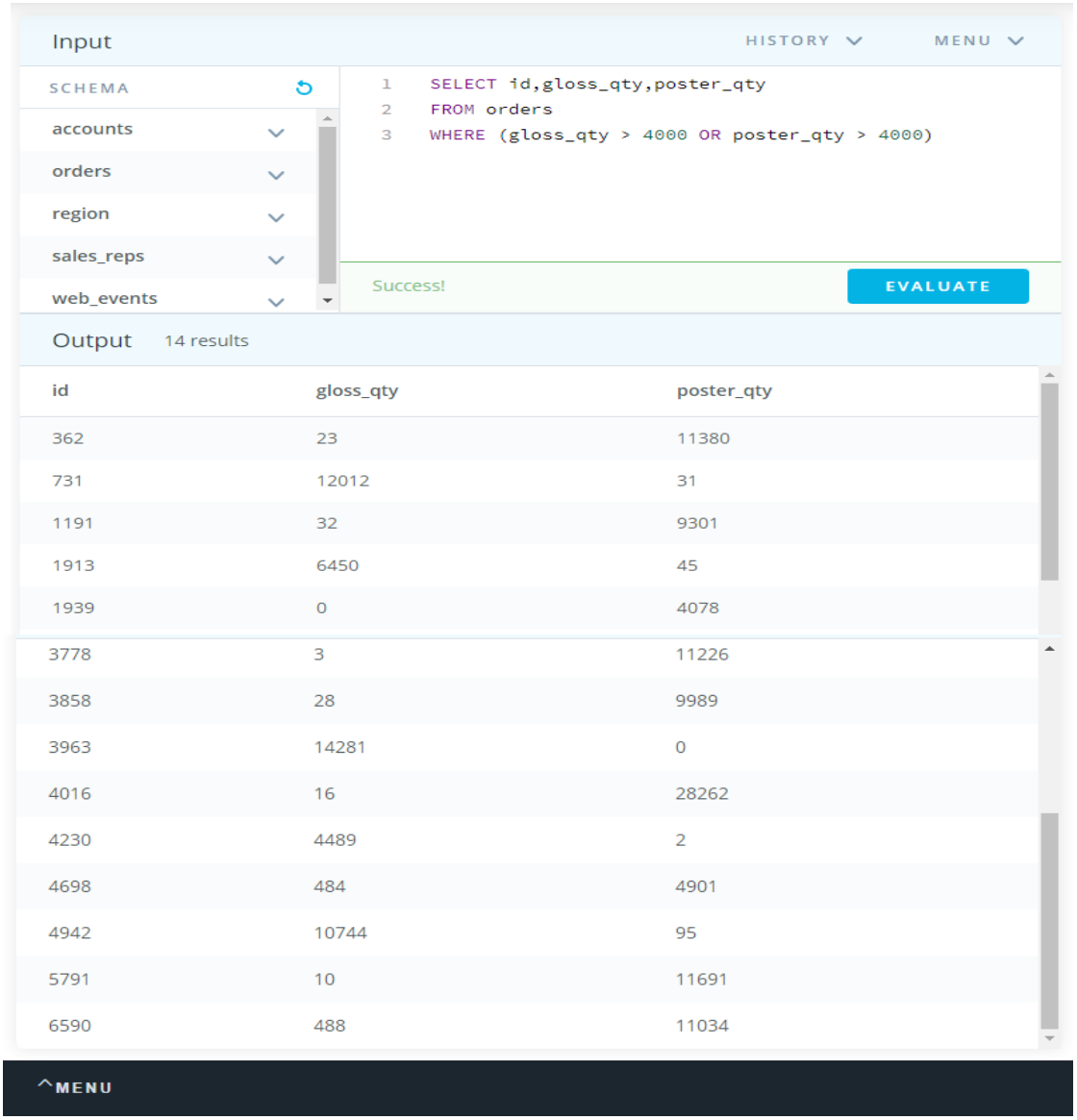


You'll notice that this time we wrap the **OR** statements in parentheses. You can thing of the parentheses as turning three logical statements into one big logical statement, that will resolve to true or false. As long as one of these three statements is true, the entire block in the parentheses is considered to be true. The next logical statement after **AND**, will also be evaluated as true or false. And the query will only return rows for which the entire set of **OR** statements is true, and the data filter is true. Looking at this first row, you can see that **" occurred_at "** is after October 1st 2016, and **" poster_qty "** is zero, fulfilling the logical statement created by the parantheses.

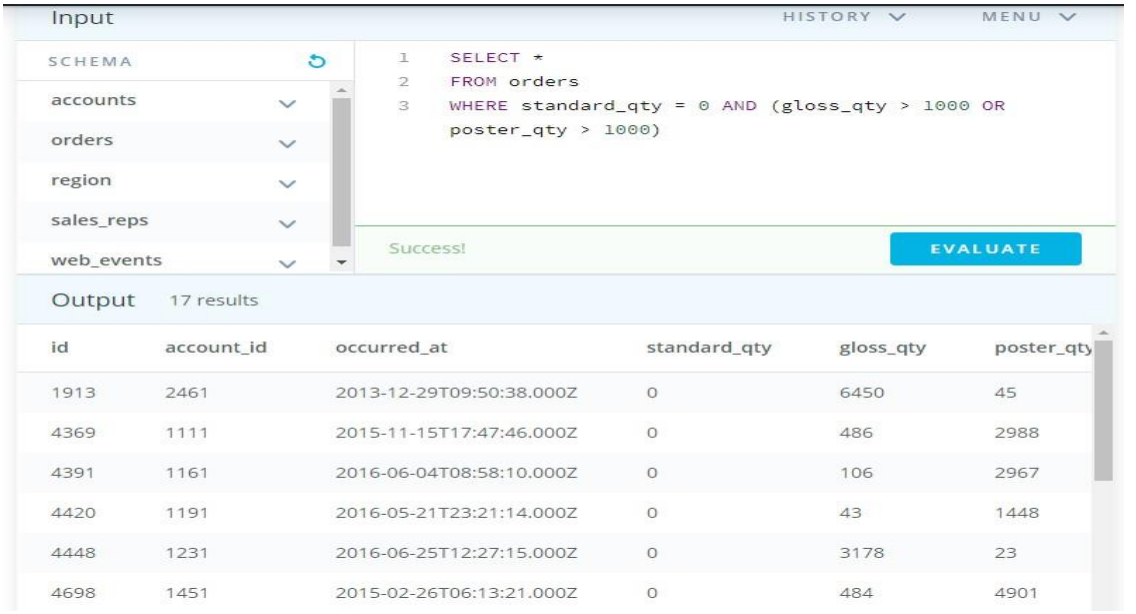
Quiz: OR

Questions using the OR operator

1. Find list of **orders** ids where either gloss_qty or poster_qty is greater than 4000. Only include the id field in the resulting table.

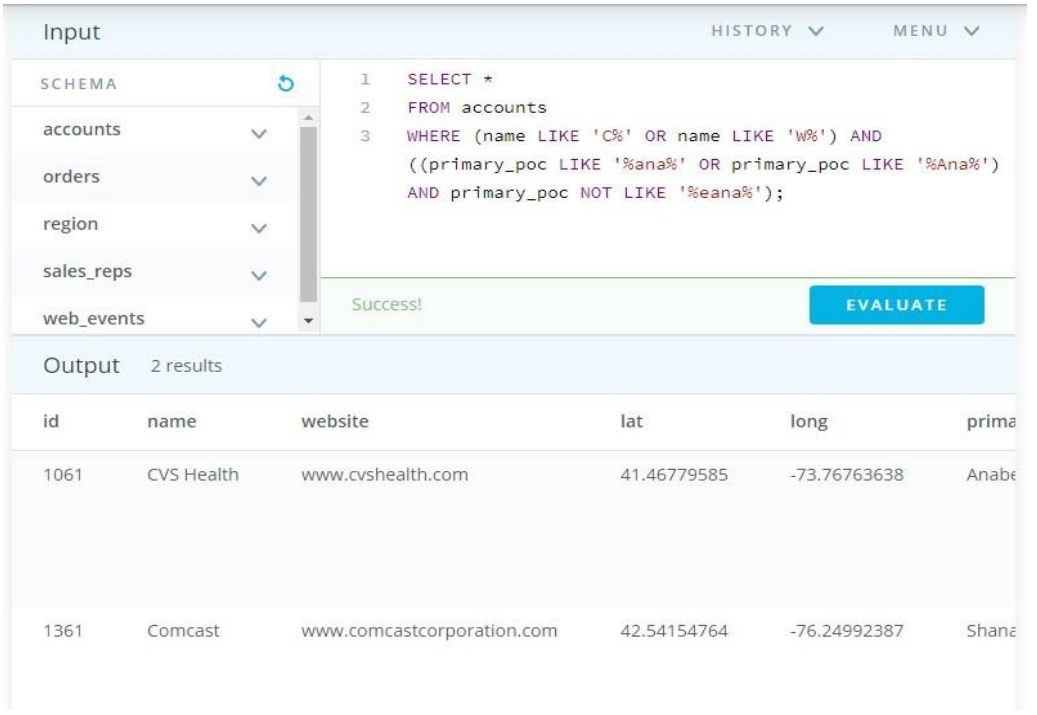


2. Write a query that returns a list of **orders** where the standard_qty is zero and either the gloss_qty or poster_qty is over 1000.



id	account_id	occurred_at	standard_qty	gloss_qty	poster_qty
1913	2461	2013-12-29T09:50:38.000Z	0	6450	45
4369	1111	2015-11-15T17:47:46.000Z	0	486	2988
4391	1161	2016-06-04T08:58:10.000Z	0	106	2967
4420	1191	2016-05-21T23:21:14.000Z	0	43	1448
4448	1231	2016-06-25T12:27:15.000Z	0	3178	23
4698	1451	2015-02-26T06:13:21.000Z	0	484	4901

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



id	name	website	lat	long	prima
1061	CVS Health	www.cvshealth.com	41.46779585	-73.76763638	Anabe
1361	Comcast	www.comcastcorporation.com	42.54154764	-76.24992387	Shana

Recap & Looking Ahead

Commands

You have already learned a lot about writing code in SQL! Let's take a moment to recap all that we have covered before moving on:

Statement	How to Use It	Other Details
SELECT	SELECT **Col1**, **Col2**, ...	Provide the columns you want
FROM	FROM **Table**	Provide the table where the columns exist
LIMIT	LIMIT **10**	Limits based number of rows returned
ORDER BY	ORDER BY **Col**	Orders table based on the column. Used with **DESC**.
WHERE	WHERE **Col > 5**	A conditional statement to filter your results
LIKE	WHERE **Col LIKE '%me%'**	Only pulls rows where column has 'me' within the text
IN	WHERE **Col IN ('Y', 'N')**	A filter for only rows with column of 'Y' or 'N'
NOT	WHERE **Col NOT IN ('Y', 'N')**	**NOT** is frequently used with **LIKE** and **IN**
AND	WHERE **Col1 > 5 AND Col2 < 3**	Filter rows where two or more conditions must be true
OR	WHERE **Col1 > 5 OR Col2 < 3**	Filter rows where at least one condition must be true
BETWEEN	WHERE **Col BETWEEN 3 AND 5**	Often easier syntax than using an **AND**

Other Tips

Though SQL is **not case sensitive** (it doesn't care if you write your statements as all uppercase or lowercase), we discussed some best practices. **The order of the key words does matter!** Using what you know so far, **you will want to write your statements as:**

```
SELECT col1, col2
FROM table1
WHERE col3 > 5 AND col4 LIKE '%os%'
ORDER BY col5
LIMIT 10;
```

Notice, you can retrieve different columns than those being used in the **ORDER BY** and **WHERE** statements. Assuming all of these column names existed in this way (col1, col2, col3, col4, col5) within a table called table1, this query would run just fine.

Looking Ahead

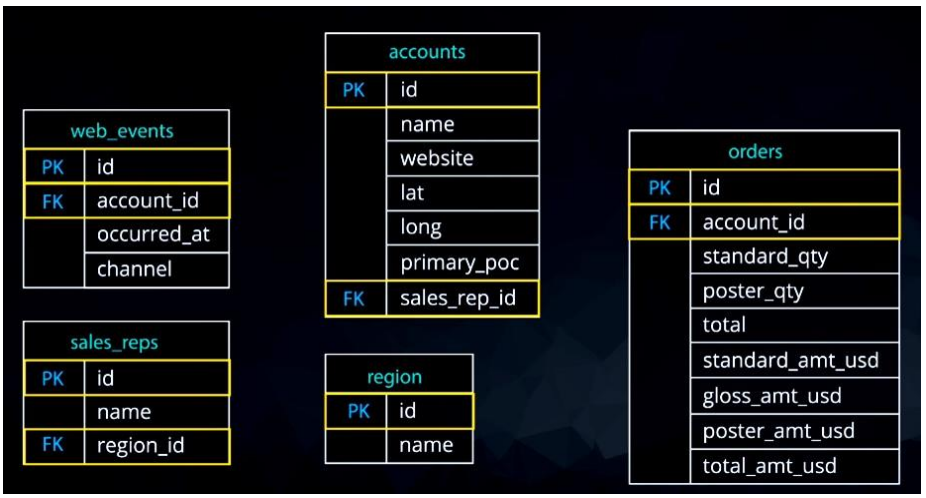
In the next lesson, you will be learning about **JOIN**s. This is the real secret (well not really a secret) behind the success of SQL as a language. **JOIN**s allow us to combine multiple tables together. All of the operations we learned here will still be important moving forward, but we will be able to answer much more complex questions by combining information from multiple tables! You have already mastered so much - potentially writing your first code ever, but it is about to get so much better!

LESSON 27

Motivation

Up to this point, we've only been working with one table at a time. But the real power of SQL comes from working with data across multiple tables at once.

The term relational database refers to the fact that tables within it relate to one another. they contain common identifiers that allow information from multiple tables to be easily combined.



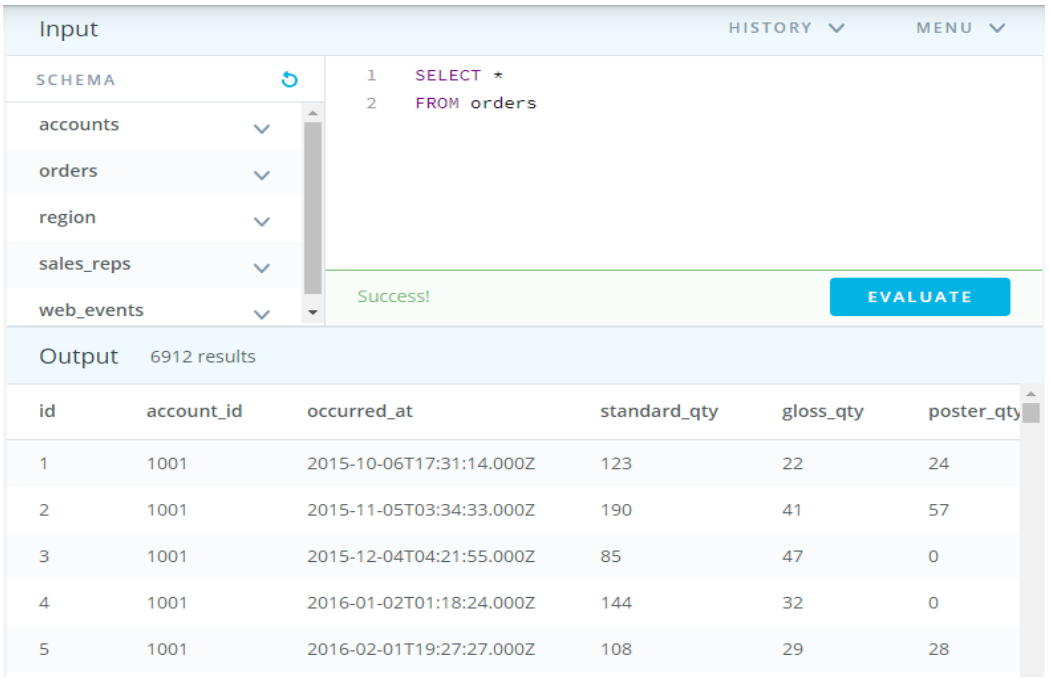
Keeping all of the company's data, from purchasing transactions, to employee job satisfaction, to inventory in a single Excel dataset doesn't make a ton of sense. The table with hold a ton of information and it'd be hard to determine a row call and structure for so many different types of data.

Databases give us the flexibility to keep things organized neatly in their own tables, so that they're easy to find and work with while also allowing us to combine tables as needed to solve problems that require several types of data.

In this lesson, we're going to take a look at how to leverage SQL to link tables together. You're about to see why SQL is one of the most popular environments for working with data as we learn who to write what are known as joints.

Why Would We Want to Split Data Into Separate Tables?

To understand what JOINs are and why they're helpful, let's think about Parch & Posey's orders table. looking back at this table, you'll notice that none of the orders say the name of the customer.



Instead, the table refers to customers by numerical values in the **account_id** column, we'll need to JOIN another table in order to connect this data to names. but first, why isn't the customer's name in this table in the first place?

There are a few reasons that someone might have made the decision to separate orders from the information about the customers placing those orders.

There are several reasons why relation databases are like this.

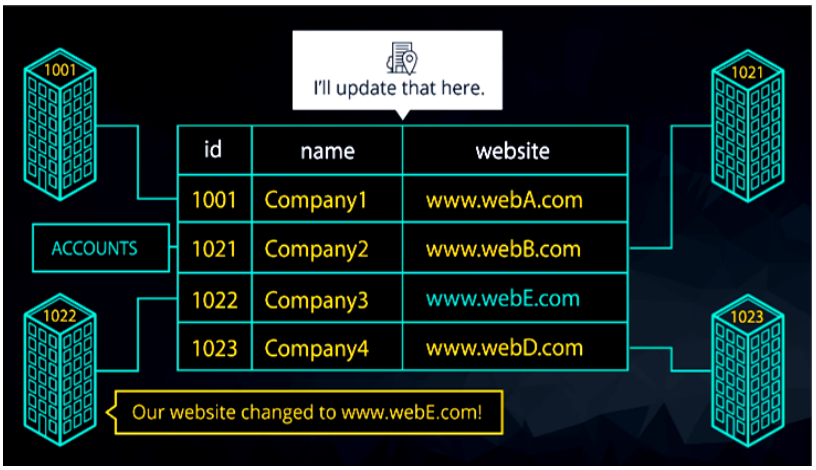
let's focus on two of the most important once:

first, orders and accounts are different types of objects and will be easier to organize if kept separate.

Second, this multi-table structure allows queries to execute more quickly.

let's look at each of these points more closely. The accounts and orders tables store fundamentally different types of objects.

Parch & Posey probably only wants one account per company, and they wanted to be up-to-date with the latest information.



Orders, on the other hand, are likely to stay the same once they are entered, and certainly, once they're filled. A given customer might have multiple orders, and rather than change a past order, Parch & Posey might just add a new one. Because these objects operate differently, it makes sense for them to live in different tables.

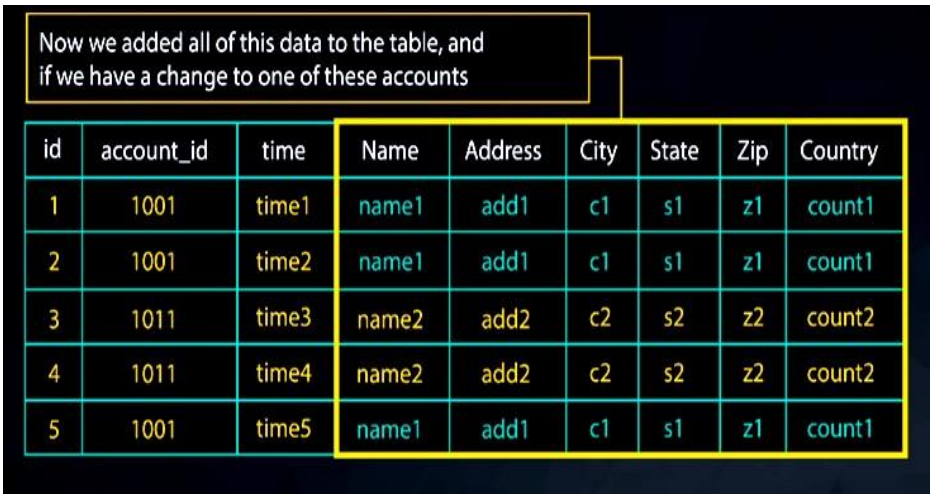
Another reason accounts and orders might be stored separately has to do with database can modify data. When you write a query, its execution speed depends on the amount of data you're asking the database to read and the number and type of calculations you're asking it to make.

Imagine a world where the account names and addresses are added into the orders table.

This means, the table would have six additional columns:

One for account name, and five that make up the street address, city, state, zip code, and country.

Let's say, customer change address. In this world, the five address columns need to be update retroactively for every single order. This means five updates times the number of orders that they've made.



id	account_id	time	Name	Address	City	State	Zip	Country
1	1001	time1	name1	add1	c1	s1	z1	count1
2	1001	time2	name1	add1	c1	s1	z1	count1
3	1011	time3	name2	add2	c2	s2	z2	count2
4	1011	time4	name2	add2	c2	s2	z2	count2
5	1001	time5	name1	add1	c1	s1	z1	count1

By contrast, keeping account details in a separate table makes this only total of five updates. The larger the data set, the more that this matters.

Database Normalization

When creating a database, it is really important to think about how data will be stored. This is known as **normalization**, and it is a huge part of most SQL classes. If you are in charge of setting up a new database, it is important to have a thorough understanding of database **normalization**.

There are essentially three ideas that are aimed at database normalization:

1. Are the tables storing logical groupings of the data?
2. Can I make changes in a single location, rather than in many tables for the same information?
3. Can I access and manipulate data quickly and efficiently?

However, most analysts are working with a database that was already set up with the necessary properties in place. As analysts of data, you don't really need to think too much about data **normalization**. You just need to be able to pull the data from the database, so you can start drawing insights. This will be our focus in this lesson.

So, we understand that all of the information related to an account is not in the orders table, but **why not?**

There are more reasons for this kind of structure but for now, what you really need to know is how to a connect the account data and order data. The mechanism for doing that is a JOIN, which you'll learn next.

Introduction to JOINs

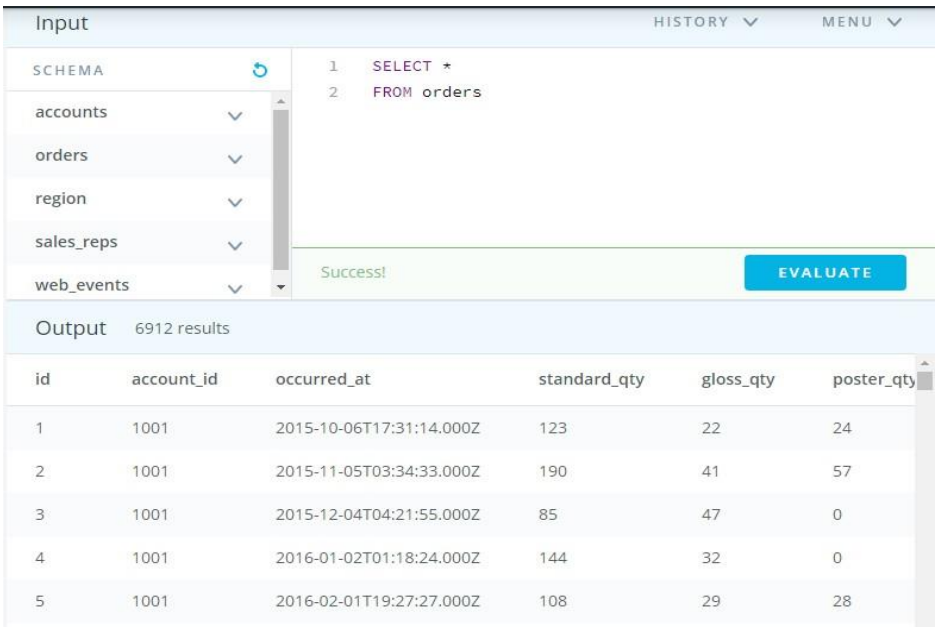
This entire lesson will be aimed at **JOIN**s. The whole goal of **JOIN** statements is to allow us to pull from more than one table at a time.

Again - **JOIN**s are useful for allowing us to pull data from multiple tables. This is both simple and powerful all at the same time.

With the addition of the **JOIN** statement to our toolkit, we will also be adding the **ON** statement.

JOINS will make your queries a little bit more complicated. Let's break down all the part of a JOIN to see what they do before you write some of your own. We'll start by writing what's called an INNER JOIN and we'll move on to other types of JOINS shortly.

In order to write a **JOIN**, the first thing will need are **SELECT** and **FROM** clauses just like any other query. In this case, we're looking to add account names to each order. So we'll start with the orders table. You'll notice I'm running this query even though it's not yet finished.

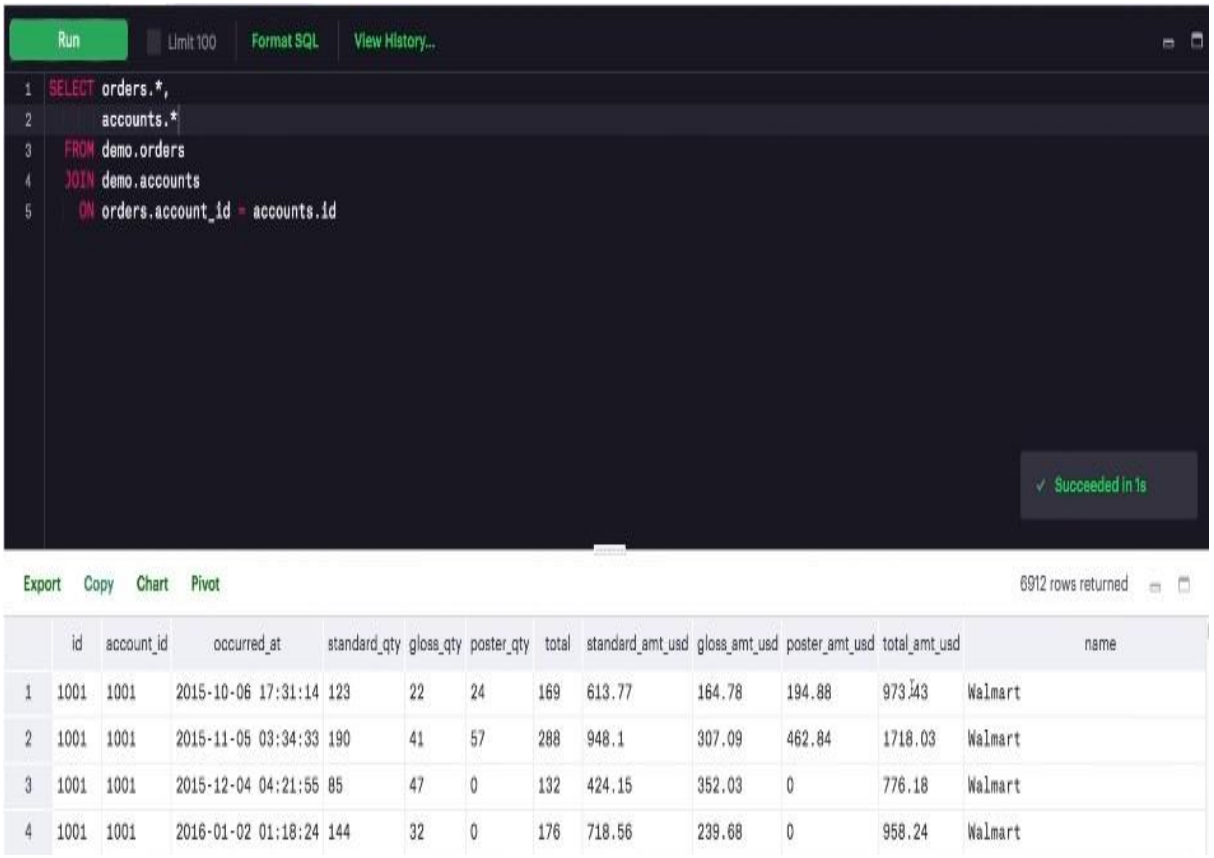


This is an easy way to make sure I haven't made any typos or errors in the first part of my query. Since it worked, we can move on to write the **JOIN**.

The next thing we need is a **JOIN** clause. You can think of this like a second **FROM** clause. It identifies the table where the data that we want to join lives.

In this case, account names live in the accounts table, so that's what we've written. Finally, we need to specify the relationship between the two tables. This happens by writing a logical statement in the ON clause.

Now that the JOIN is complete, let's add some columns from the accounts table into our SELECT statement to complete the query.



	id	account_id	occurred_at	standard_qty	gloss_qty	poster_qty	total	standard_amt_usd	gloss_amt_usd	poster_amt_usd	total_amt_usd	name
1	1001	1001	2015-10-06 17:31:14	123	22	24	169	613.77	164.78	194.88	973.43	Walmart
2	1001	1001	2015-11-05 03:34:33	190	41	57	288	948.1	307.09	462.84	1718.03	Walmart
3	1001	1001	2015-12-04 04:21:55	85	47	0	132	424.15	352.03	0	776.18	Walmart
4	1001	1001	2016-01-02 01:18:24	144	32	0	176	718.56	239.68	0	958.24	Walmart

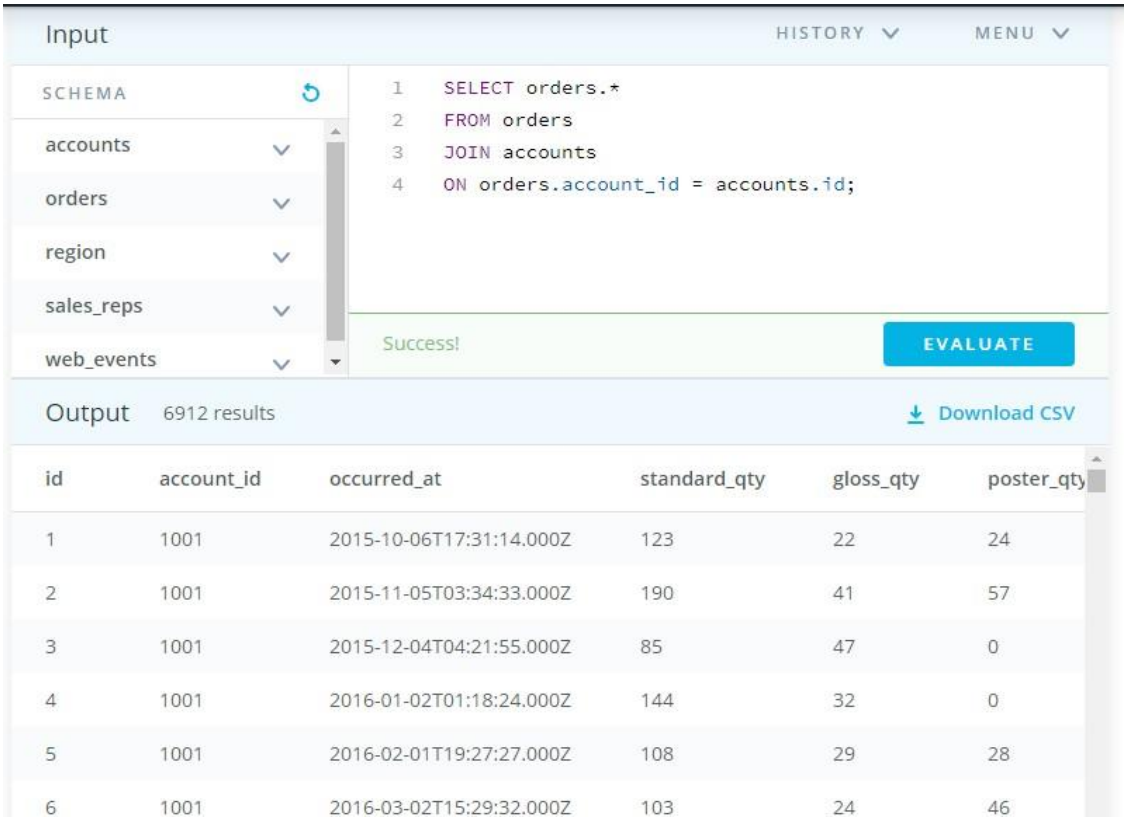
You can see that the results now contain information about the orders and the accounts that placed them.

Your First JOIN

Write Your First JOIN

Below is a **JOIN**, you will get a lot of practice writing these, and there is no better way to learn than practice. You will notice, we have introduced two new parts to our regular queries: **JOIN** and **ON**. The **JOIN** introduces the second table from which you would like to pull data, and the **ON** tells you how you would like to merge the tables in the **FROM** and **JOIN** statements together.

Example:



id	account_id	occurred_at	standard_qty	gloss_qty	poster_qty
1	1001	2015-10-06T17:31:14.000Z	123	22	24
2	1001	2015-11-05T03:34:33.000Z	190	41	57
3	1001	2015-12-04T04:21:55.000Z	85	47	0
4	1001	2016-01-02T01:18:24.000Z	144	32	0
5	1001	2016-02-01T19:27:27.000Z	108	29	28
6	1001	2016-03-02T15:29:32.000Z	103	24	46

What to Notice

We are able to pull data from two tables:

1. **orders**
2. **accounts**

Above, we are only pulling data from the **orders** table since in the SELECT statement we only reference columns from the **orders** table.

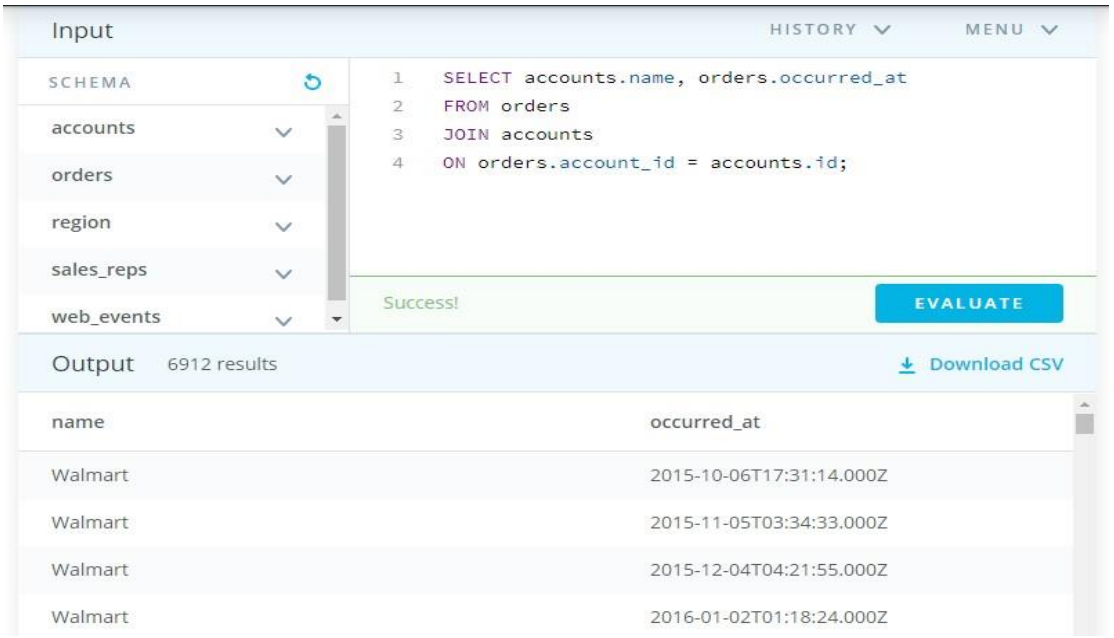
The **ON** statement holds the two columns that get linked across the two tables. This will be the focus in the next concepts.

Additional Information

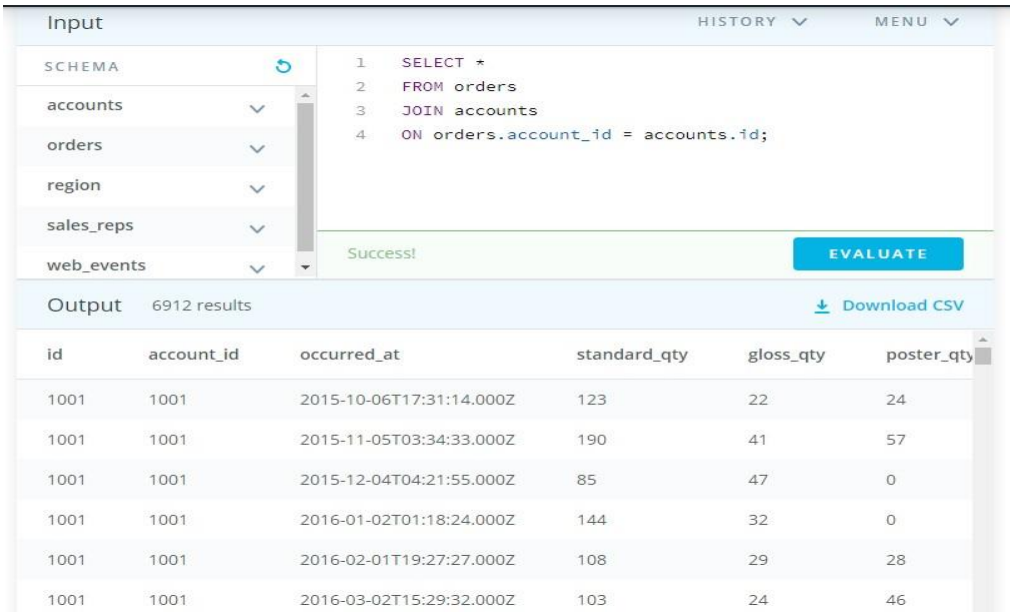
If we wanted to only pull individual elements from either the **orders** or **accounts** table, we can do this by using the exact same information in the **FROM** and **ON** statements. However, in your **SELECT** statement, **you will need to know how to specify tables and columns in the SELECT statement:**

1. The table name is always **before** the period.
2. The column you want from that table is always **after** the period.

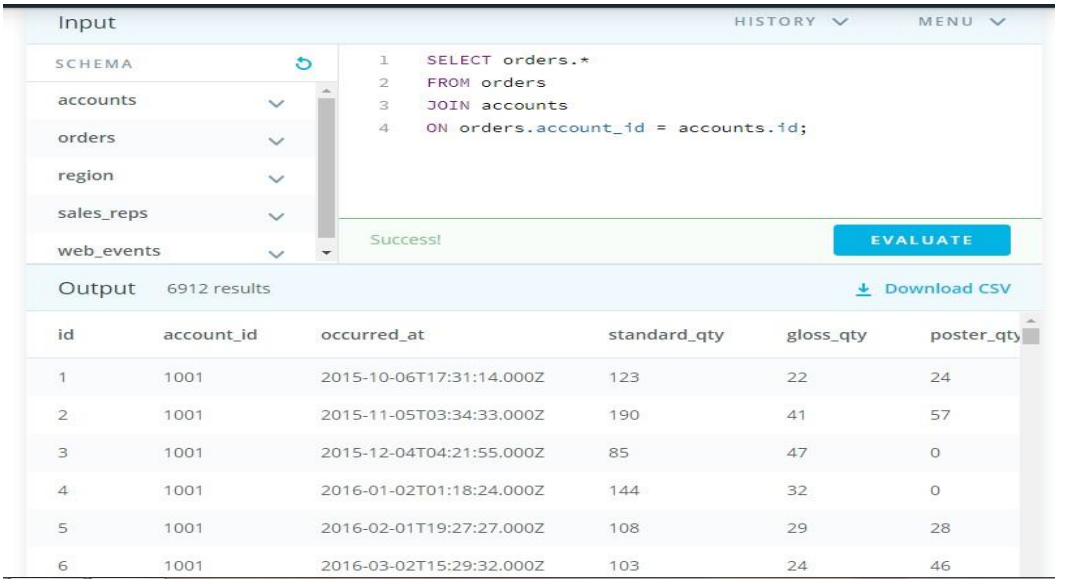
For example, if we want to pull only the **account name** and the dates in which that account placed an order, but none of the other columns, **we can do this with the following query:**



This query only pulls two columns, not all the information in these two tables. Alternatively, the below query pulls all the columns from both the **accounts** and **orders** table.



And the first query you ran pull all the information from only the orders table:

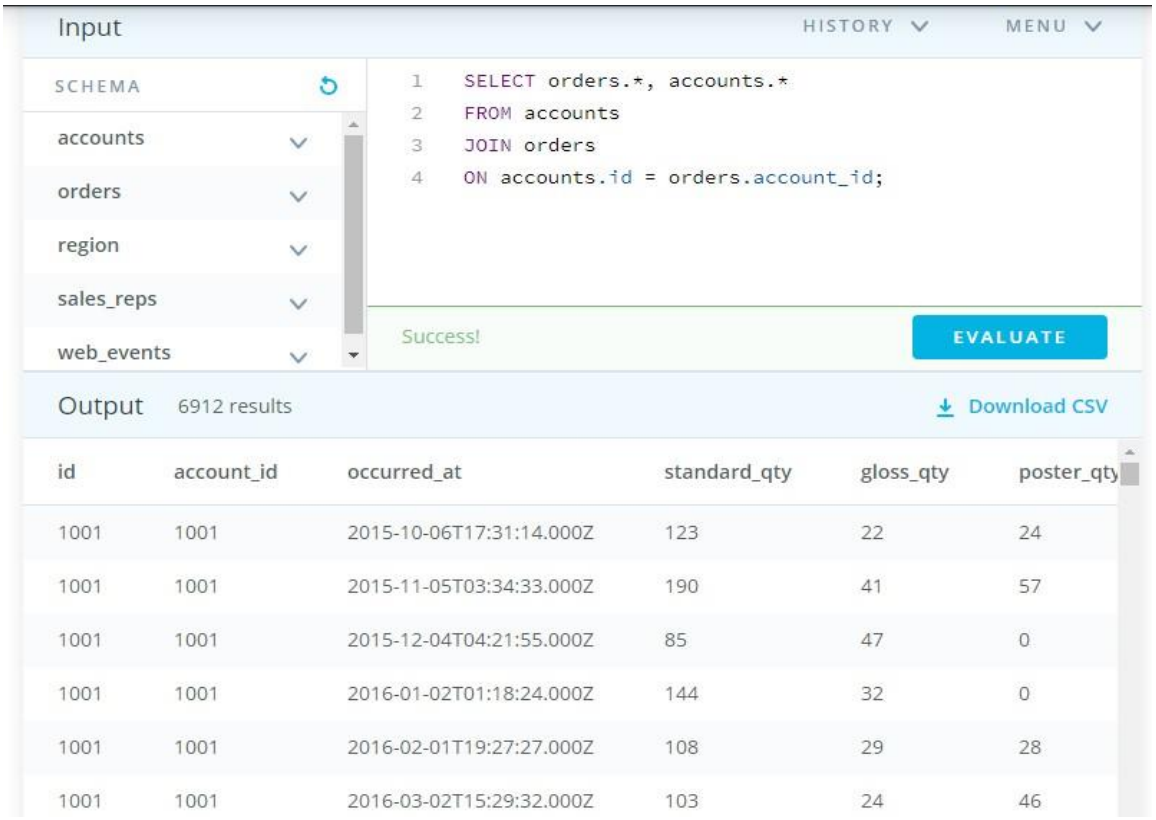


Joining tables allows you access to each of the tables in the **SELECT** statement through the table name, and the columns will always follow a **.** after the table name.

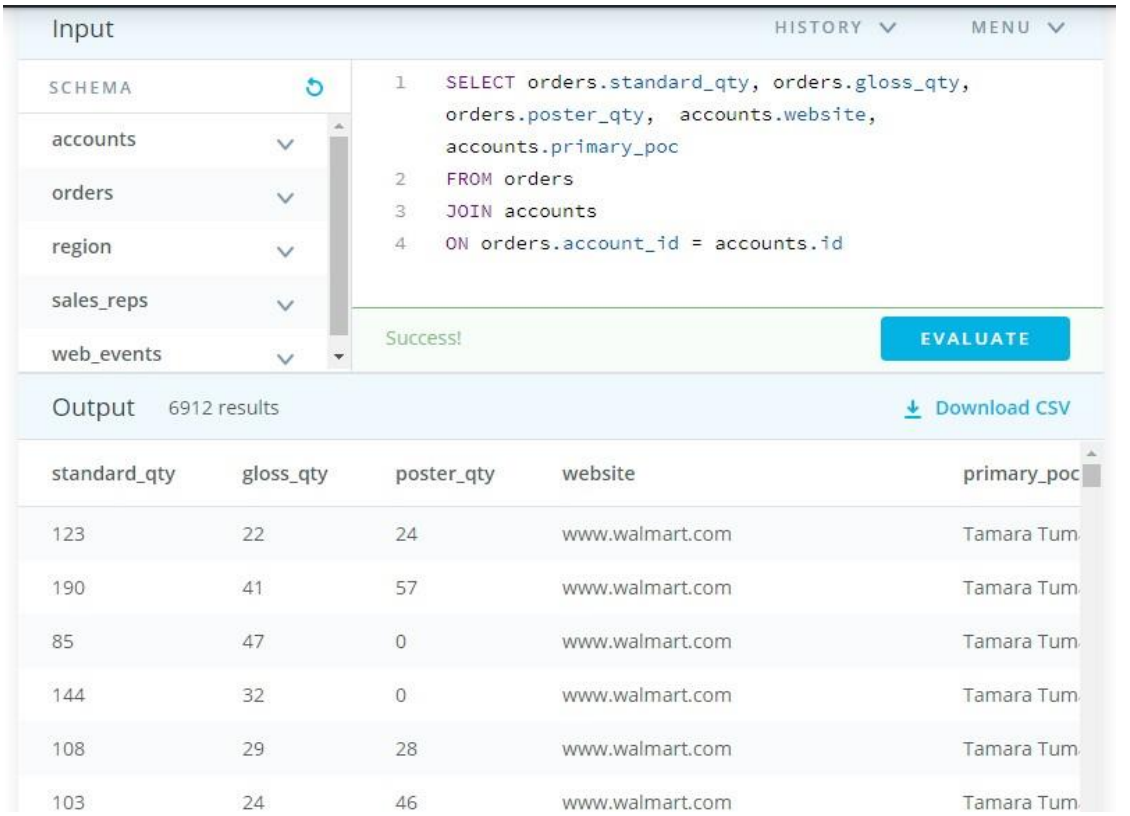
Quiz: Your First JOIN

Quiz Questions

1. Try pulling all the data from the **accounts** table, and all the data from the **orders** table.



2. Try pulling **standard_qty**, **gloss_qty**, and **poster_qty** from the **orders** table, and the **website** and the **primary_poc**from the **accounts** table.

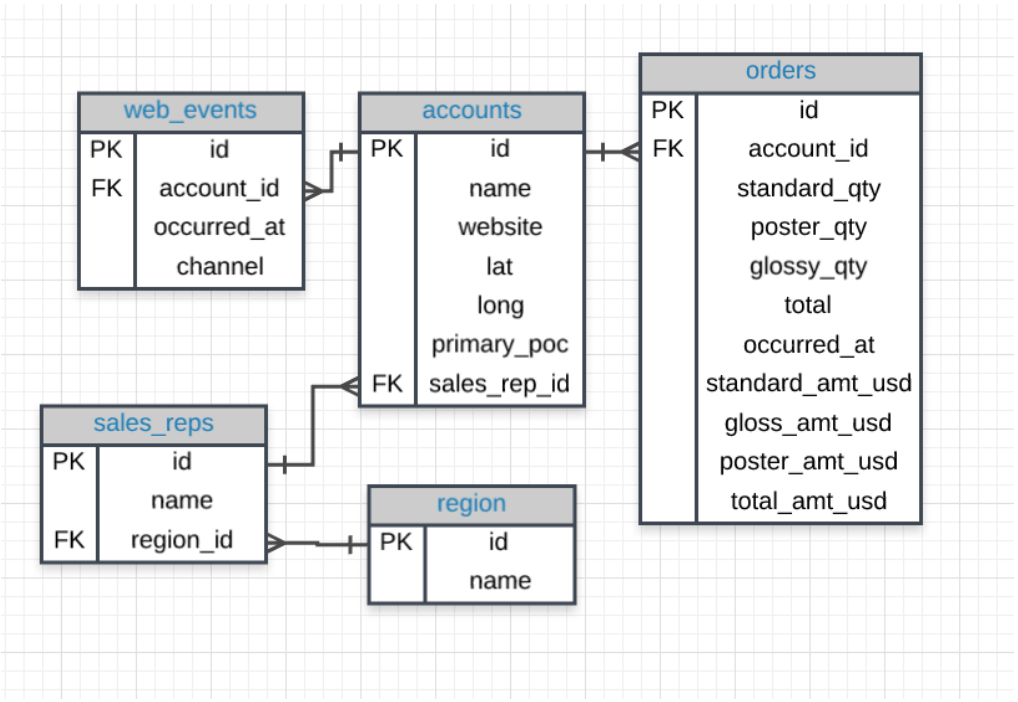


ERD Reminder

Entity Relationship Diagrams

From the last lesson, you might remember that an entity relationship diagram **(ERD)** is a common way to view data in a database. It is also a key element to understanding how we can pull data from multiple tables.

It will be beneficial to have an idea of what the **ERD** looks like for Parch & Posey handy, so I have posted it again below.



Tables & Columns

In the Parch & Posey database there are 5 tables:

1. web_events
2. accounts
3. orders
4. sales_reps
5. region

You will notice some of the columns in the tables have **PK** or **FK** next to the column name, while other columns don't have a label at all.

If you look a little closer, you might notice that the **PK** is associated with the first column in every table. The **PK** here stands for **primary key**. **A primary key exists in every table, and it is a column that has a unique value for every row.**

If you look at the first few rows of any of the tables in our database, you will notice that this first, **PK**, column is always unique. For this database it is always called id, but that is not true of all databases.

Primary and Foreign Keys

Primary Key (PK)

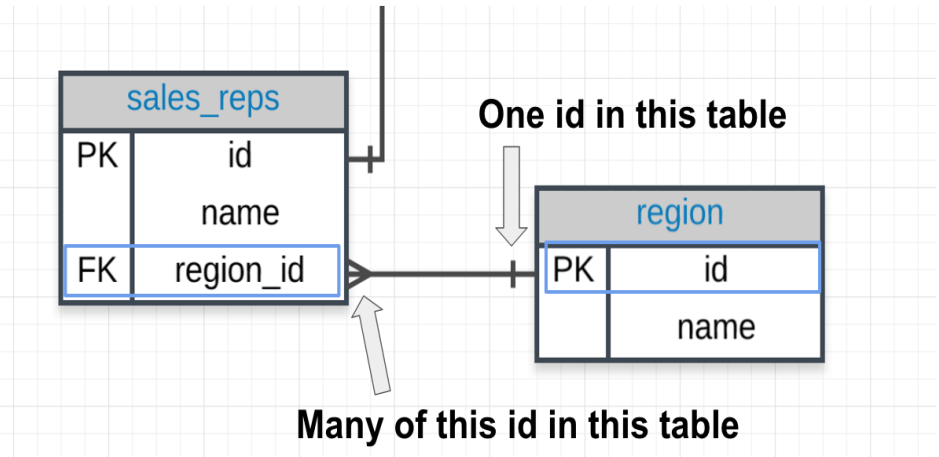
A **primary key** is a unique column in a particular table. This is the first column in each of our tables. Here, those columns are all called **id**, but that doesn't necessarily have to be the name. **It is common that the primary key is the first column in our tables in most databases.**

Foreign Key (FK)

A **foreign key** is when we see a primary key in another table. **We can see these in the previous ERD the foreign keys are provided as:**

1. region_id
2. account_id
3. sales_rep_id

Each of these is linked to the **primary key** of another table. **An example is shown in the image below:**



Primary - Foreign Key Link

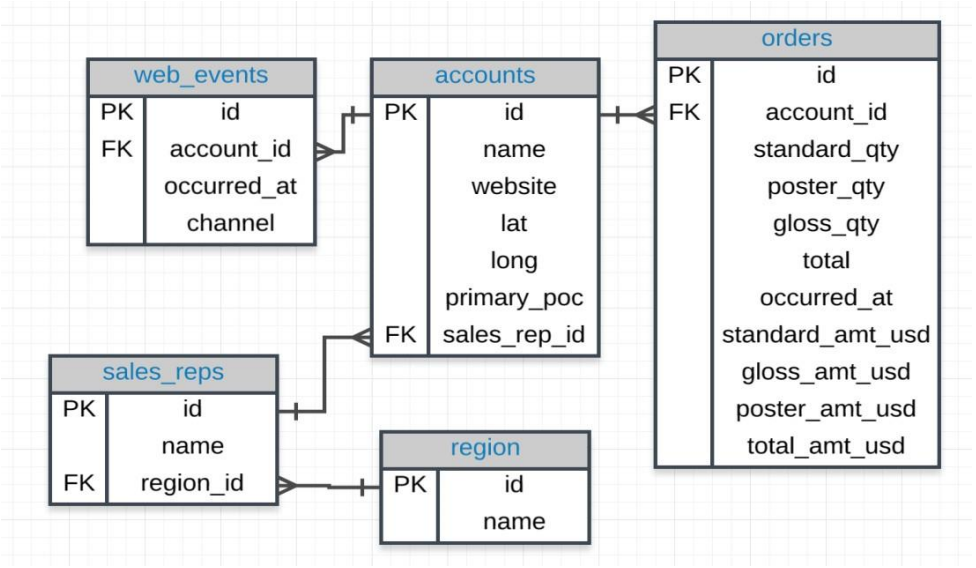
In the above image you can see that:

1. The **region_id** is the foreign key.
2. The region_id is **linked** to id - this is the primary-foreign key link that connects these two tables.
3. The crow's foot shows that the **FK** can actually appear in many rows in the **sales_reps** table.
4. While the single line is telling us that the **PK** shows that id appears only once per row in this table.

If you look through the rest of the database, you will notice this is always the case for a primary-foreign key relationship.

Quiz: Primary - Foreign Key Relationship

Helpful ERD For Answering the Below Questions



QUESTION 1 OF 3

Match the appropriate definition or description to each term or column.

ON accounts.id = web_events.id

ON web_events.id = accounts.id

ON accounts.account_id = web_events.id

DEFINITION OR COLUM N DESCRIPTION	TERM OR COLU M N
Has a unique value for every row in that column. There is one in every table.	Primary Key
The **link** to the primary key that exists in another table.	Foreign Key
The primary key in every table of our example database.	id
A foreign key that exists in both the **web_events** and **orders** tables.	account_id
The **ON** statement associated with a **JOIN** of the **web_events** and **accounts** tables.	ON web_events.account_id =accounts.id

QUESTION 2 OF 3

Select all that are true for primary keys.

- [x] There is one and only one of these columns in every table.
- [x] They are a column in a table.
- [ ] There might be more than one primary key for a table.
- [ ] They are a row in a table.
- [ ] Every database only has one primary key for the whole database.

QUESTION 3 OF 3

Select all that are true of foreign keys.

- [x] They are always linked to a primary key.
- [ ] They are unique for every row in a table.
- [ ] Every table must have a foreign key.
- [ ] A table can only have one foreign key.
- [x] In the above database, every foreign key is associated with the crow-foot notation, which suggests it can appear multiple times in the column of a table.

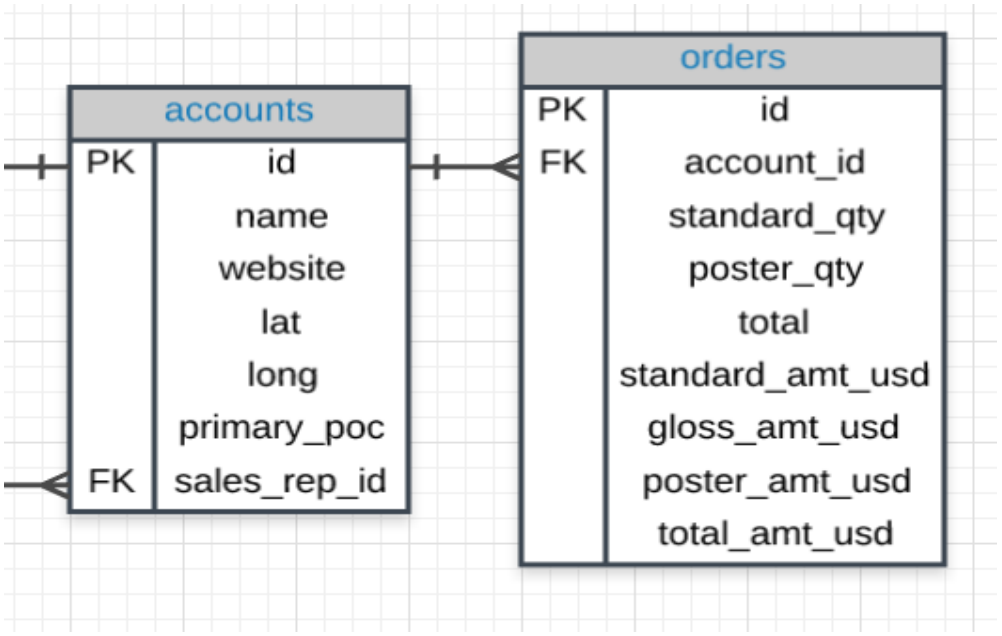
JOIN Revisited

JOIN Revisited

Let's look back at the first JOIN you wrote.

```
SELECT orders.*
FROM orders
JOIN accounts
ON orders.account_id = accounts.id;
```

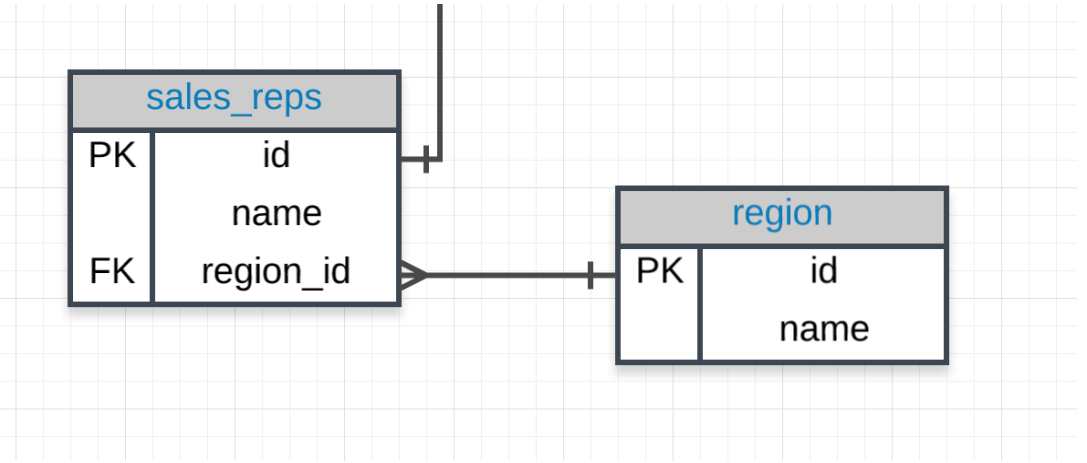
Here is the ERD for these two tables:



Notice

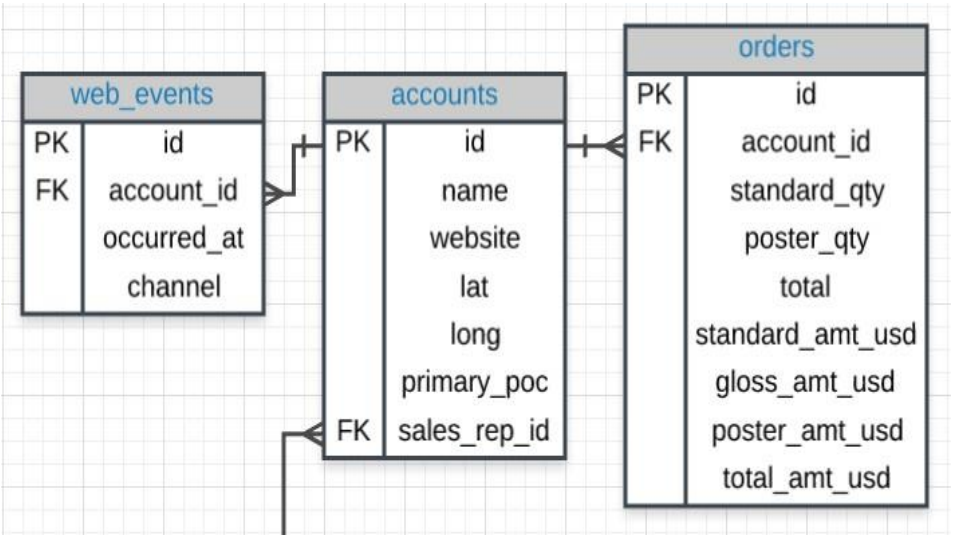
Notice our SQL query has the two tables we would like to join - one in the **FROM** and the other in the **JOIN**. Then in the **ON**, we will **ALWAYs** have the **PK** equal to the **FK**:

The way we join any two tables is in this way: linking the **PK** and **FK** (generally in an **ON** statement).



JOIN More than Two Tables

This same logic can actually assist in joining more than two tables together. Look at the three tables below.



The Code

If we wanted to join all three of these tables, we could use the same logic. The code below pulls all of the data from all of the joined tables.

```
SELECT *
FROM web_events
JOIN accounts
ON web_events.account_id = accounts.id
JOIN orders
ON accounts.id = orders.account_id
```

Alternatively, we can create a **SELECT** statement that could pull specific columns from any of the three tables. Again, our **JOIN** holds a table, and **ON** is a link for our **PK** to equal the **FK**.
To pull specific columns, the **SELECT** statement will need to specify the table that you are wishing to pull the column from, as well as the column name. We could pull only three columns in the above by changing the select statement to the below, **but maintaining the rest of the JOIN information:**

```
SELECT web_events.channel, accounts.name, orders.total
```

We could continue this same process to link all of the tables if we wanted. For efficiency reasons, we probably don't want to do this unless we actually need information from all of the tables.

Quiz: JOIN Revisited

QUIZ QUESTION

Practice

Use the image above to assist you. If we wanted to join the sales_reps and region tables together, how would you do it

- ON sales_reps.id = region.id
- ON sales_reps.id = region.name
- ON sales_reps.region_id = region.id
- ON region.id = sales_reps.id

Alias

When we **JOIN** tables together, it is nice to give each table an **alias**. Frequently an alias is just the first letter of the table name. You actually saw something similar for column names in the **Arithmetic Operators** concept.

Example:

```
FROM table name AS t1
JOIN tablename2 AS t2
```

Before, you saw something like:

```
SELECT col1 + col2 AS total, col3
```

Frequently, you might also see these statements without the **AS** statement. Each of the above could be written in the following way instead, and they would still produce the **exact same results**:

```
FROM table name t1
JOIN tablename2 t2
```

and

```
SELECT col1 + col2 total, col3
```

Aliases for Columns in Resulting Table

While aliasing tables is the most common use case. It can also be used to alias the columns selected to have the resulting table reflect a more readable name.

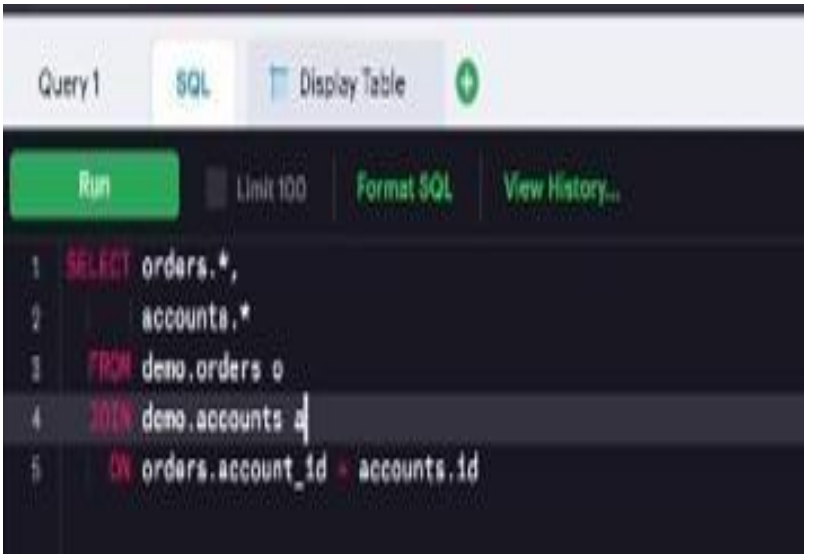
Example:

```
Select t1.column1 alias name, t2.column2 aliasname2
FROM table name AS t1
JOIN tablename2 AS t2
```

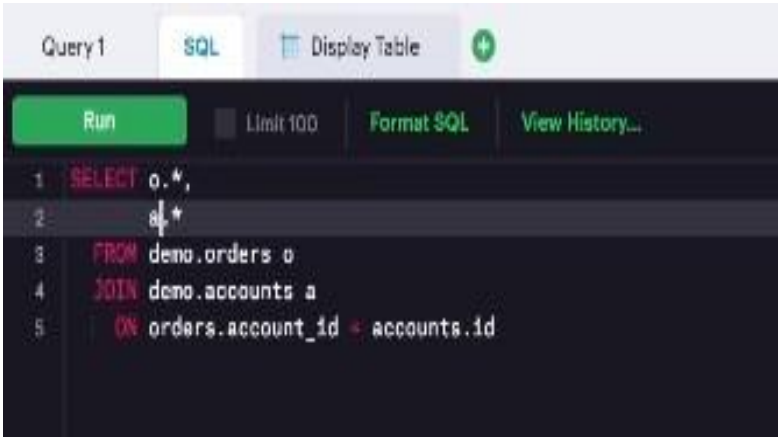
The alias name fields will be what shows up in the returned table instead of t1.column1 and t2.column2

aliasname	aliasname2
example row	example row
example row	example row

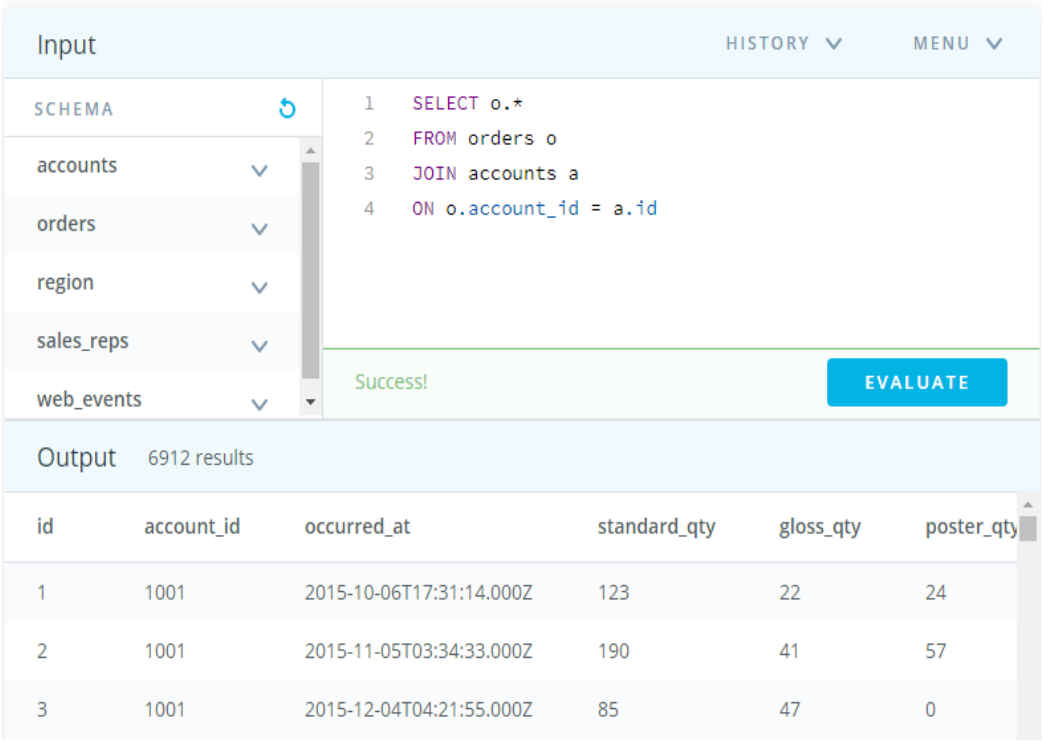
When performing joins, it's easiest to give your table names aliases. Orders is a pretty long and annoying name to type. O is much easier. You can give a table an alias by adding a space after the table name and typing the intended name of the alias. As with column names, the best practice here is to use all lower case letters and underscores instead of spaces.



Now that we've given our tables aliases, we can refer to columns in the SELECT clause by using those alias names. We can also do the same thing anywhere else in the query that columns from these tables appear.



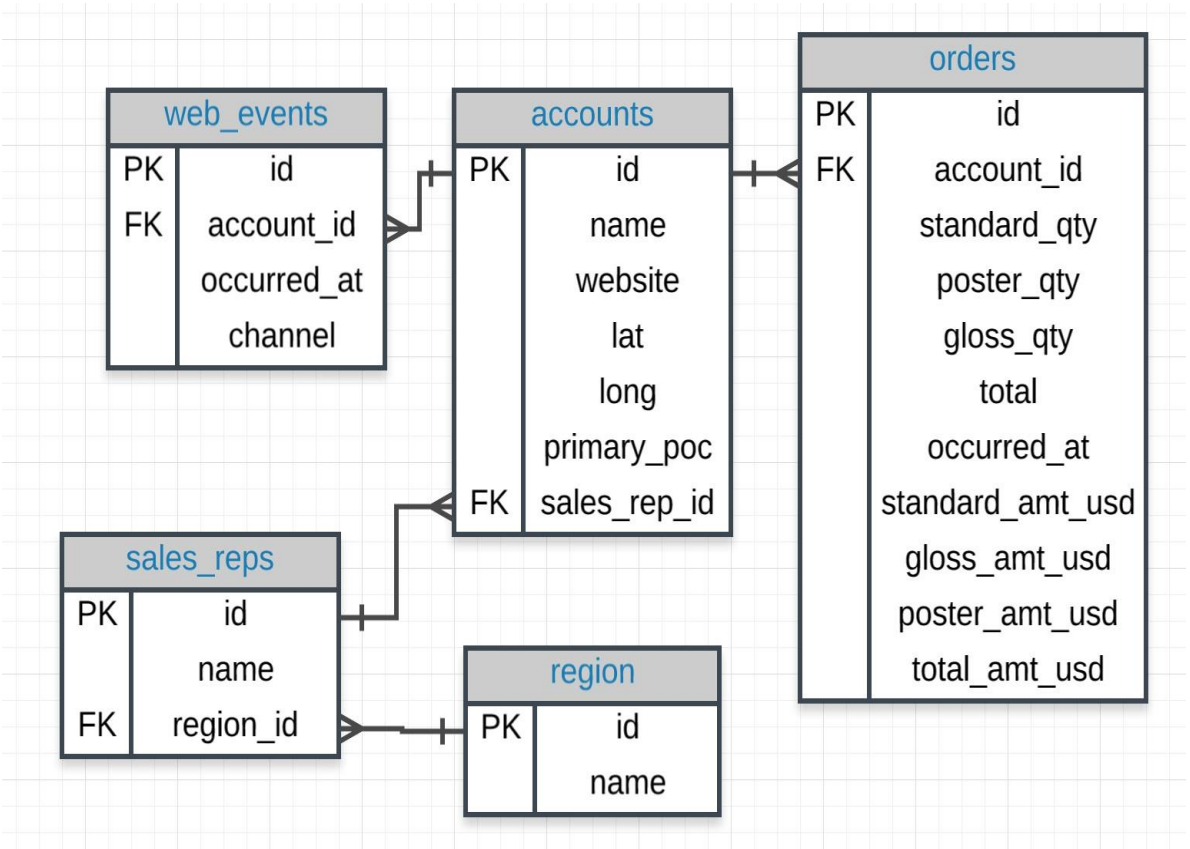
In this case, the only other places the ON clause. As you can see, the results are exactly the same as the previous query without the aliases.



Quiz: JOIN Questions Part I

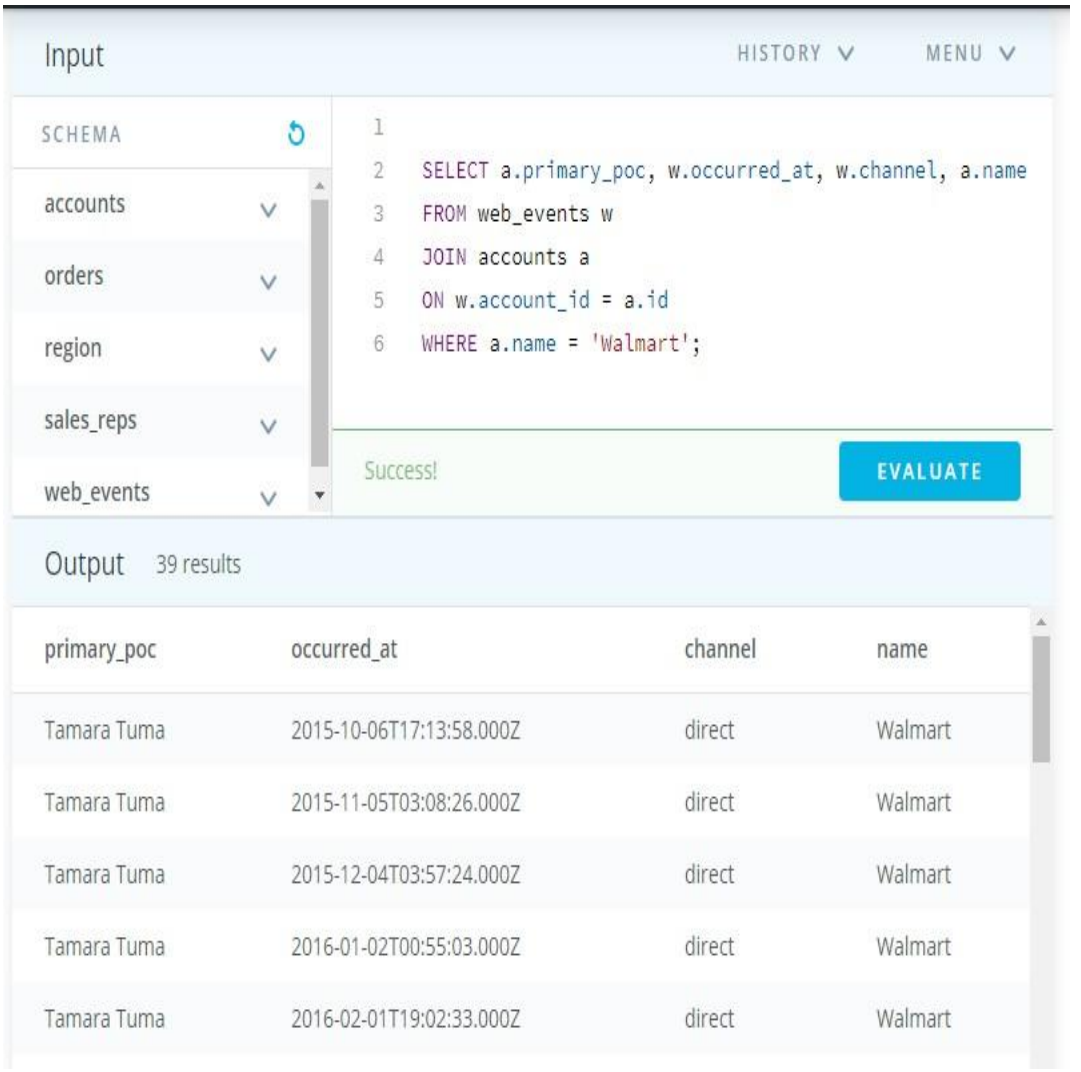
Question Mania

Now that you have been introduced to JOINs, let's practice to build your skills and comfort with this new tool. Below I have provided the **ERD** and a bunch of questions. The solutions for the questions can be found on the next concept for you to check your answers or just in case you get stuck!

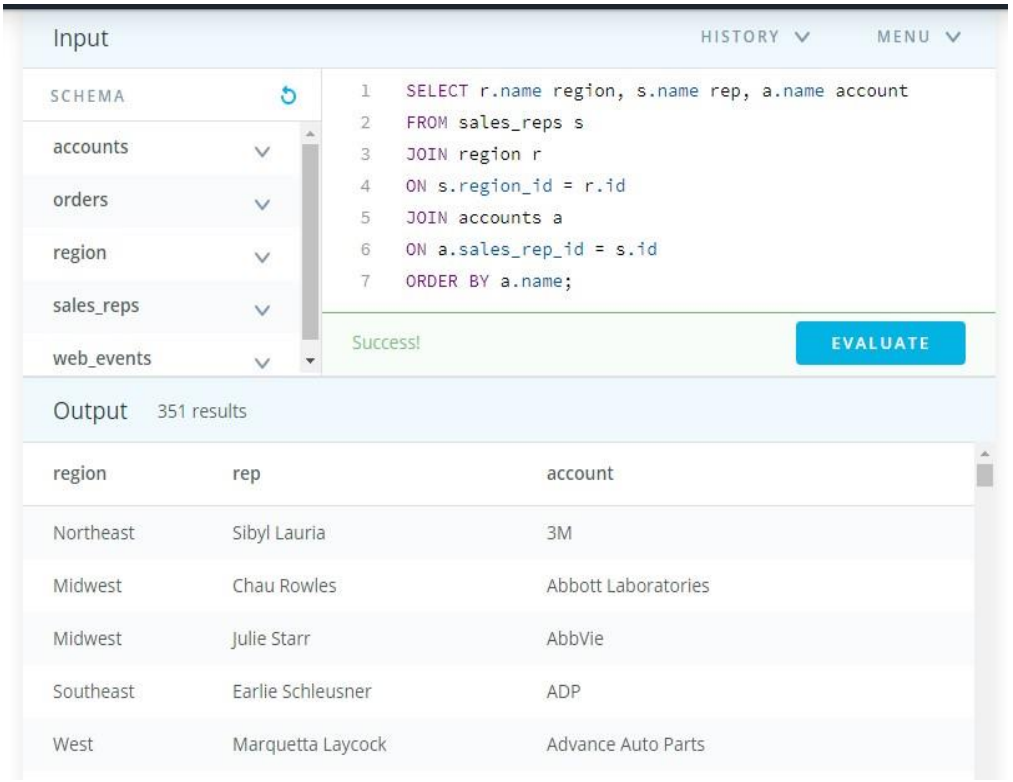


Questions

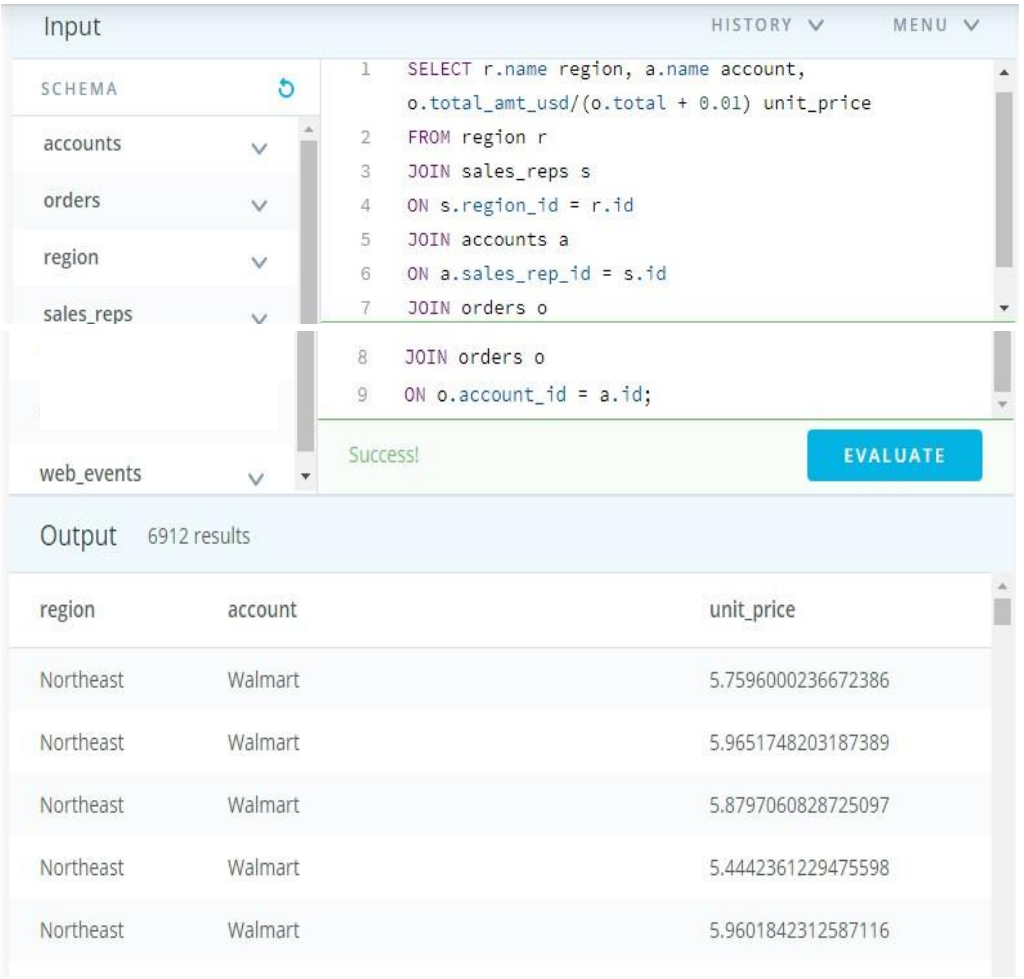
1. Provide a table for all **web_events** associated with account **name** of Walmart. There should be three columns. Be sure to include the primary_poc, time of the event, and the channel for each event. Additionally, you might choose to add a fourth column to assure only Walmart events were chosen.



2. Provide a table that provides the region for each sales_rep along with their associated accounts. Your final table should include three columns: the region **name**, the sales rep name, and the account name. Sort the accounts alphabetically (A-Z) according to account name.



3. Provide the **name** for each region for every **order**, as well as the account **name** and the **unit price** they paid (total_amt_usd/total) for the order. Your final table should have 3 columns: **region name**, **account name**, and **unit price**. A few accounts have 0 for **total**, so I divided by (total + 0.01) to assure not dividing by zero.



QUESTION 1 OF 2

Mark all the below that are true.

- [x] The **ON** statement **should** always occur with the foreign key being equal to the primary key.
- [ ] **SQL requires** the **ON** statement always occur with the foreign key being equal to the primary key.
- [x] **JOIN** statements allow us to pull data from multiple tables in a **SQL** database.
- [x] You can use all of the commands we saw in the first lesson along with **JOIN** statements.
- [ ] The only way to pull data together from multiple tables is using a **JOIN** statement.

QUESTION 2 OF 2

Select all of the below statements that are true.

- [ ] If we want to alias a table, we **must** use an **AS** in our query.
- [ ] If we want to alias a column, we **must** use an **AS** in our query.
- [x] Aliasing is common to shorten table names when we start **JOIN**ing multiple tables together.

Motivation for Other JOINs

Expert Tip

You have had a bit of an introduction to these **one-to-one** and **one-to-many** relationships when we introduced **PK**s and **FK**s. Notice, traditional databases do not allow for **many-to-many** relationships, as these break the schema down pretty quickly The types of relationships that exist in a database matter less to analysts, but you do need to understand why you would perform different types of **JOIN**s, and what data you are pulling from the database.

One thing you might have noticed when working with inner joins is that the data you join can have a one to one or a one to many relationship. In other ward, one account might only have placed a single order while another account might have placed many orders.

Since we're focused primarily on the orders, you probably missed a different kind of relationship.

A few of these accounts have just developed brand new relationships with Parch & Posey sales team and haven't placed any orders yet. So, when we join these tables, what happens to the accounts without any orders? Where do they go?

It turns out you might want to keep them around or drop them from your result set depending on what you're going for.

For example, if you're simply attaching account names to each order, then excluding these accounts without orders is probably fine. That's exactly what we did in our inner join earlier.

If your goal is to count up all of the accounts in the region along with their quantities of paper purchased, you might want to include the accounts without any orders. This is done using an outer join.

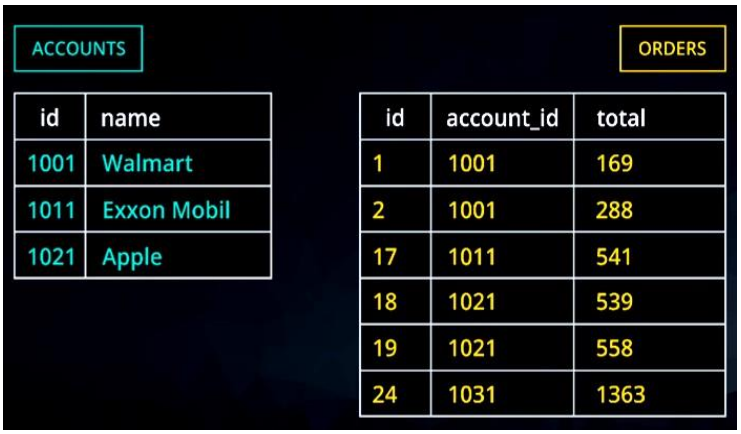
The next section, I'll show you a few different types of joins and give the opportunity to practice writing some yourself.

LEFT and RIGHT JOINs

JOINs

Notice each of these new **JOIN** statements pulls all the same rows as an **INNER JOIN**, which you saw by just using **JOIN**, but they also potentially pull some additional rows.
If there is not matching information in the **JOIN**ed table, then you will have columns with empty cells. These empty cells introduce a new data type called **NULL**. You will learn about **NULL**s in detail in the next lesson, but for now you have a quick introduction as you can consider any cell without data as **NULL**.

For the following example, I'll use small subsets of the accounts and orders tables shown here.



ACCOUNTS

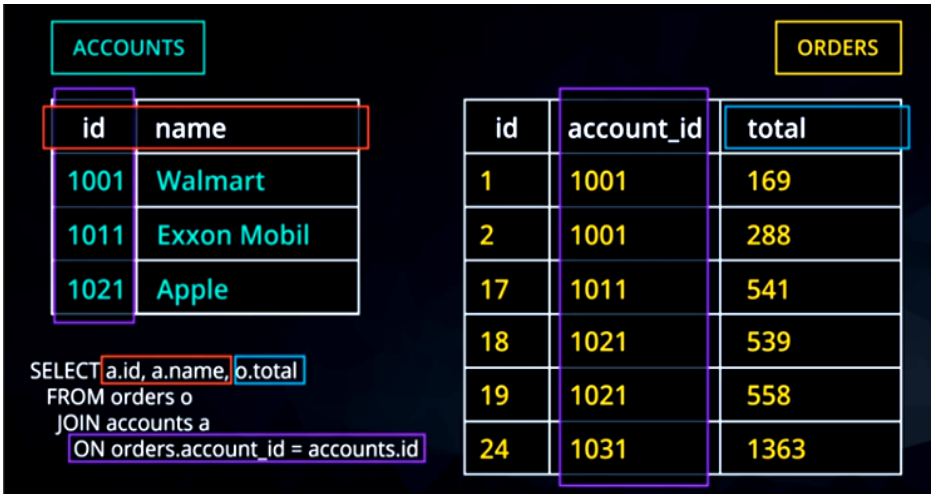
id	name
1001	Walmart
1011	Exxon Mobil
1021	Apple

ORDERS

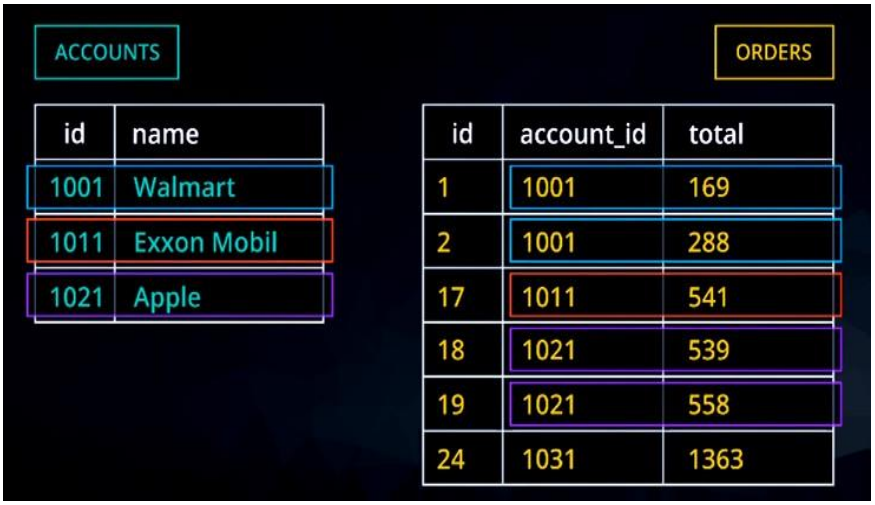
id	account_id	total
1	1001	169
2	1001	288
17	1011	541
18	1021	539
19	1021	558
24	1031	1363

First, let's quickly recap the inner join. This will return only rows that appear in both tables.

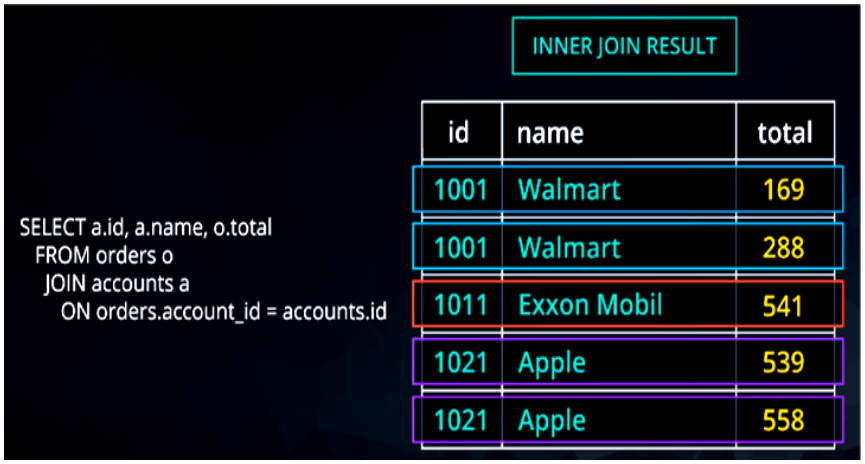
Writing the following INNER JOIN query to join the accounts and orders tables will pull these columns from the accounts table and these column in the orders table, only for the values of the account_id in the orders table that match with the id field in the accounts table.



This will pull these three rows from the accounts table and match them with these five rows from the orders table.

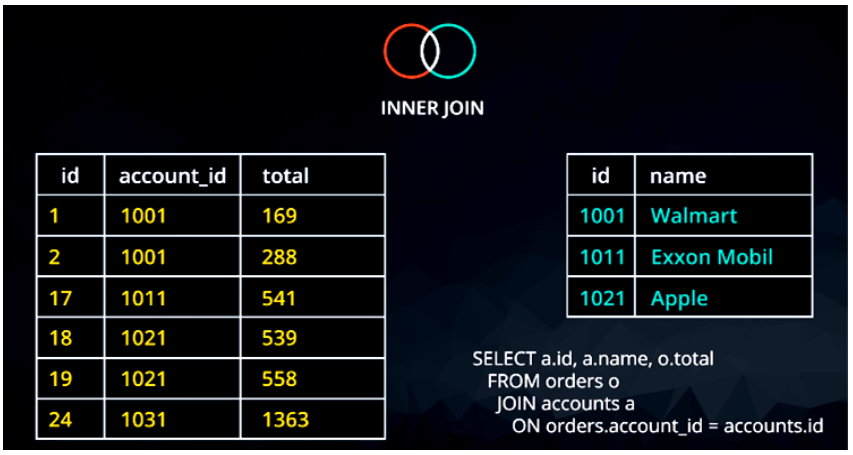


So the table we'll get back will look like this.



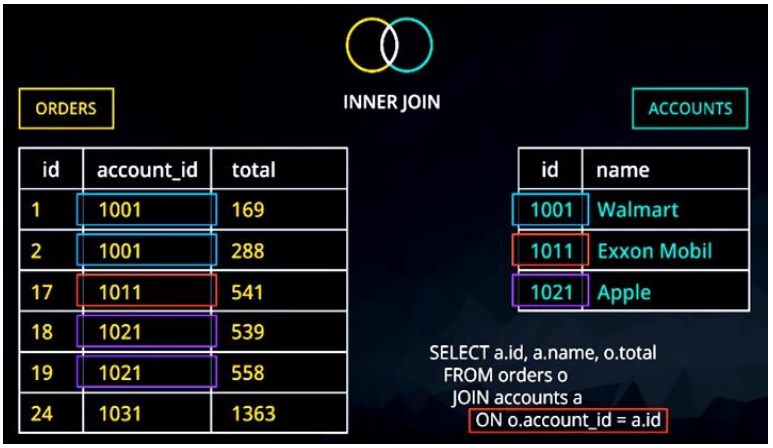
Notice we have only the three account IDs that were in both tables.

A common way to visualize different JOINs is with a venn diagram.



Each circle in the diagram represents table. The left circle includes all rows of data in the table in our FROM clause, which is the orders table. The right circle represents all rows of data in the table in our JOIN clause. In this case, that's the accounts table.

The overlapping middle section represent all rows for which the ON clause is true. An INNER JOIN, as shown here, will return only rows at the intersection of these to circles.



So what about this new account that hasn't placed any orders, yet ?

If we want to show that there are existing accounts that do not appear in the orders table, an **INNER JOIN** is not going to help us. In the upcoming sections, we'll work with other **OUTER JOINs** to pull that type information.

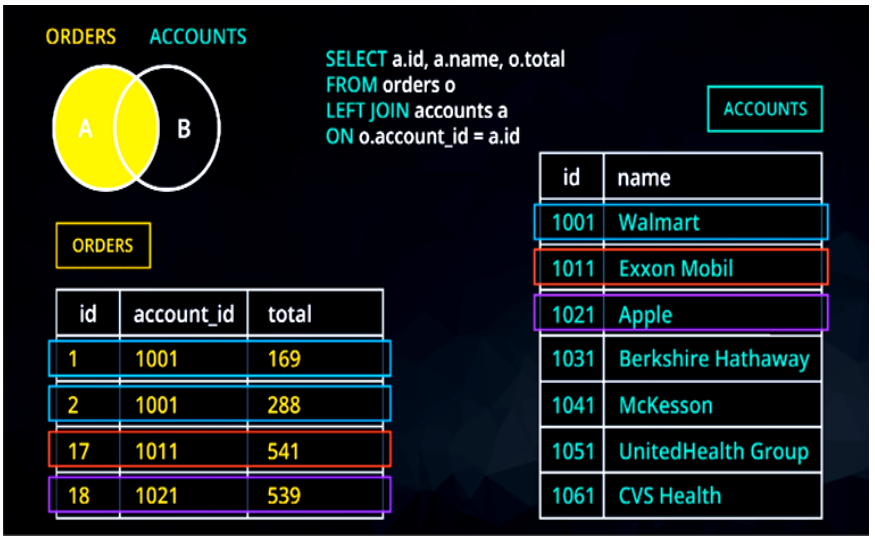
If we want to include data that doesn't exist in both tables, but only in one of the two tables we are using in our joint statement, **there are three types of joins we might use: a left join, a right join and a full outer join.**

Each of these joins will provide all the resulting rows of an INNER JOIN, but we may also gain some additional rows. So the result of an OUTER JOIN will always have at least as many rows as an INNER JOIN, with the same logic in the, on clause.

This lesson is going to focus primarily on LEFT and RIGHT JOINS. When we read our SELECT statements, the table listed in the FROM statement is considered the left table, while the table in the join considered the right table.

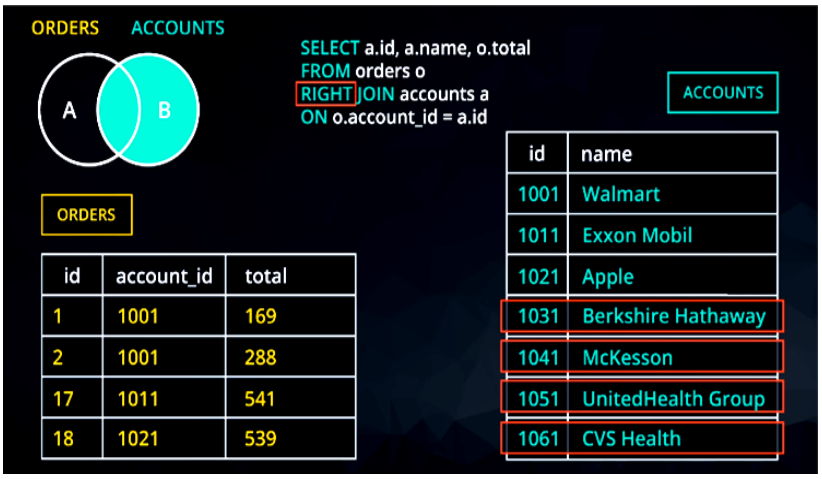
By simply writing the word , left. The result that we get back, will include of all the results that match with the right table, just like in an inner join. It will also return any results that are in the left table that did not match.

Using these subsets of the accounts and orders tables, every single row in the orders table here, can be matched to a row in the accounts table, with an ID in the accounts table the corresponds to an account ID in the orders table.



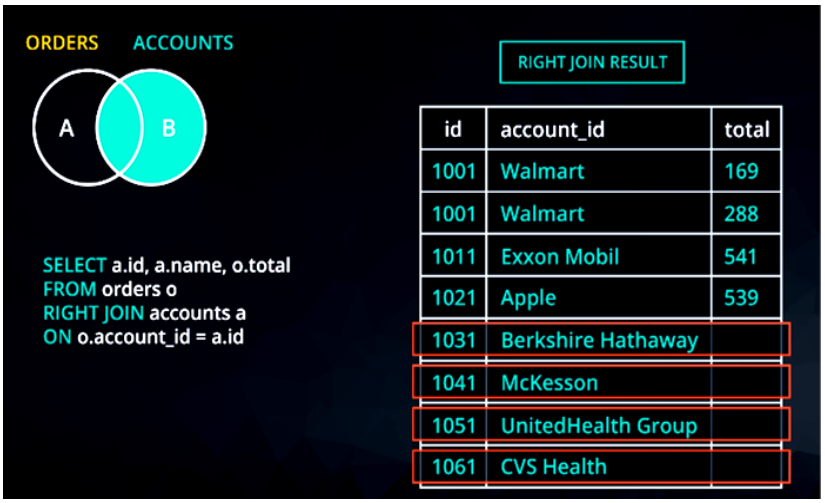
This result set exactly the same as if we had used an inner join.

A right join would produce a different set of results, though. There are four accounts without any orders, which means four rows of data that do not satisfy the condition of this join.

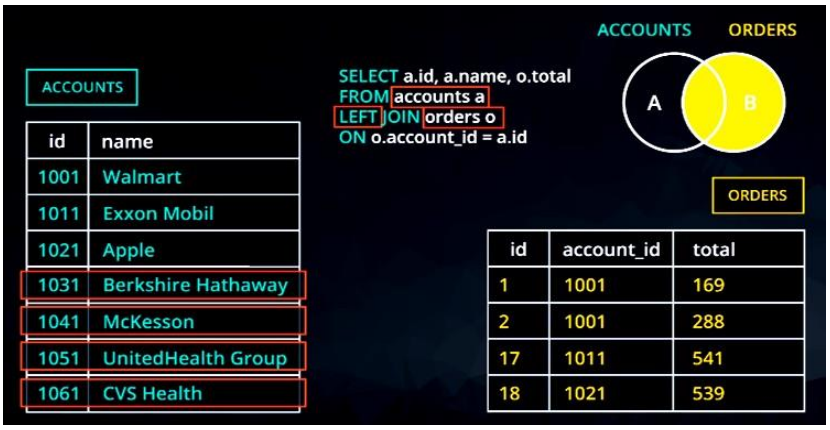


Since it's right join and not an inner join, these rows will be included at the bottom of the result set. They don't match with rows in the left table, so any columns from the left table will contain no data for these rows. Will discuss more about that this means soon enough.

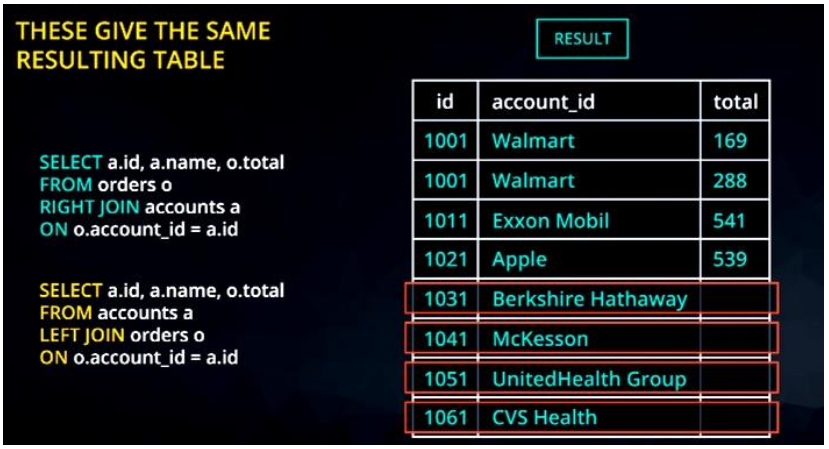
You can see that after we join, this final table has all the same rows as the original inner join table. But it has theses extra rows from the accounts table at the bottom that were not matched with any orders.



Left and right joins are somewhat interchangeable. If we change this query so that the accounts table is in the FROM clause, and the orders table is in the join clause, the extra rows will be in the left table.



We can now run a left join, and the results will be exactly the same as the right join we did with the previous query.



Because left and right joins are effectively interchangeable, we use left joins through the rest of this course. The world is by and large standardized around this behavior, so you'll rarely see a right join in the wild. This makes it easier to read other people's queries, and quickly understand what they intended to accomplish.

Consistency is especially important if you're writing a query with multiple joins. We're not quite there yet, but we will be soon.

Other JOIN Notes

JOIN Check In

INNER JOINs

Notice **every** JOIN we have done up to this point has been an **INNER JOIN**. That is, we have always pulled rows only if they exist as a match across two tables.

Our new **JOIN**s allow us to pull rows that might only exist in one of the two tables. This will introduce a new data type called **NULL**. This data type will be discussed in detail in the next lesson.

Quick Note

You might see the SQL syntax of

```
LEFT OUTER JOIN
```

OR

```
RIGHT OUTER JOIN
```

These are the exact same commands as the **LEFT JOIN** and **RIGHT JOIN** we learned about in the previous.

OUTER JOINS

The last type of join is a full outer join. This will return the inner join result set, as well as any unmatched rows from either of the two tables being joined.

Again this returns rows that **do not match** one another from the two tables. The use cases for a full outer join are **very rare**.

Similar to the above, you might see the language **FULL OUTER JOIN**, which is the same as **OUTER JOIN**.

*Quiz : LEFT and RIGHTJOIN

QUESTION 1 OF 4

Select all of the below that are true.

- [x] A **LEFT JOIN** and **RIGHT JOIN** do the same thing if we change the tables that are in the **FROM** and **JOIN**statements.
- [x] A **LEFT JOIN** will **at least** return all the rows that are in an **INNER JOIN**.
- [x] **JOIN** and **INNER JOIN** are the same.
- [x] A **LEFT OUTER JOIN** is the same as **LEFT JOIN**.

Country

countryid	countryName
1	India
2	Nepal
3	United States
4	Canada
5	Sri Lanka
6	Brazil

State

stateid	countryid	stateName
1	1	Maharashtra
2	1	Punjab
3	2	Kathmandu
4	3	California
5	3	Texas
6	4	Alberta

Above are two small tables for you to test your knowledge of **JOIN**s. You can click on the image to get a better view.

Country has 6 rows and 2 columns:

- **Country id** and **country Name**

State has 6 rows and 3 columns:

- **State id**, **country id**, and **state Name**

Use the above tables to determine the solution to the following questions.

QUESTION 2 OF 4

Match each statement to the item it describes.

Country.contryName	State.stateName
DESCRIPTION	**ITEM**
The primary key of the **Country** table.	Country.countryid
The primary key of the **State** table.	State.stateid
The foreign key that would be used in **JOIN**ing the tables.	State.countryid

If you were to perform the following query:

```
SELECT c.countryid, c.countryName, s.stateName
FROM Country c
JOIN State s
ON c.countryid = s.countryid;
```

QUESTION 3 OF 4

Match the results of the query to the description.

8 **5** **4** **7** **1** **12**

DESCRIPTION	RESULT
The number of columns in resulting table.	**3**
The number of rows in the resulting table.	**6**
The number of times countryid 1 will show up in resulting table.	**2**
The number of times countryid 6 will show up in the resulting table.	**0**

Note :

That's right! Since this is a JOIN (INNER JOIN technically), we only get rows that show up in both tables. Therefore our resulting table will essentially look like the right table with the country Name pulled in as a column. Since 1, 2, 3, and 4 are country ids in both tables, this information will be pulled together. The country ids of 5 and 6 only show up in the Country table. Therefore, these will be dropped.

If you were to perform the following query:

```
SELECT c.countryid, c.countryName, s.stateName
FROM Country c
LEFT JOIN State s
ON c.countryid = s.countryid;
```

QUESTION 4 OF 4

Match the results of the query to the description.

10 0 4 9 7 6 5

DESCRIPTION	RESULTS
The number of columns in resulting table.	3
The number of rows in the resulting table.	8
The number of times countryid 1 will show up in resulting table.	2
The number of times countryid 6 will show up in the resulting table.	1

NOTE :

That's right! We have a column for each of the identified elements in our **SELECT** statement. We will have all of the same rows as in a **JOIN** statement, but we also will obtain the additional two rows in the **Country** table that are not in the **State** table for **Sri Lanka** and **Brazil**.

JOINs and Filtering

A simple rule to remember this is that, when the database executes this query, it executes the join and everything in the **ON** clause first. Think of this as building the new result set. That result set is then filtered using the **WHERE** clause.

The fact that this example is a left join is important. Because inner joins only return the rows for which the two tables match, moving this filter to the **ON** clause of an inner join will produce the same result as keeping it in the **WHERE** clause.

The tricky thing about joins is that there are means to do other types of analysis. Combining data isn't an end unto itself. It's a tool that allows you to filter or aggregate with an expanded set of information. That means queries involving joins can get pretty complicated.

In order to get the exact results you're after, you need to be careful about exactly how you filter the data. As with joins, there are multiple points.

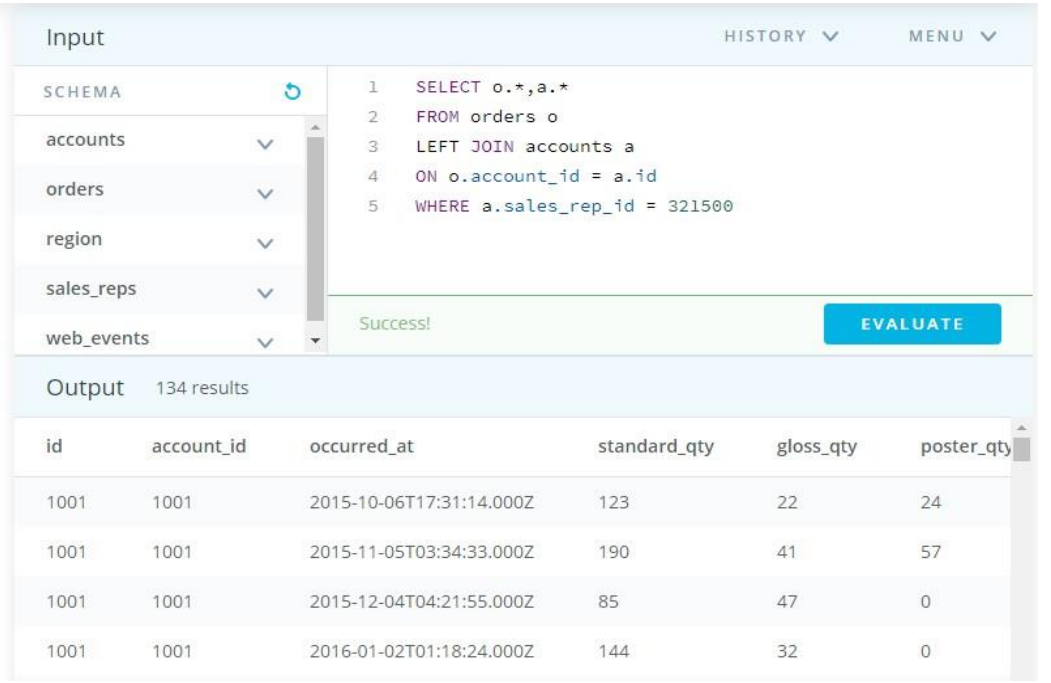
To illustrate this, let's take a look at all the orders that were brought in by one particular sales rep.

Example :

Imagine yourself as a sales manager at Parch & Posey. You'd like to make it easy for a sales rep to find her own deals rather than having to sift through all the orders and try and remember which were hers.

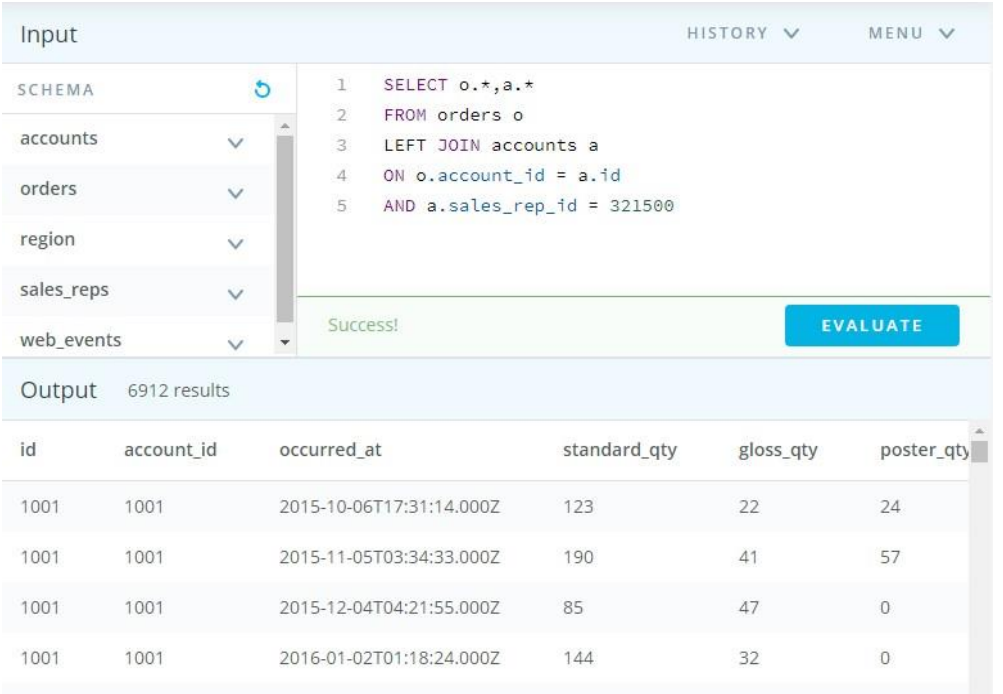
Remember, sales rep ID isn't in the orders table, so joins isn't necessary to get that information. The first way to filter this data is something we're already familiar with, the WHERE clause.

Let's take our combined orders and accounts tables and filter the results set to contain only the orders booked by sales rep 321500.



As you can see, the only rows in our result set are the rows that fit the criteria in the WHERE clause. The ones that have a sales rep ID equal to **321500**. But, what if we just want to mark all the orders that belong to rep **321500** and keep all the other orders in the result set as well ?

We can do that by filtering in the ON clause. Changing where to AND, we're moving this logical statement to become part of the ON clause.

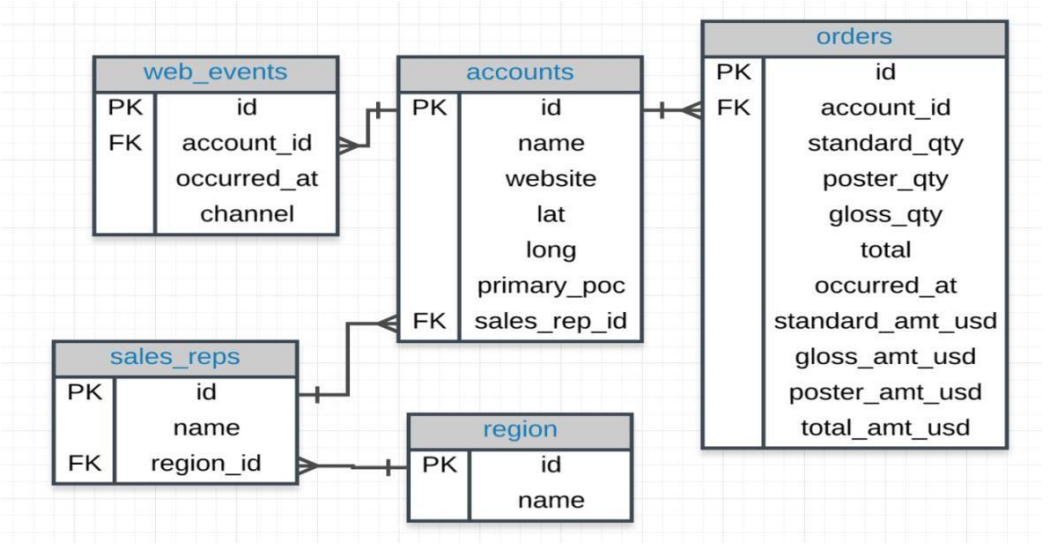


This effectively pre-filters the right table to include only rows with sales rep ID **321500** before the join is executed. In another words, it's like a WHERE clause that applies before the join rather than after. You can almost think of this like joining orders to a different table. One that only includes a subset of the rows in the original accounts table.

Since this is a left join, we'll see result set that includes all rows from the orders table as well as any data in this new pre-filtered table that match with the account ID in the orders table.

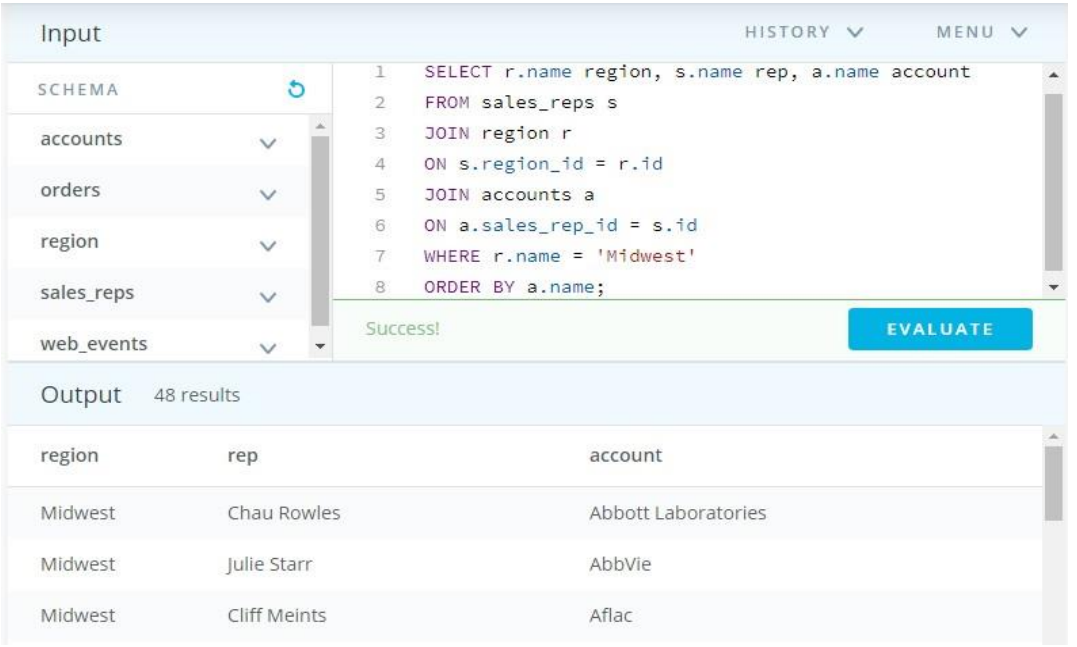
As you can see, sales rep 321500 only had a few accounts. So, a lot of these orders remain unmatched. It might seem frivolous to leave so many orders in this result set without account names. But, this type of join will be incredibly useful once you move on to data aggregation in the next lesson.

Quiz: Last Check

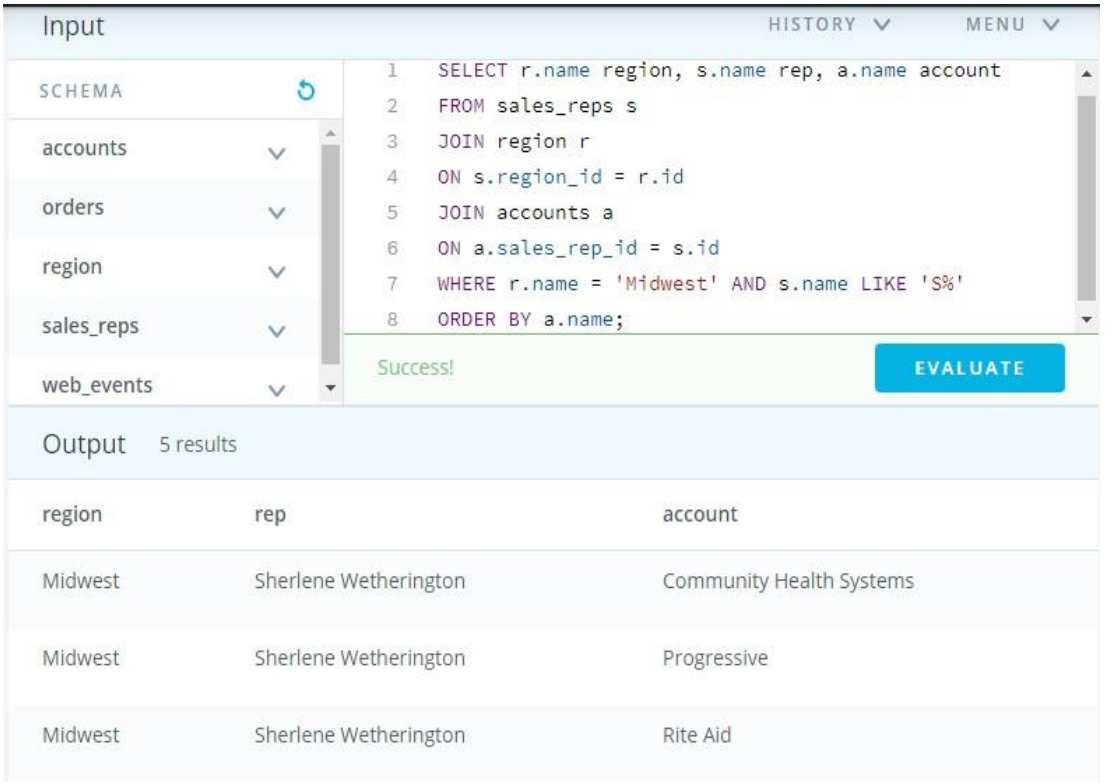


Questions

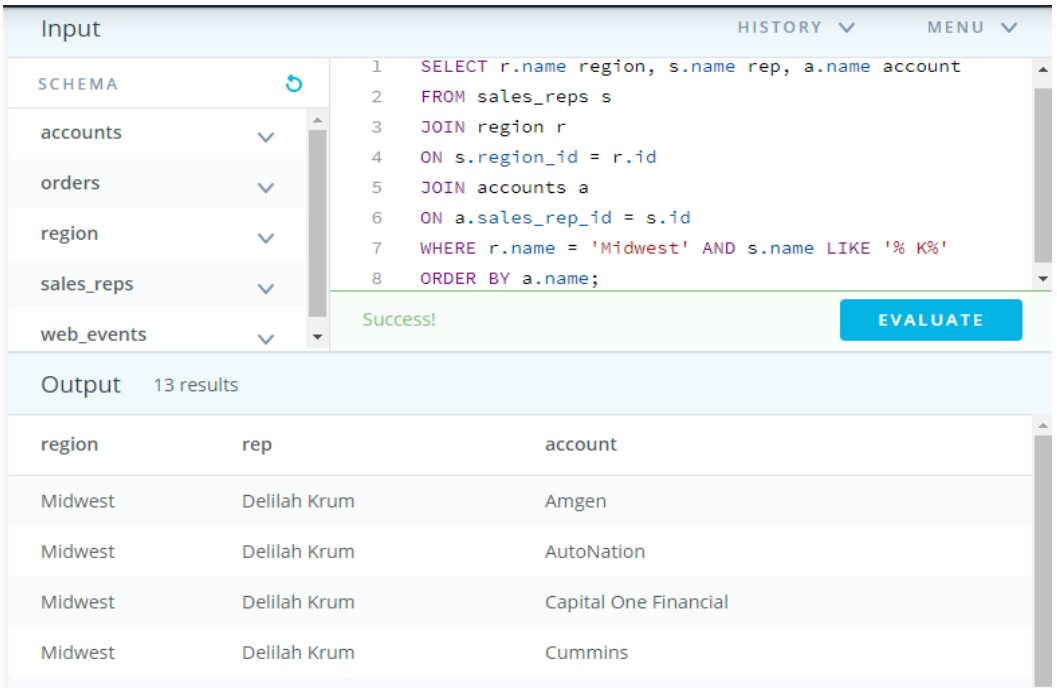
1. Provide a table that provides the **region** for each **sales_rep** along with their associated **accounts**. This time only for the Midwest region. Your final table should include three columns: the region **name**, the sales rep **name**, and the account **name**. Sort the accounts alphabetically (A-Z) according to account name.



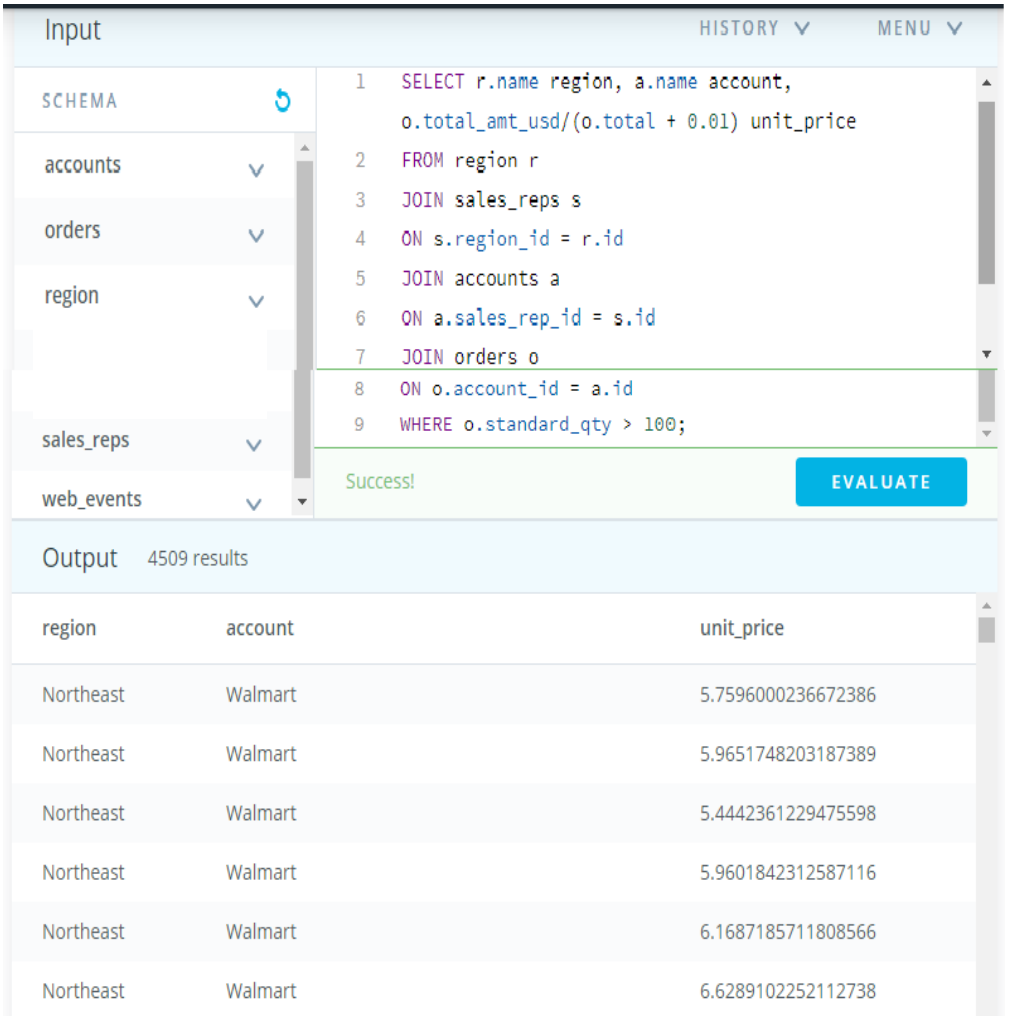
2. Provide a table that provides the **region** for each **sales_rep** along with their associated **accounts**. This time only for accounts where the sales rep has a first name starting with S and in the Midwest region. Your final table should include three columns: the region **name**, the sales rep **name**, and the account **name**. Sort the accounts alphabetically (A-Z) according to account name.



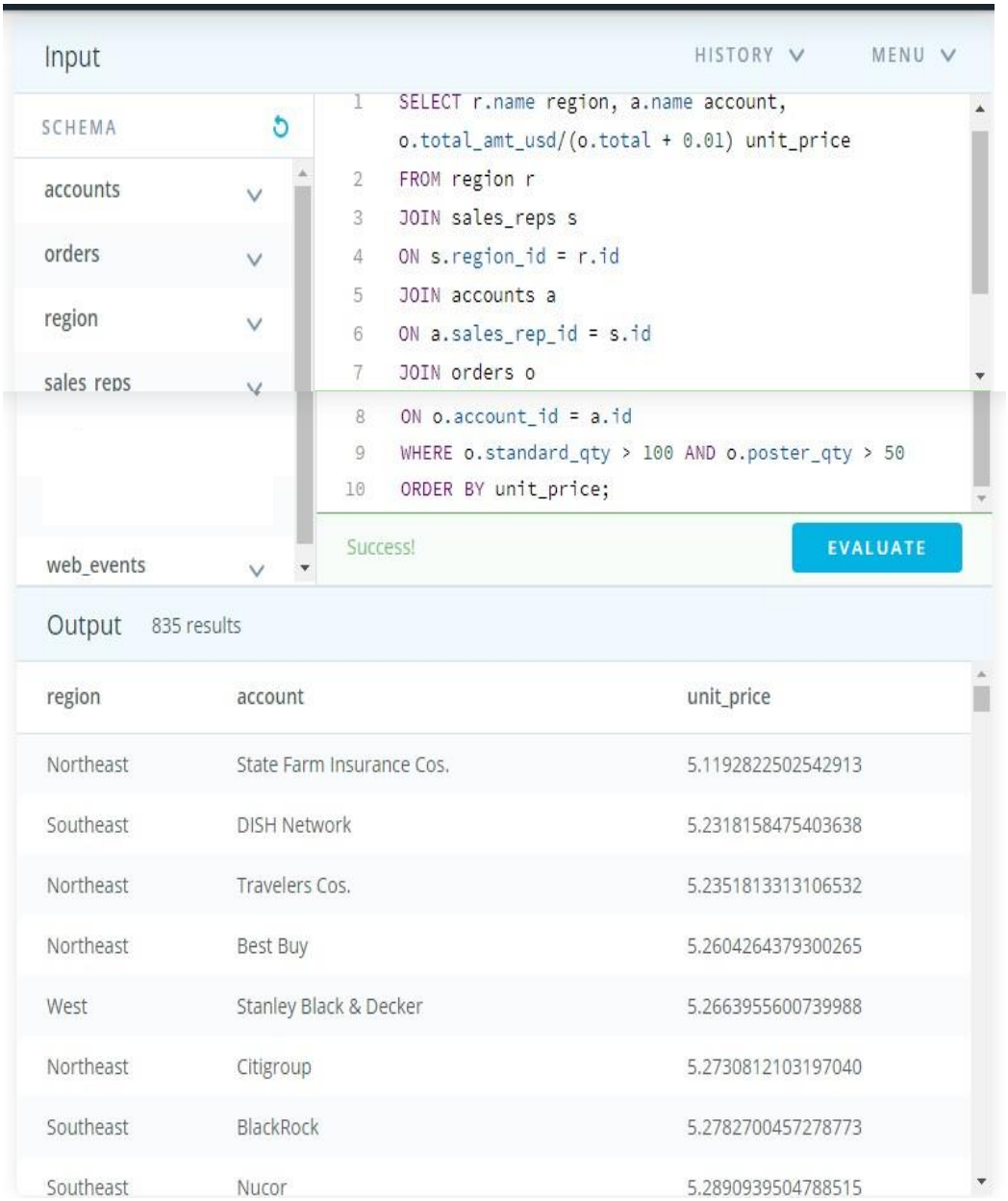
3. Provide a table that provides the **region** for each **sales_rep** along with their associated **accounts**. This time only for accounts where the sales rep has a **last** name starting with K and in the Midwest region. Your final table should include three columns: the region **name**, the sales rep **name**, and the account **name**. Sort the accounts alphabetically (A-Z) according to account name.



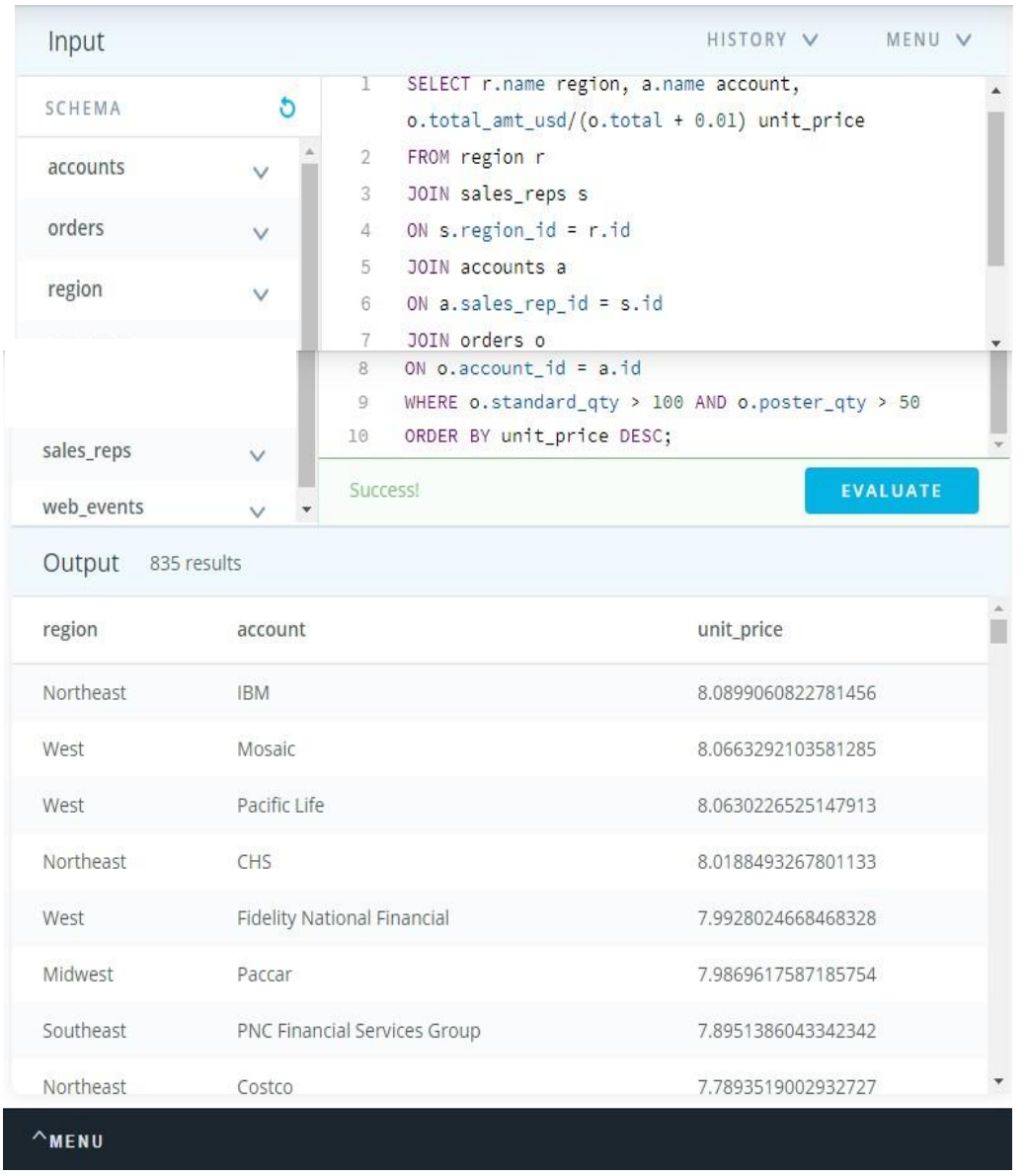
4. Provide the **name** for each region for every **order**, as well as the account **name** and the **unit price** they paid (total_amt_usd/total) for the order. However, you should only provide the results if the **standard order quantity**exceeds 100. Your final table should have 3 columns: **region name**, **account name**, and **unit price**. In order to avoid a division by zero error, adding .01 to the denominator here is helpful total_amt_usd/(total+0.01).



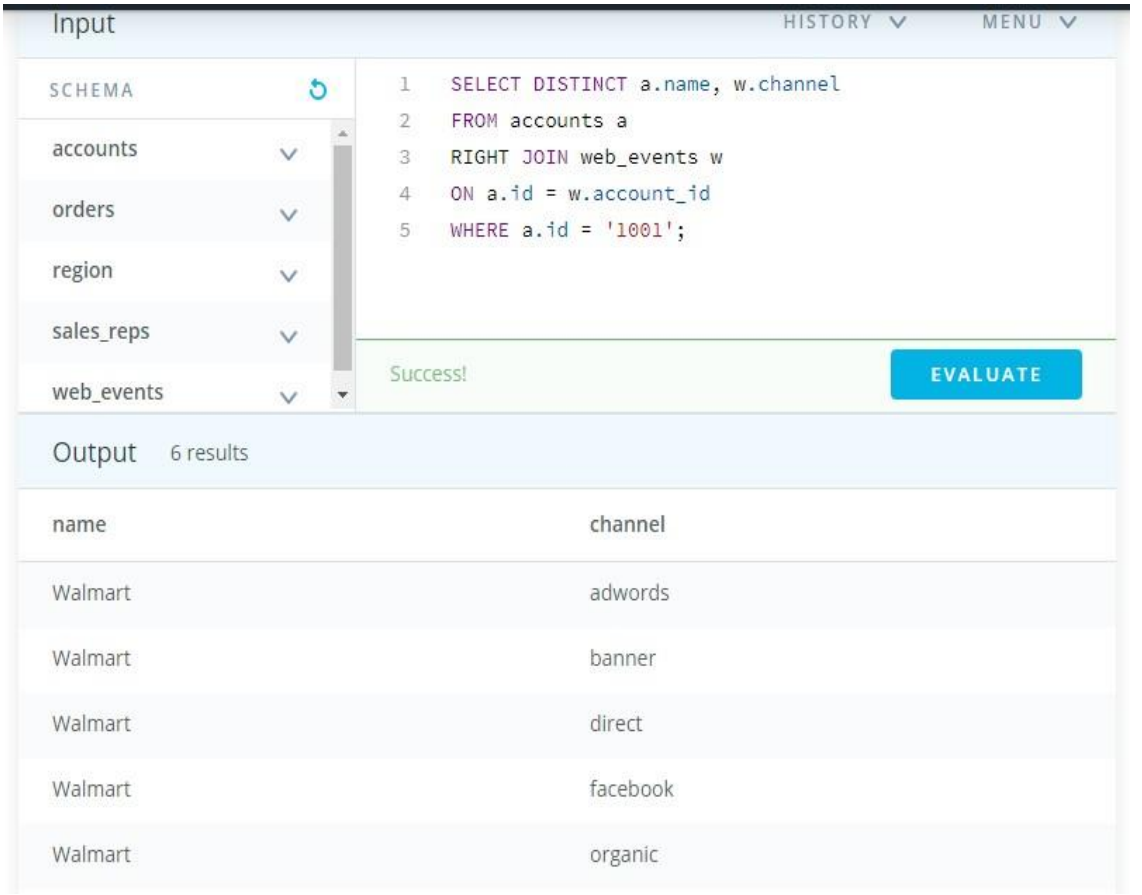
5. Provide the **name** for each region for every **order**, as well as the account **name** and the **unit price** they paid (total_amt_usd/total) for the order. However, you should only provide the results if the **standard order quantity**exceeds 100 and the **poster order quantity** exceeds 50. Your final table should have 3 columns: **region name**, **account name**, and **unit price**. Sort for the smallest **unit price** first. In order to avoid a division by zero error, adding .01 to the denominator here is helpful (total_amt_usd/(total+0.01).



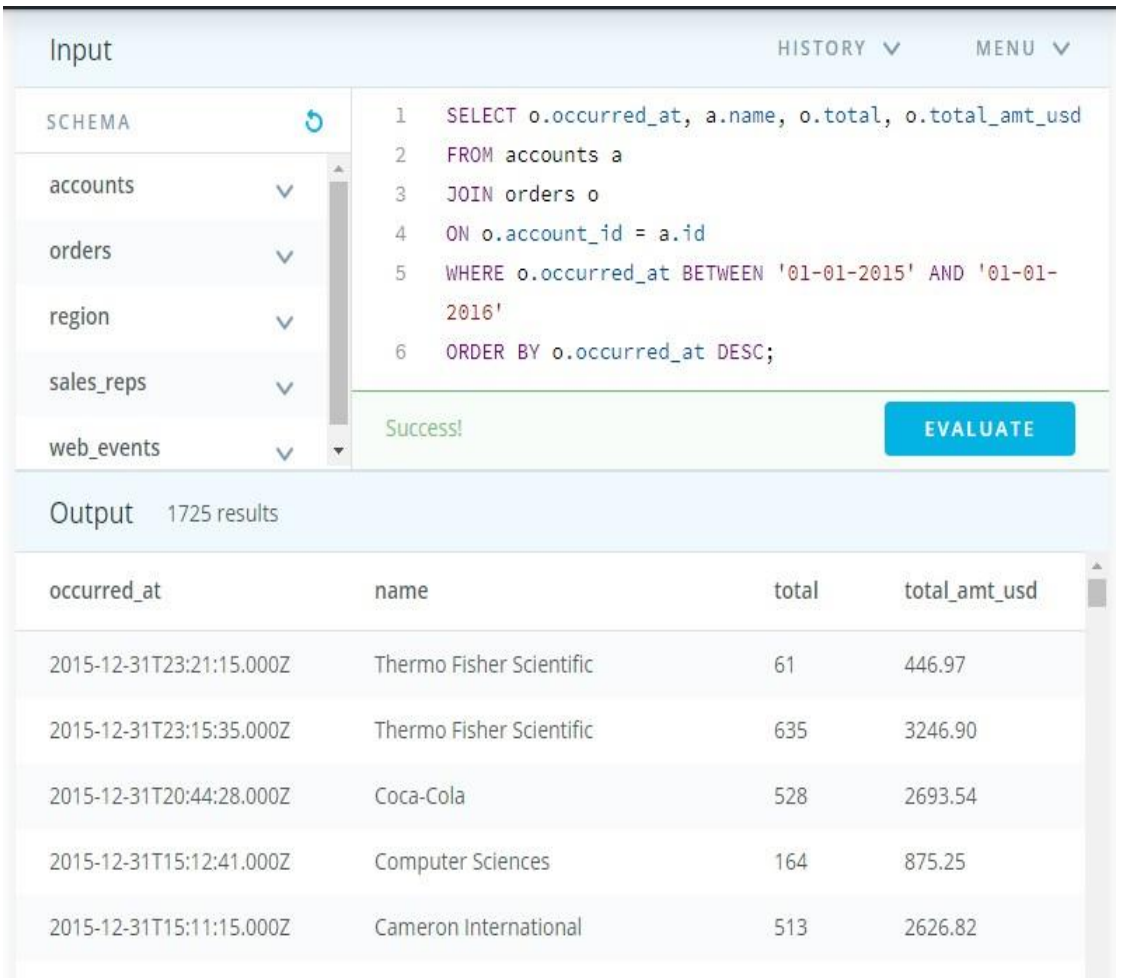
6. Provide the **name** for each region for every **order**, as well as the account **name** and the **unit price** they paid (total_amt_usd/total) for the order. However, you should only provide the results if the **standard order quantity**exceeds 100 and the **poster order quantity** exceeds 50. Your final table should have 3 columns: **region name**, **account name**, and **unit price**. Sort for the largest **unit price** first. In order to avoid a division by zero error, adding .01 to the denominator here is helpful (total_amt_usd/(total+0.01)



7. What are the different **channel**s used by **account id** 1001? Your final table should have only 2 columns: **account name** and the different **channel**s. You can try **SELECT DISTINCT** to narrow down the results to only the unique values.



8. Find all the orders that occurred in 2015. Your final table should have 4 columns: **occurred_at**, **account name**, **order total**, and **order total_amt_usd**.



Recap & Looking Ahead

Primary and Foreign Keys

You learned a key element for **JOIN**ing tables in a database has to do with primary and foreign keys:

- **primary keys** - are unique for every row in a table. These are generally the first column in our database (like you saw with the **id** column for every table in the Parch & Posey database).
- **foreign keys** - are the **primary key** appearing in another table, which allows the rows to be non-unique.

Choosing the set up of data in our database is very important, but not usually the job of a data analyst. This process is known as **Database Normalization**.

JOINs

In this lesson, you learned how to combine data from multiple tables using **JOIN**s. The three **JOIN** statements you are most likely to use are:

1. **JOIN** - an **INNER JOIN** that only pulls data that exists in both tables.
2. **LEFT JOIN** - a way to pull all of the rows from the table in the **FROM** even if they do not exist in the **JOIN** statement.
3. **RIGHT JOIN** - a way to pull all of the rows from the table in the **JOIN** even if they do not exist in the **FROM** statement.

There are a few more advanced **JOIN**s that we did not cover here, and they are used in very specific use cases. UNION and UNION ALL, CROSS JOIN, and the tricky SELF JOIN. These are more advanced than this course will cover, but it is useful to be aware that they exist, as they are useful in special cases.

Alias

You learned that you can alias tables and columns using **AS** or not using it. This allows you to be more efficient in the number of characters you need to write, while at the same time you can assure that your column headings are informative of the data in your table.

Looking Ahead

The next lesson is aimed at **aggregating** data. You have already learned a ton, but **SQL** might still feel a bit disconnected from **statistics** and using **Excel** like platforms. Aggregations will allow you to write **SQL** code that will allow for more complex queries, which assist in answering questions like:

- Which **channel** generated more revenue?
- Which **account** had an order with the most items?
- Which **sales_rep** had the most orders? or least orders? How many orders did they have?

LESSON 28

Introduction to Aggregation

In the following concepts you will be learning in detail about each of the aggregate functions mentioned as well as some additional aggregate functions that are used in SQL all the time.

So far in this course, we've used SQL to join tables using left, right and inner joins, and filter the results using Where and On clauses.

Sometimes, it's useful to view row-level data. For example, we might want to look at all the individual orders for a given sales person to see how they're performing in a given month. But other times, row-level data is overwhelming, and ultimately, less valuable than aggregated data. Counting up all the orders in each region in each month, for example, will be pretty inconvenient when looking at row-level data.

One rep's order volume is low enough to look at individual rows, but the entire region's volume is way too high. Instead, a single number of orders for each month will be much better.

Fortunately, databases are great at aggregating data. The SQL syntax is easy to learn. If you're familiar with Excel, SQL commands are pretty similar.

- Count, counts how many rows are in a particular column.
- Sum, adds together all the values in a particular column.
- Min and Max, return the lowest and highest values in a particular column.
- Average, calculates the average of all the values in a particular column.

These functions operate down columns, not across rows. So you can do things like sum up all the quantities of paper over delivered. Instead of getting results set many thousands of rows long, you just get one line with the answer.

In practice, you'll find yourself using row-level output for the early exploratory work when searching your database to better understand the data.

As you get a sense for what the data looks like and begin to look for answer to your questions, aggregates become more helpful.

Introduction to NULLs

NULLs are a data type that specifies where no data exists in SQL. They are often ignored in our aggregation functions, which you will get a first look at in the next concept using **COUNT**.

Before we dive into aggregation, we need to cover one more concept, NULLs.

NULL means no data. It's different from a zero or a space. And there's a good reason. From a business perspective, for Parch & Posey a zero means that no paper was sold, which cloud mean that a sale was attempted but not made.

A null could mean no sale was even attempted, and that's a pretty meaningful difference.

Those could be the result of any number of things, they can be part of the data design itself or they can simply be the result of poorly looked data.

Regardless, it's important to understand what nulls mean and to know how to work with them.

NULLs and Aggregation

Notice that **NULL**s are different than a zero - they are cells where data does not exist. When identifying **NULL**s in a **WHERE** clause, we write **IS NULL** or **IS NOT NULL**. We don't use =, because **NULL** isn't considered a value in SQL. Rather, it is a property of the data.

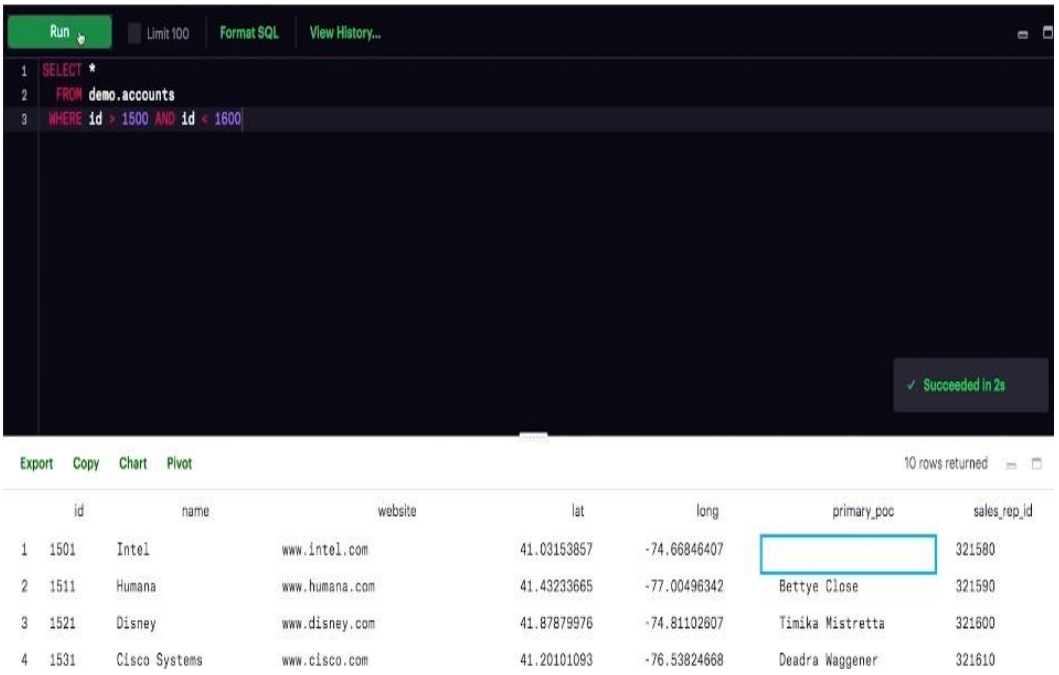
NULLs - Expert Tip

There are two common ways in which you are likely to encounter **NULL**s:

- **NULL**s frequently occur when performing a **LEFT** or **RIGHT JOIN**. You saw in the last lesson - when some rows in the left table of a left join are not matched with rows in the right table, those rows will contain some **NULL** values in the result set.
- **NULL**s can also occur from simply missing data in our database.

Example:

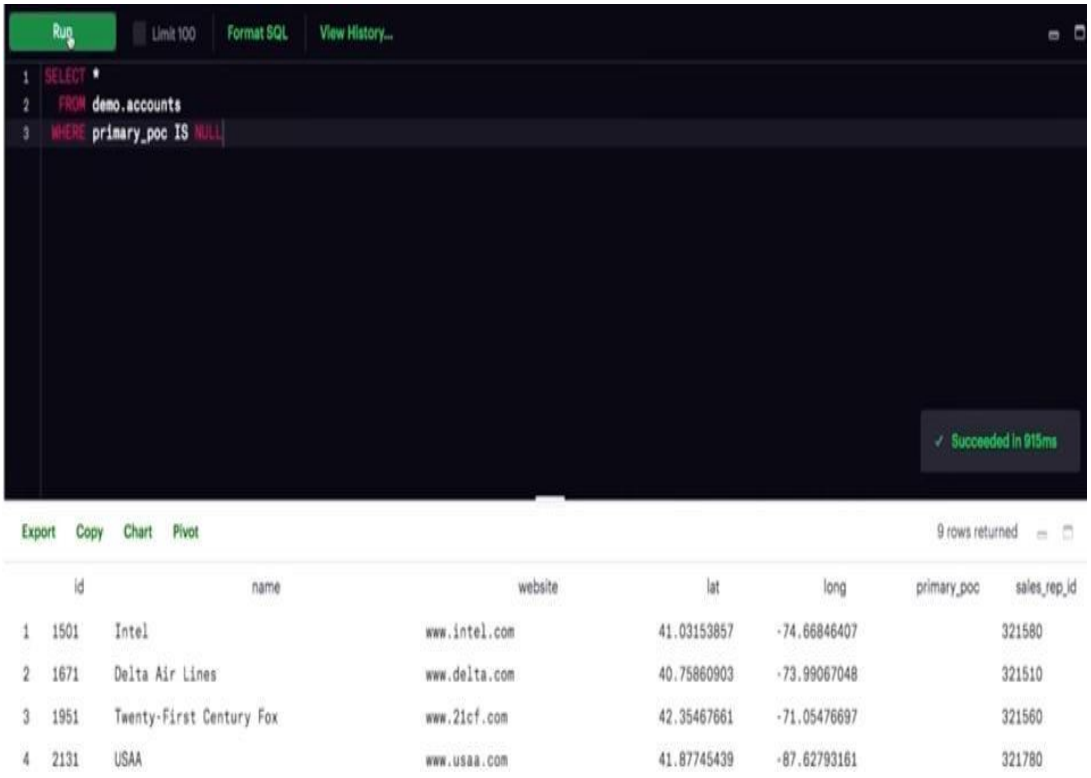
To get a feel for how nulls work, let's pull a set of records from the accounts table with IDs that fall between 1500 and 1600.



	id	name	website	lat	long	primary_poc	sales_rep_id
1	1501	Intel	www.intel.com	41.03153857	-74.66846407		321580
2	1511	Humana	www.humana.com	41.43233665	-77.00496342	Bettye Close	321590
3	1521	Disney	www.disney.com	41.87879976	-74.81102607	Timika Mistretta	321600
4	1531	Cisco Systems	www.cisco.com	41.20101093	-76.53824668	Deadra Waggener	321610

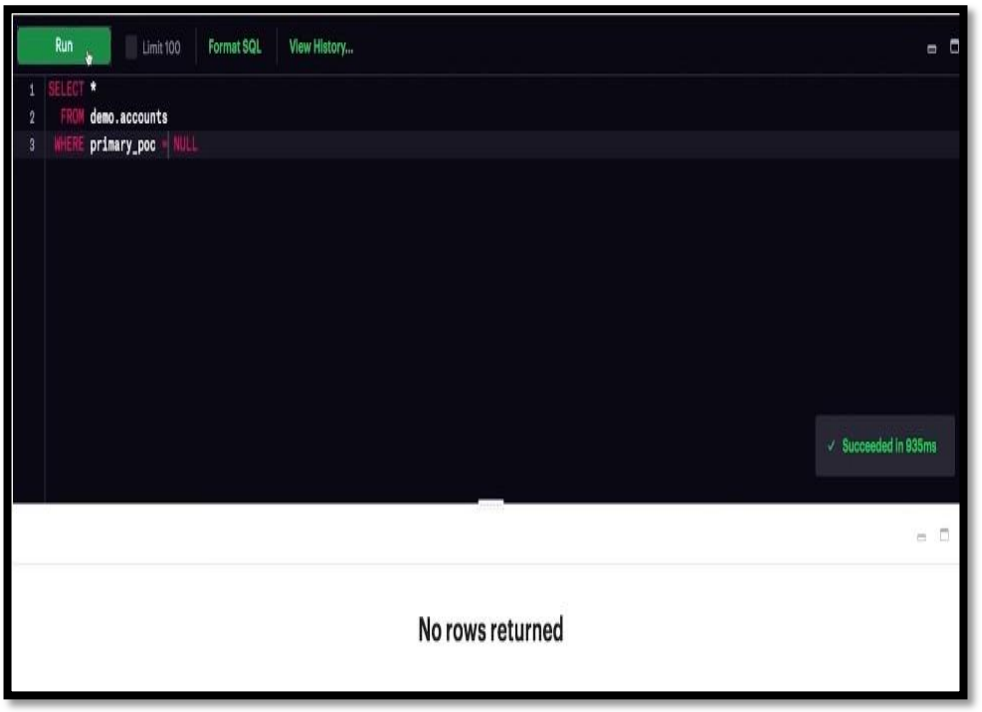
We can see that the primary_poc, which stands for point of contact, is blank for the Intel account. This could simply be an error in the data. Maybe a point of contact that was accidentally deleted at some point, or it could be that Parch & Posey's point of contact left the company and they don't yet have a new point of contact for that account. Either way, there is no data in this particular cell. This cell is null.

Imagining yourself as a sales manager at Parch & Posey, you may want to know all of the accounts for which the primary_poc is null. If you don't have a point of contact, chances are you're not going to be able to keep that customer for much longer. In order to find all these accounts, we'll have to use some special syntax in our WHERE clause.



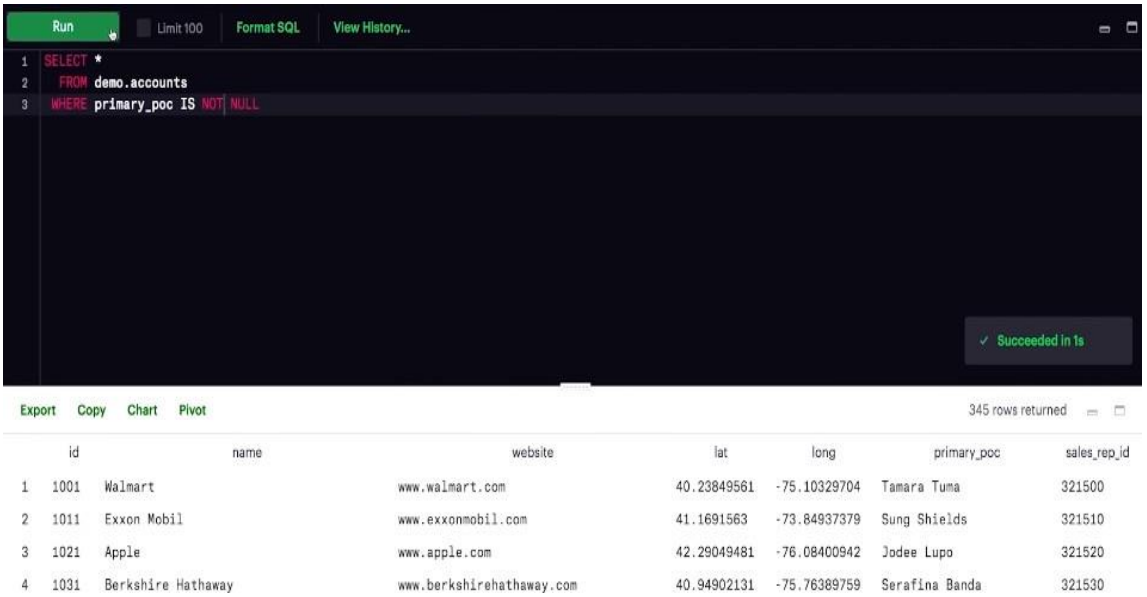
	id	name	website	lat	long	primary_poc	sales_rep_id
1	1501	Intel	www.intel.com	41.03153857	-74.66846407		321580
2	1671	Delta Air Lines	www.delta.com	40.75860903	-73.99067048		321510
3	1951	Twenty-First Century Fox	www.21cf.com	42.35467661	-71.05476697		321560
4	2131	USAA	www.usaa.com	41.87745439	-87.62793161		321780

Turns out, there are quite a few accounts without points of contact. So, the special thing about nulls that you need to write is NULL instead of EQUALS NULL.



The reason this doesn't work is that NULL is not a value, it's a property of the data. This is different from zero or a space, both of which are values.

If you want to find the inverse of our previous result set, you can use this syntax, is not NULL.



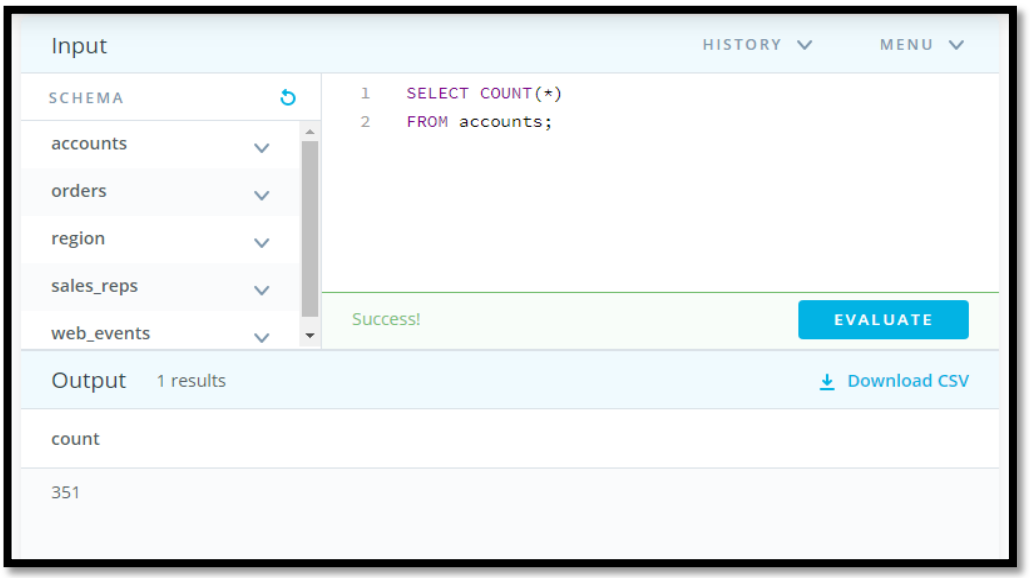
	id	name	website	lat	long	primary_poc	sales_rep_id
1	1001	Walmart	www.walmart.com	40.23849561	-75.10329704	Tamara Tuma	321500
2	1011	Exxon Mobil	www.exxonmobil.com	41.1691563	-73.84937379	Sung Shields	321510
3	1021	Apple	www.apple.com	42.29049481	-76.08400942	Jodee Lupo	321520
4	1031	Berkshire Hathaway	www.berkshirehathaway.com	40.94902131	-75.76389759	Serafina Banda	321530

As you can see, this results all rows for which there are values in the primary_poc column.

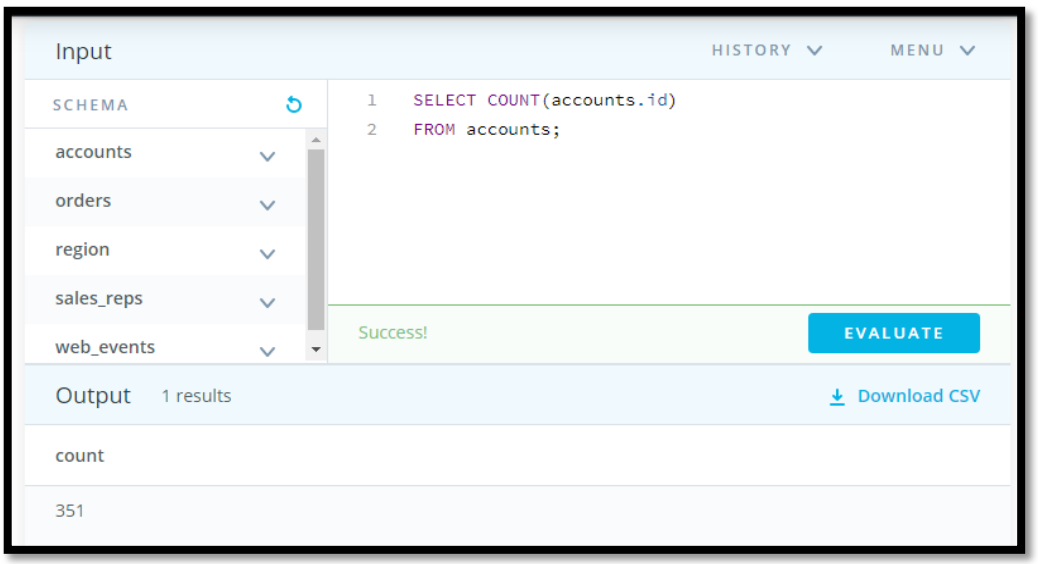
First Aggregation -COUNT

COUNT the Number of Rows in a Table

Try your hand at finding the number of rows in each table. Here is an example of finding all the rows in the **accounts** table.



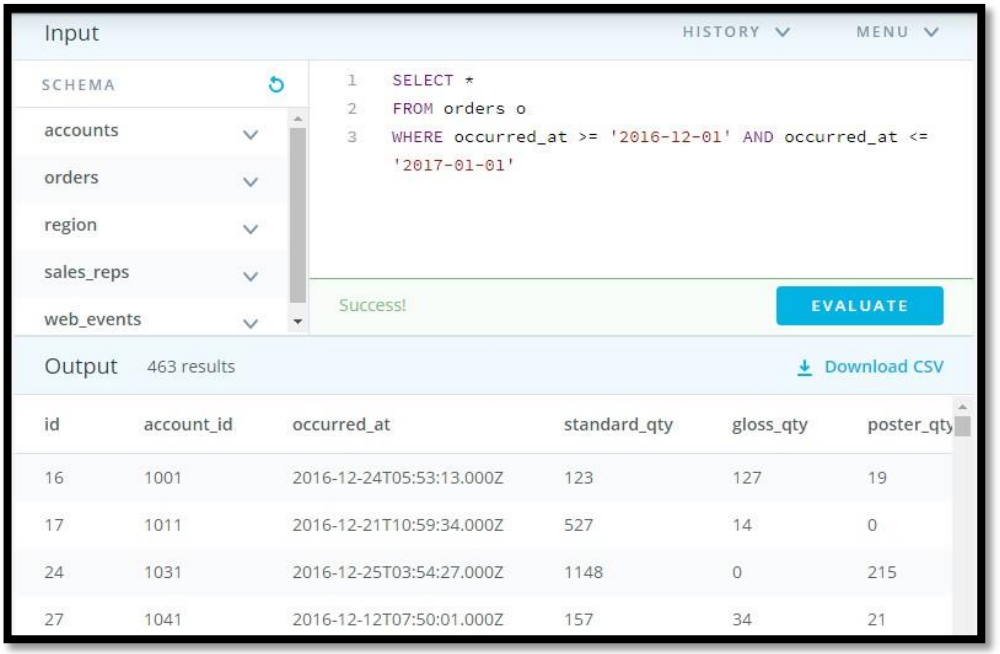
But we could have just as easily chosen a column to drop into the aggregation function:



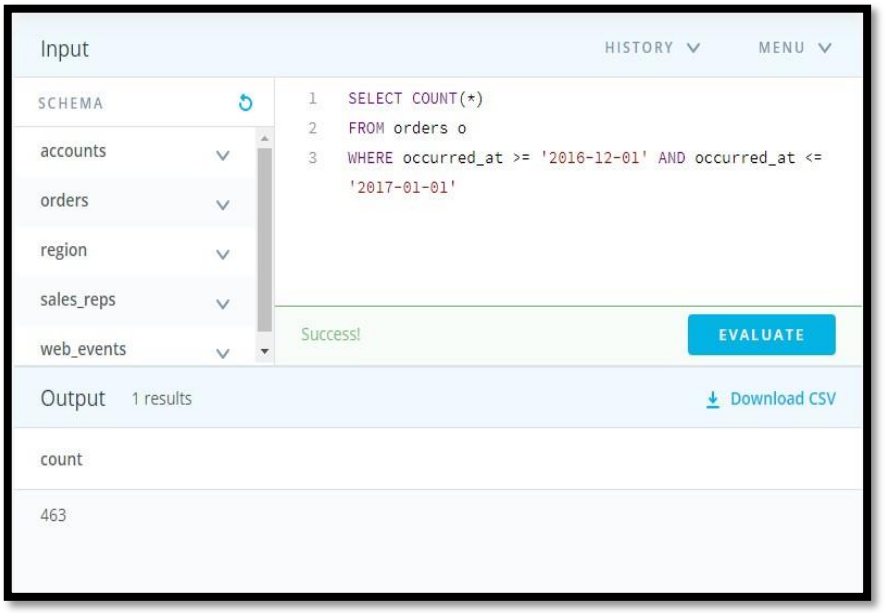
These two statements are equivalent, but this isn't always the case.

Example :

Here's a list of orders in the last month, assuming it's currently January 2017. I'm using mode to run these queries, which tells me that there are 463 rows in my result set.

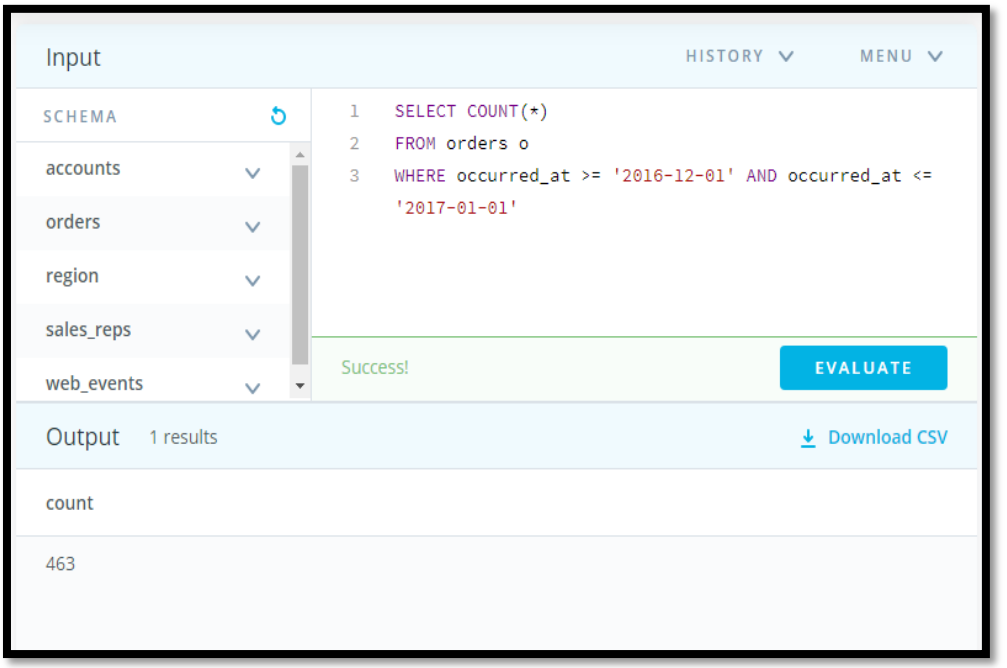


Still, you'll need to learn to count in SQL, because you can use it count in a much more fine-grained way than simply counting all the rows in a result set.
We can do this by modifying our SELECT statement to use a COUNT function.



You can see that this returns a single row of output in a single column, and the value is exactly the same as the row COUNT we saw before.

Let's give that column an alias, so that it's a little easier to read.

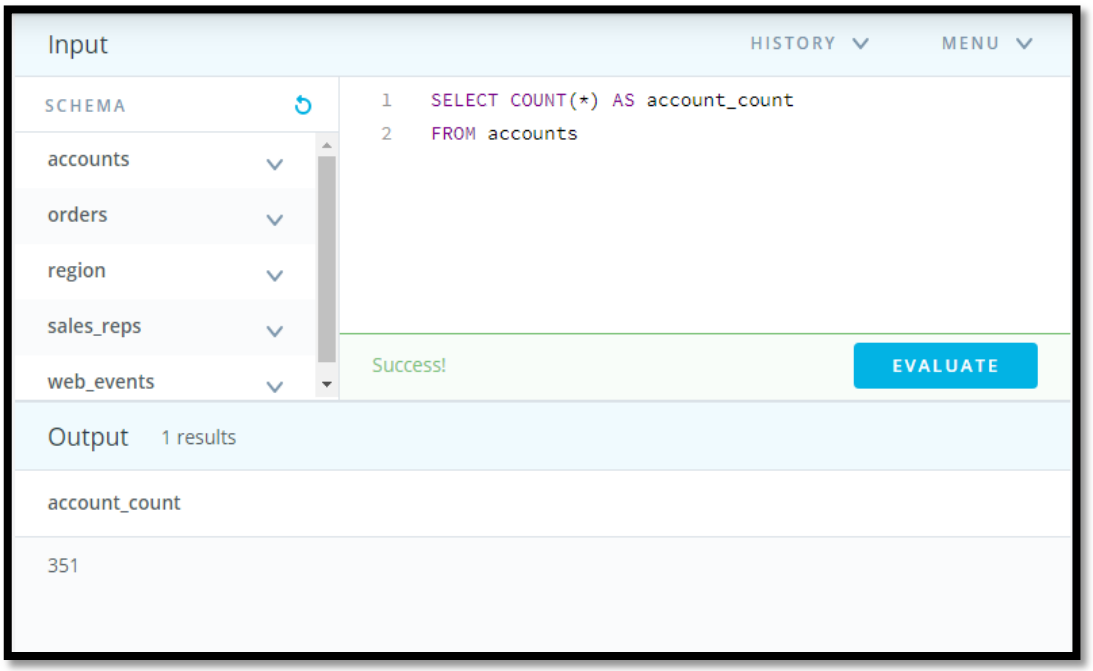


The COUNT function is returning a count of all the rows that contain some non-null data. It's very unusual to have a row that is entirely null. So, the result produced by a **COUNT(*)** is typically equal to the number of rows in the table.

COUNT & NULLs

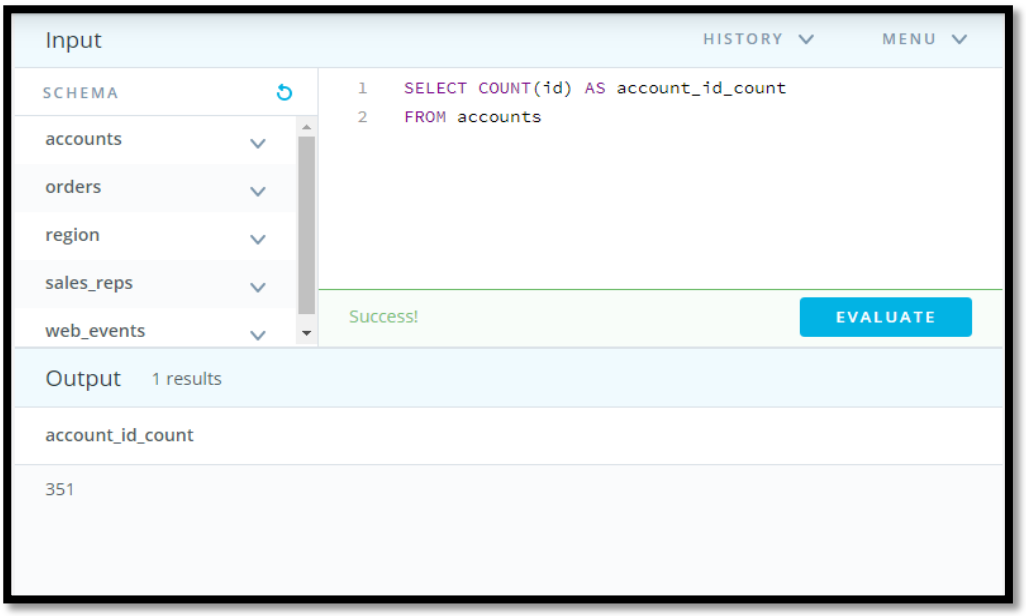
Notice that **COUNT** does not consider rows that have **NULL** values. Therefore, this can be useful for quickly identifying which rows have missing data. You will learn **GROUP BY** in an upcoming concept, and then each of these aggregators will become much more useful.

The COUNT function can also be used to count the number of non-null records in an individual column. To illustrated this, let's take a look at the accounts table.



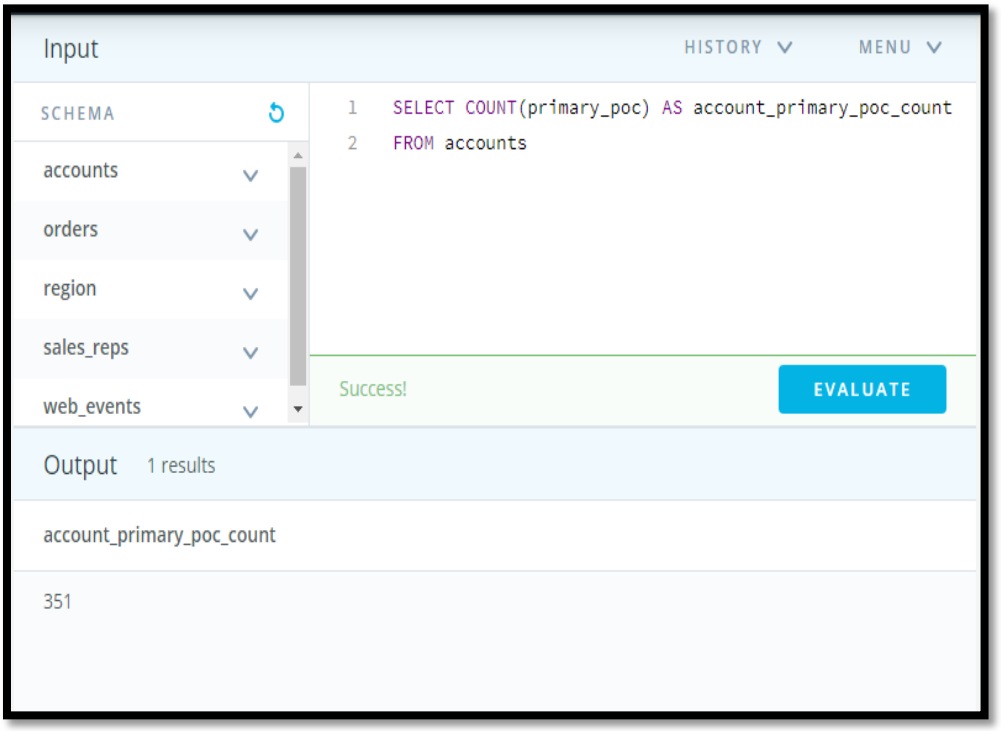
As you can see, there are 354 rows in the accounts table. Now, substituting ID for the star, we can see how many non-nulls records there are in the ID column.

Let's give the column a more sensible name while we are at it.



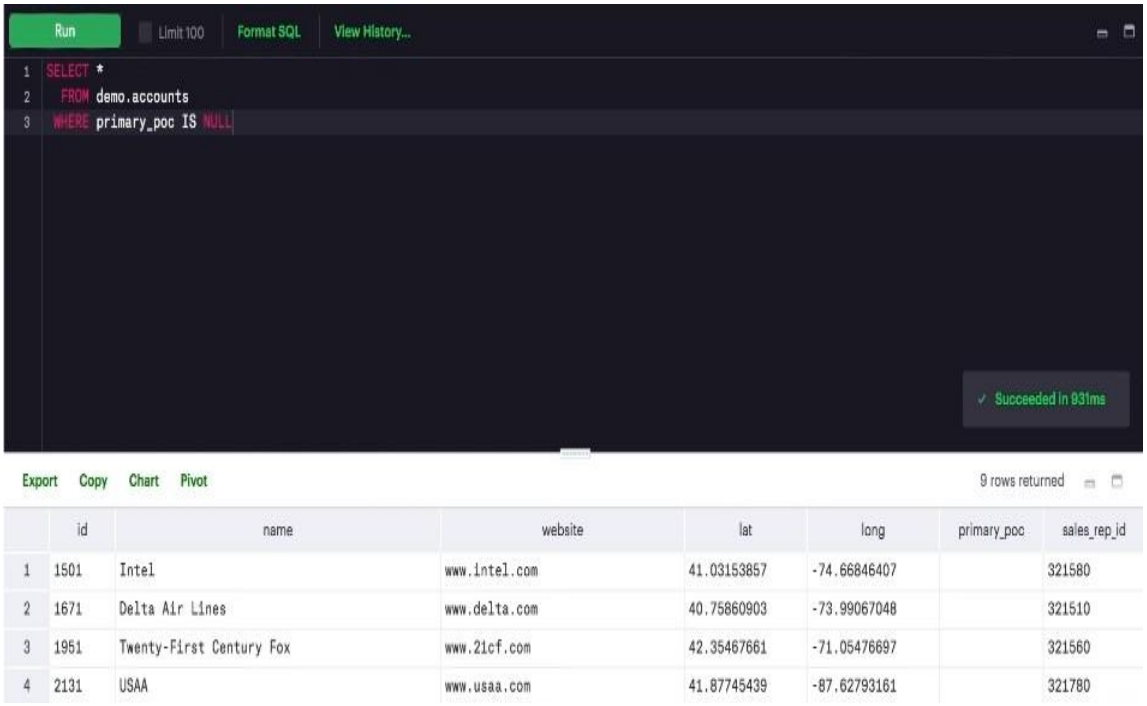
Since there are no null values in the ID column, it return the same result as count star.

Now, let's try this with a column that we know contains some null values, primary_poc.



This time, we get a result than is nine lower than our previous result.

Let's run a quick query to verify that there are nine null values in the primary_poc column.



	id	name	website	lat	long	primary_poc	sales_rep_id
1	1501	Intel	www.intel.com	41.03153857	-74.66846407		321580
2	1671	Delta Air Lines	www.delta.com	40.75860903	-73.99067048		321510
3	1951	Twenty-First Century Fox	www.21cf.com	42.35467661	-71.05476697		321560
4	2131	USAA	www.usaa.com	41.87745439	-87.62793161		321780

And here they are, the nine rows with null values. One thing you may have noticed is that count can be used in columns with non numerical values. This is not true of all aggregation function, but it make sense.

The count function is just looking for non-null data and text is not null. Some of the functions we're about to learn like sum and average are impossible to apply to next because, well, how do you take the average of a bunch of account names.

SUM

Unlike **COUNT**, you can only use **SUM** on numeric columns. However, **SUM** will ignore **NULL** values, as do the other aggregation functions you will see in the upcoming lessons.

Aggregation Reminder

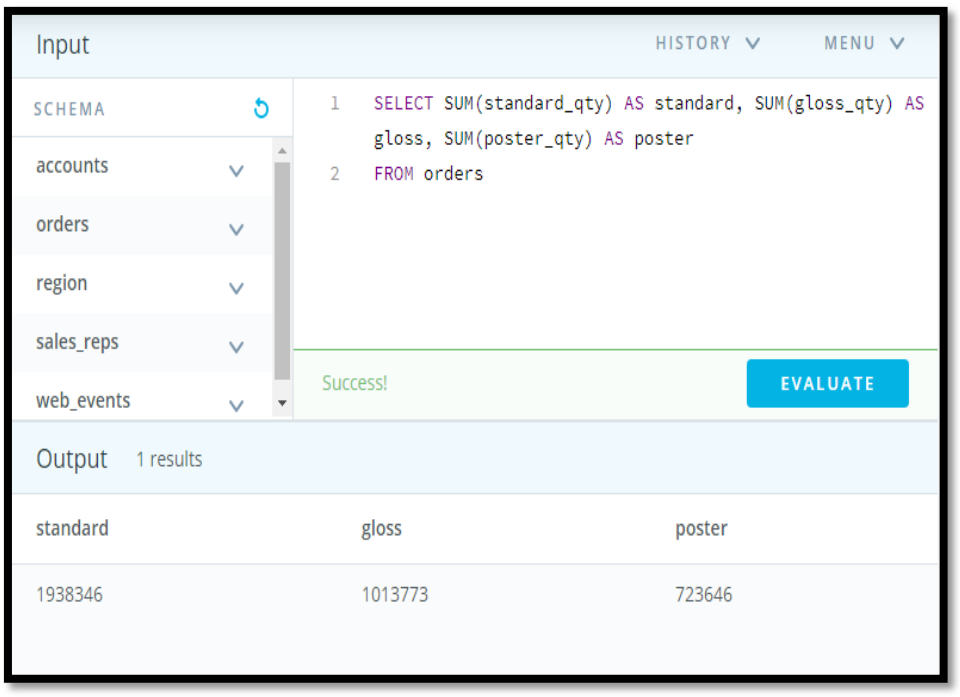
An important thing to remember: **aggregators only aggregate vertically - the values of a column**. If you want to perform a calculation across rows.

We saw this in the first lesson if you need a refresher, but the quiz in the next concept should assure you still remember how to aggregate across rows.

Example :

Imagine yourself as an operations manager at Parch & Posey. You're trying to do some inventory planning, and you want to know how much of each paper type of produce.

A good place to start might be to total up all sales of each paper type and compare them to one another. We'll do this using SUM. It works similar to COUNT, except that you'll wont to specify column names rather than using star.



It looks like standard is more popular than both of the non-standard paper types combined.

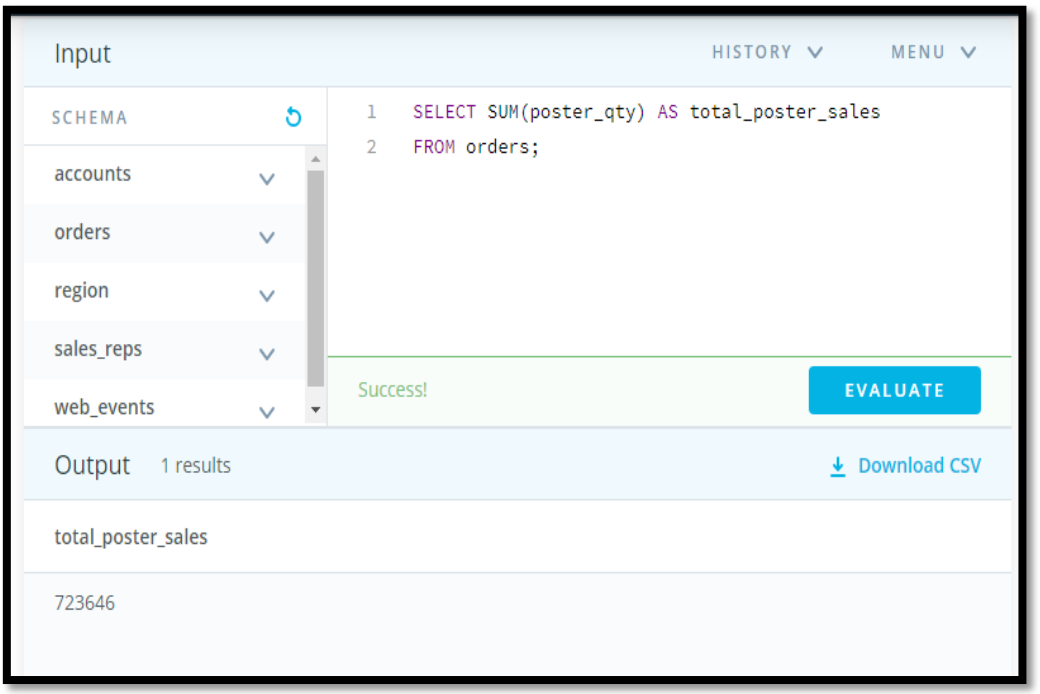
Unlike COUNT, you can only use the SUM function on columns containing numerical values. You don't need to worry too much about the presence of nulls. The SUM function will just start nulls as zero.

Quiz: SUM

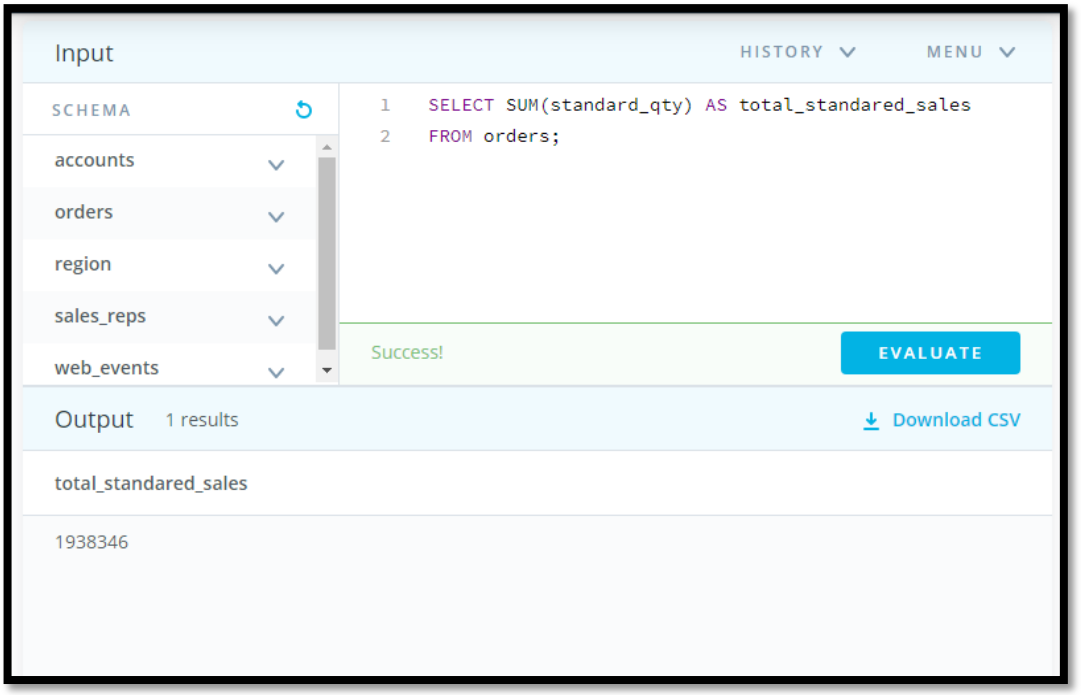
Aggregation Questions

Use the **SQL** environment below to find the solution for each of the following questions. If you get stuck or want to check your answers, you can find the answers at the top of the next concept.

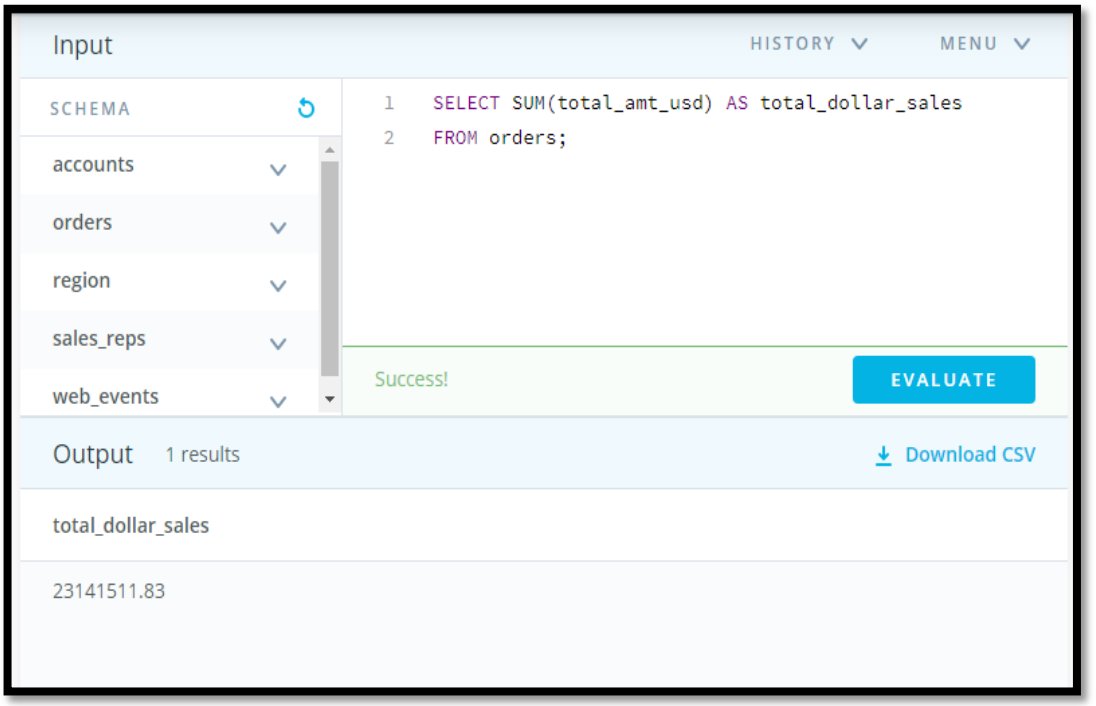
1. Find the total amount of **poster_qty** paper ordered in the **orders** table.



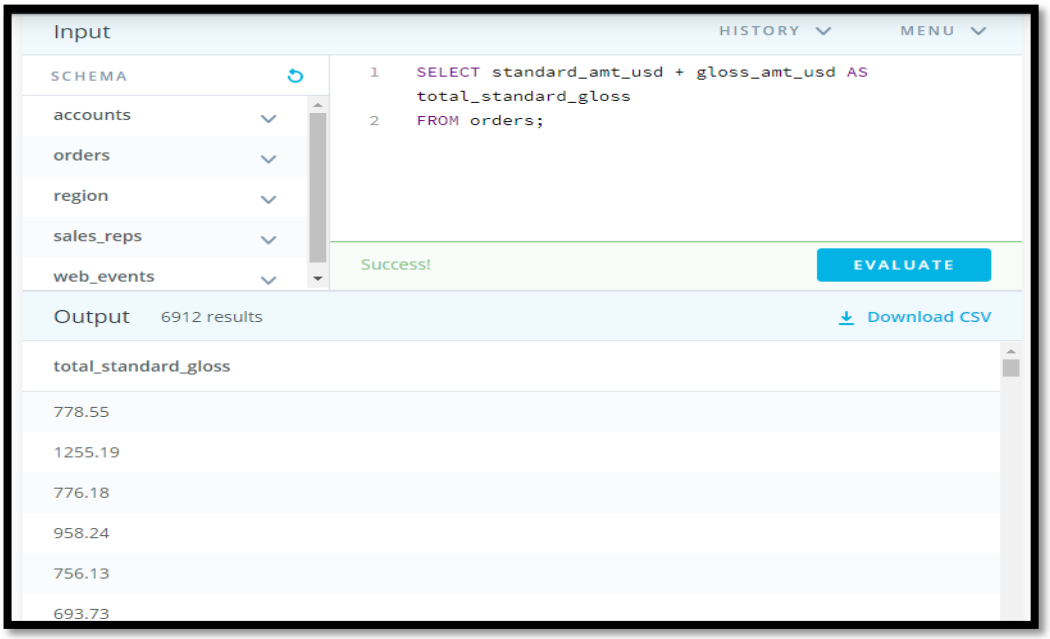
2. Find the total amount of **standard_qty** paper ordered in the **orders** table.



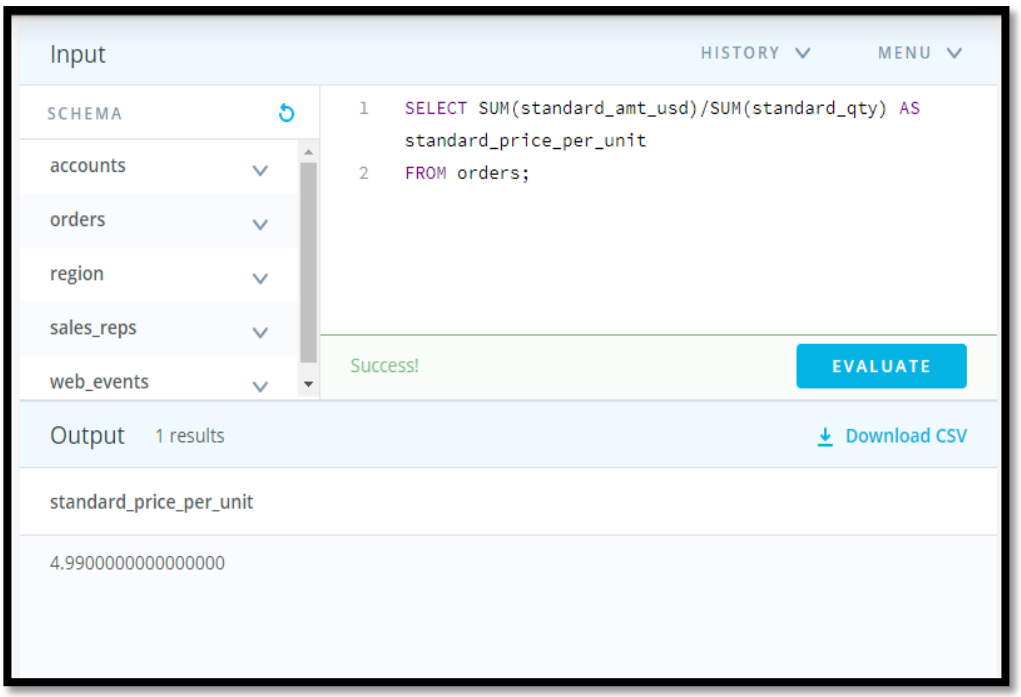
3. Find the total dollar amount of sales using the **total_amt_usd** in the **orders** table.



4. Find the total amount spent on **standard_amt_usd** and **gloss_amt_usd** paper for each order in the orders table. This should give a dollar amount for each order in the table.



5. Find the **standard_amt_usd** per unit of **standard_qty** paper. Your solution should use both an aggregation and a mathematical operator.



MIN & MAX

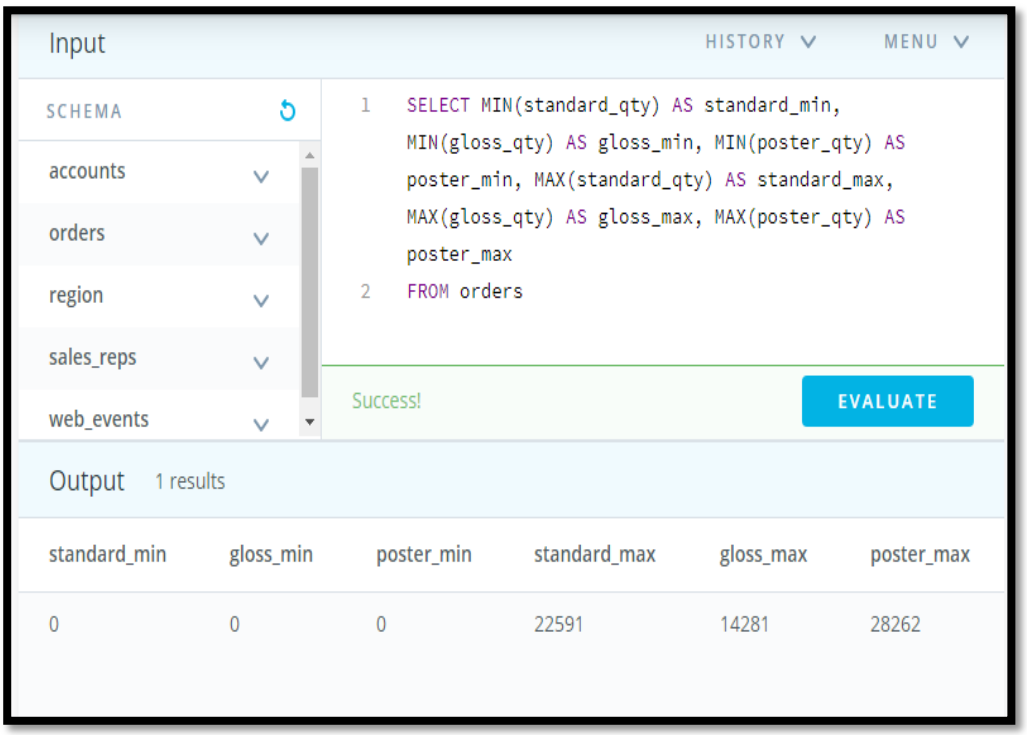
Notice that here we were simultaneously obtaining the **MIN** and **MAX** number of orders of each paper type. However, you could run each individually.

Notice that **MIN** and **MAX** are aggregators that again ignore **NULL** values.

Expert Tip

Functionally, **MIN** and **MAX** are similar to **COUNT** in that they can be used on non-numerical columns. Depending on the column type, **MIN** will return the lowest number, earliest date, or non-numerical value as early in the alphabet as possible. As you might suspect, **MAX** does the opposite—it returns the highest number, the latest date, or the non-numerical value closest alphabetically to —Z.‖

The syntax for min & max is similar to SUM & COUNT. It shouldn't be surprising that the minimum for each paper type is zero.



Some customers only order one or two types of paper. What is surprising is that the largest single order is for poster paper, despite the fact that it's the least popular overall.

There are important implications here. Even though it's not the most popular item, Parch & Posey may want to produce enough poster paper to be able to fulfill pretty big orders at any given time.

AVG

Similar to other software **AVG** returns the mean of the data - that is the sum of all of the values in the column divided by the number of values in a column. This aggregate function again ignores the **NULL** values in both the numerator and the denominator.

If you want to count **NULL**s as zero, you will need to use **SUM** and **COUNT**. However, this is probably not a good idea if the **NULL** values truly just represent unknown values for a cell.

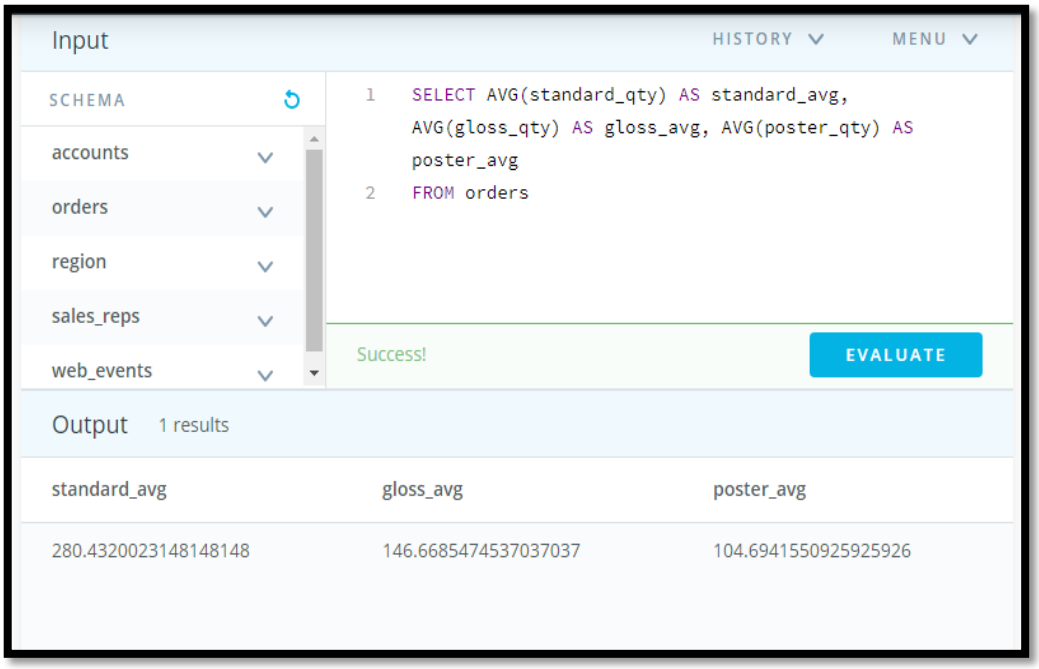
MEDIAN - Expert Tip

One quick note that a median might be a more appropriate measure of center for this data, but finding the median happens to be a pretty difficult thing to get using SQL alone — so difficult that finding a median is occasionally asked as an interview question.

So now we know which paper types are most are most popular and we have a sense of largest order size we might need to fulfill at any given time. But what's the average order size?

What can we expect to see on a regular bases ?

We'll use the average function which is typed as **AVG** and has a similar syntax to all of the other aggregation functions.



So, it's look like that large poster paper order most have been a major outlier because the average poster order is only about a third the size of the average standard order.

When you're using the average function, keep in mind that it can only be used on numerical columns. Also, ignores nulls completely, meaning that rows with null values are not counted in the numerator or the denominator when calculating the average.

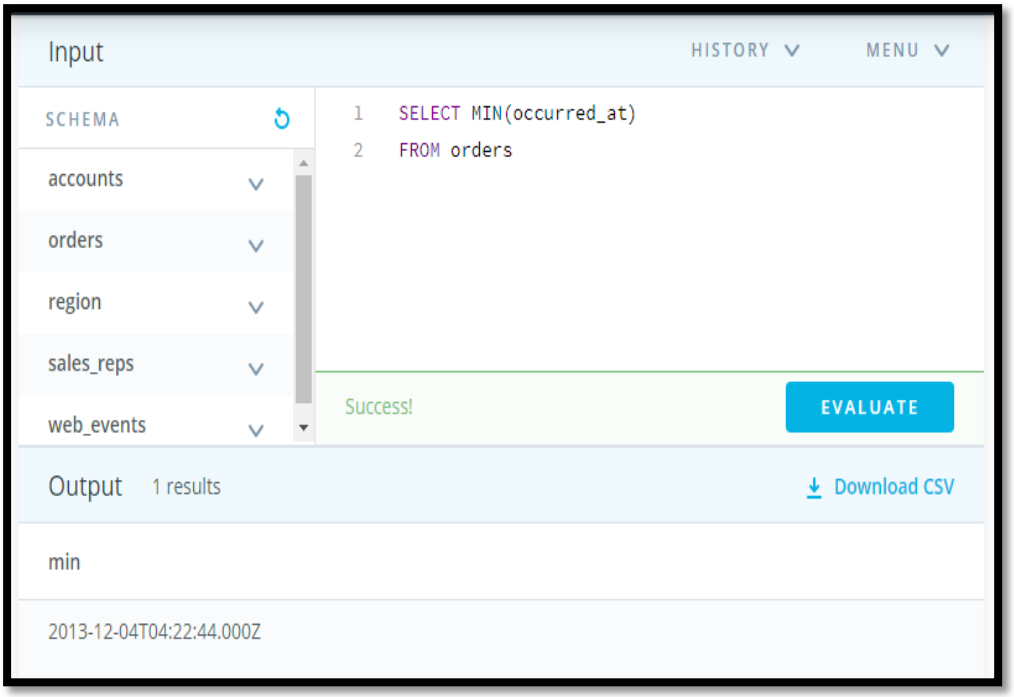
If you want to treat nulls as zero, you'll need to take a sum and divide it by the count rather than just using the average function.

Quiz: MIN, MAX, & AVG

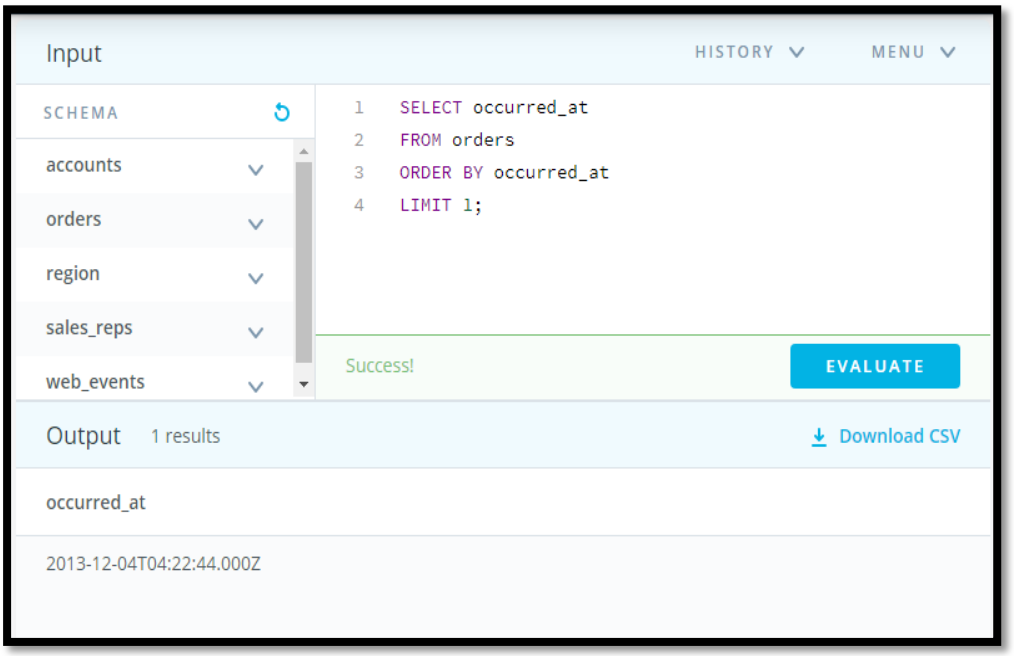
Questions: MIN, MAX, & AVERAGE

Use the **SQL** environment below to assist with answering the following questions. Whether you get stuck or you just want to double check your solutions, my answers can be found at the top of the next concept.

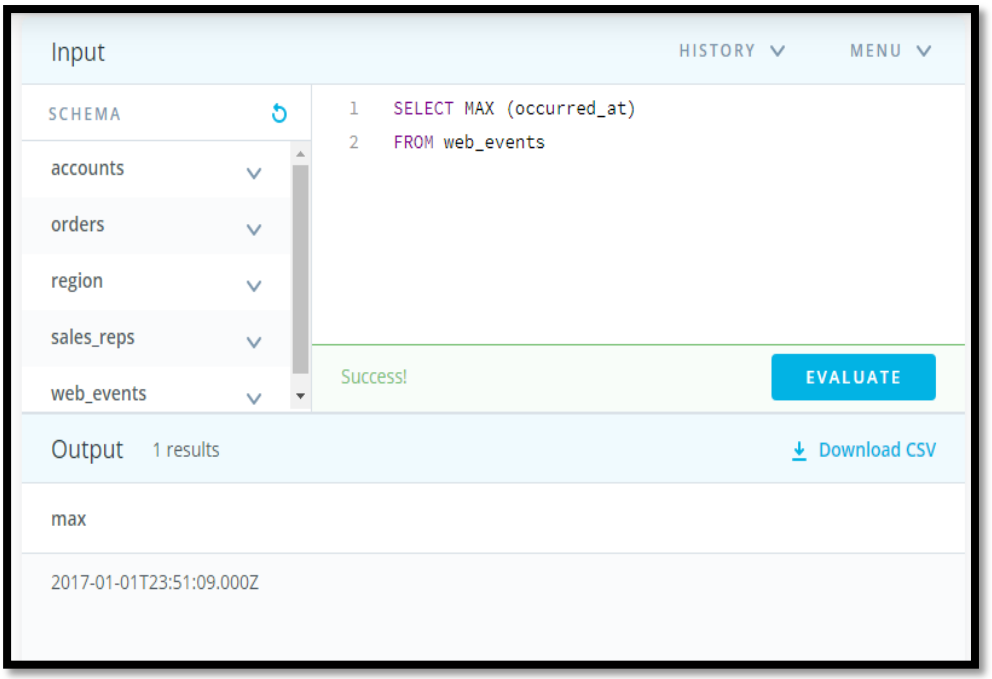
1. When was the earliest order ever placed? You only need to return the date.



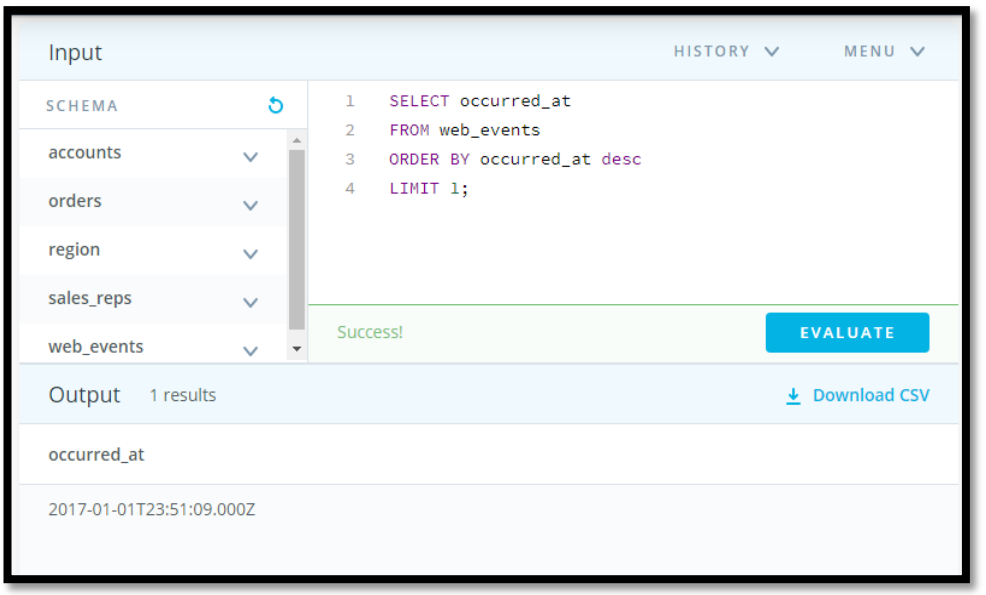
2. Try performing the same query as in question 1 without using an aggregation function.



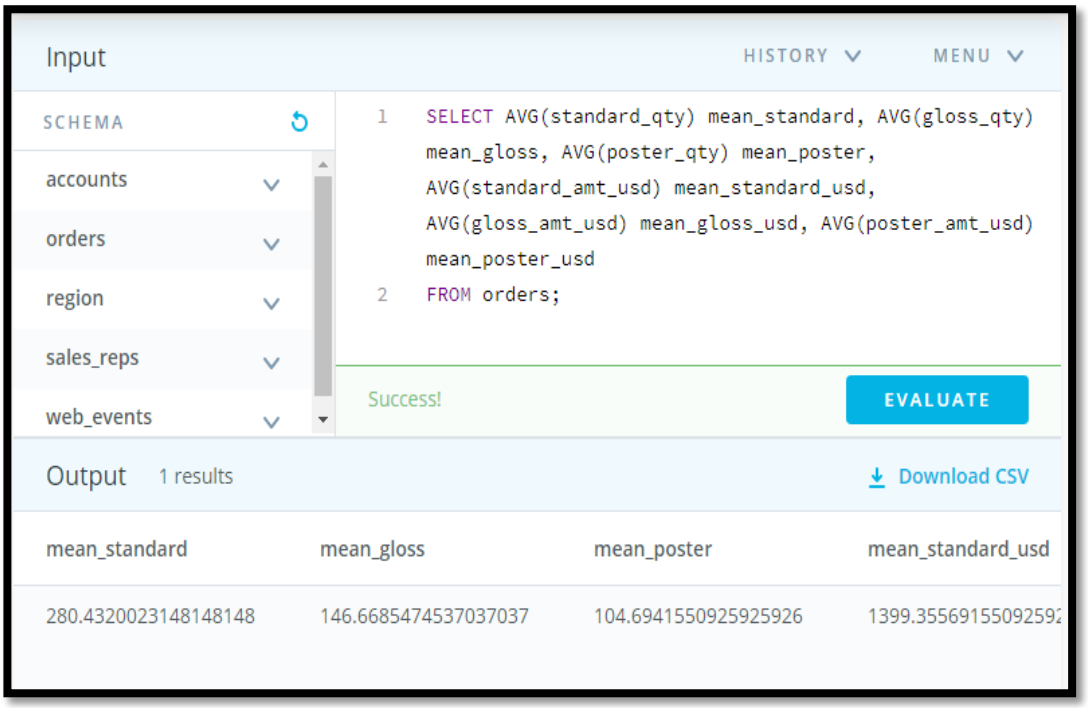
3. When did the most recent (latest) **web_event** occur?



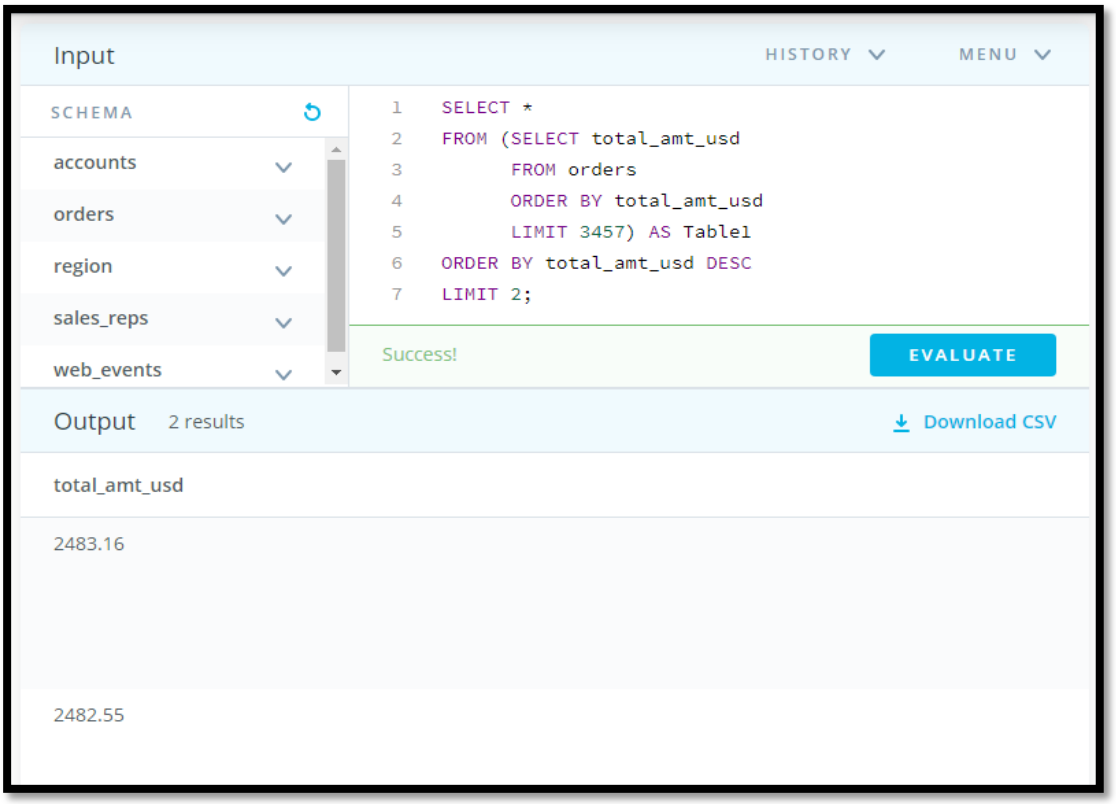
4. Try to perform the result of the previous query without using an aggregation function.



5. Find the mean (**AVERAGE**) amount spent per order on each paper type, as well as the mean amount of each paper type purchased per order. Your final answer should have 6 values - one for each paper type for the average number of sales, as well as the average amount.



6. Via the video, you might be interested in how to calculate the MEDIAN. Though this is more advanced than what we have covered so far try finding - what is the MEDIAN **total_usd** spent on all **orders**?



GROUP BY

The key takeaways here:

- **GROUP BY** can be used to aggregate data within subsets of the data. For example, grouping for different accounts, different regions, or different sales representatives.
- Any column in the **SELECT** statement that is not within an aggregator must be in the **GROUP BY** clause.
- The **GROUP BY** always goes between **WHERE** and **ORDER BY**.
- **ORDER BY** works like **SORT** in spreadsheet software.

GROUP BY - Expert Tip

Before we dive deeper into aggregations using **GROUP BY** statements, it is worth noting that SQL evaluates the aggregations before the **LIMIT** clause. If you don't group by any columns, you'll get a 1-row result—no problem there. If you group by a column with enough unique values that it exceeds the **LIMIT** number, the aggregates will be calculated, and then some rows will simply be omitted from the results.

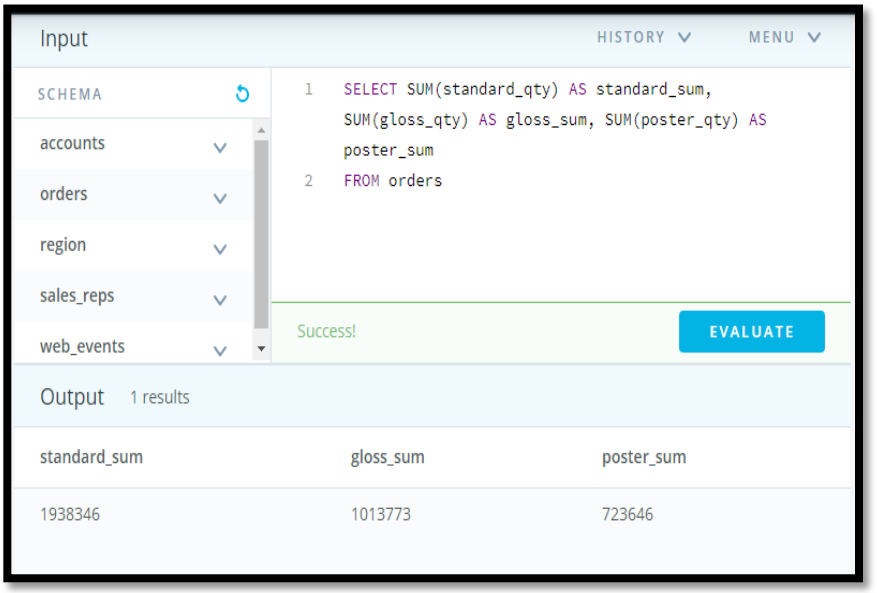
This is actually a nice way to do things because you know you're going to get the correct aggregates. If SQL cuts the table down to 100 rows, then performed the aggregations, your results would be substantially different. The above query's results exceed 100 rows, so it's a perfect example. In the next concept, use the SQL environment to try removing the **LIMIT** and running it again to see what changes.

So far, we've applied these aggregation functions a cross an entire table. You now know how to count all the records in a given table. But you might want to count records across multiple subset of data.

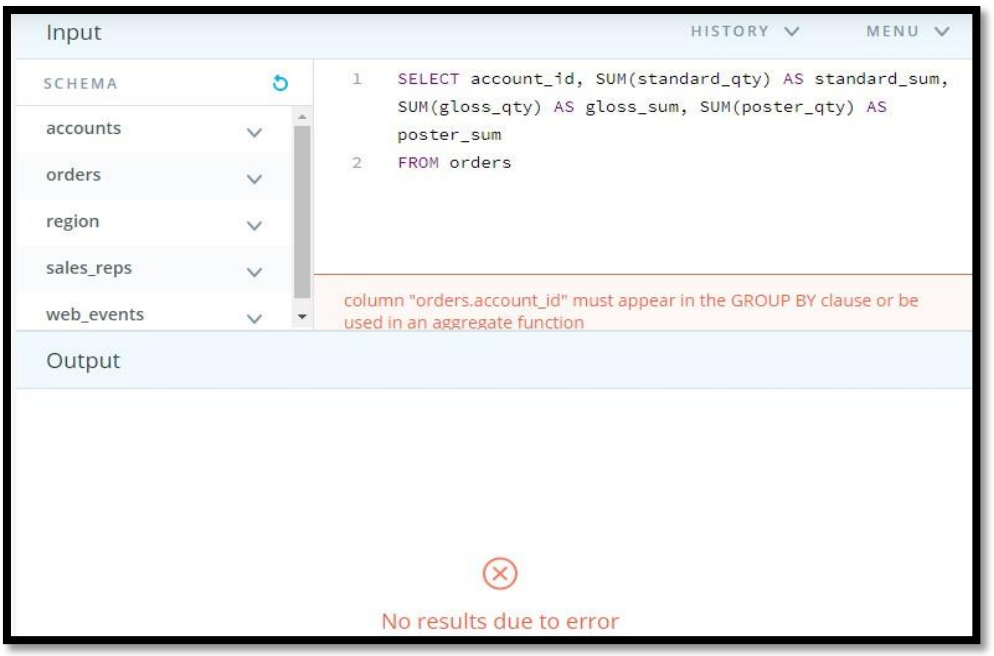
For example, as a sales manager, you might want to sum all of the sales of each paper type for each account.

GROUP BY allows you create segments that will be aggregated independently of one another. In another words, **GROUP BY** allows you take the sum of data limited to each account rather than across the entire dataset.

Let's work our way toward this answer by starting with the sums of the quantities of each type of paper.



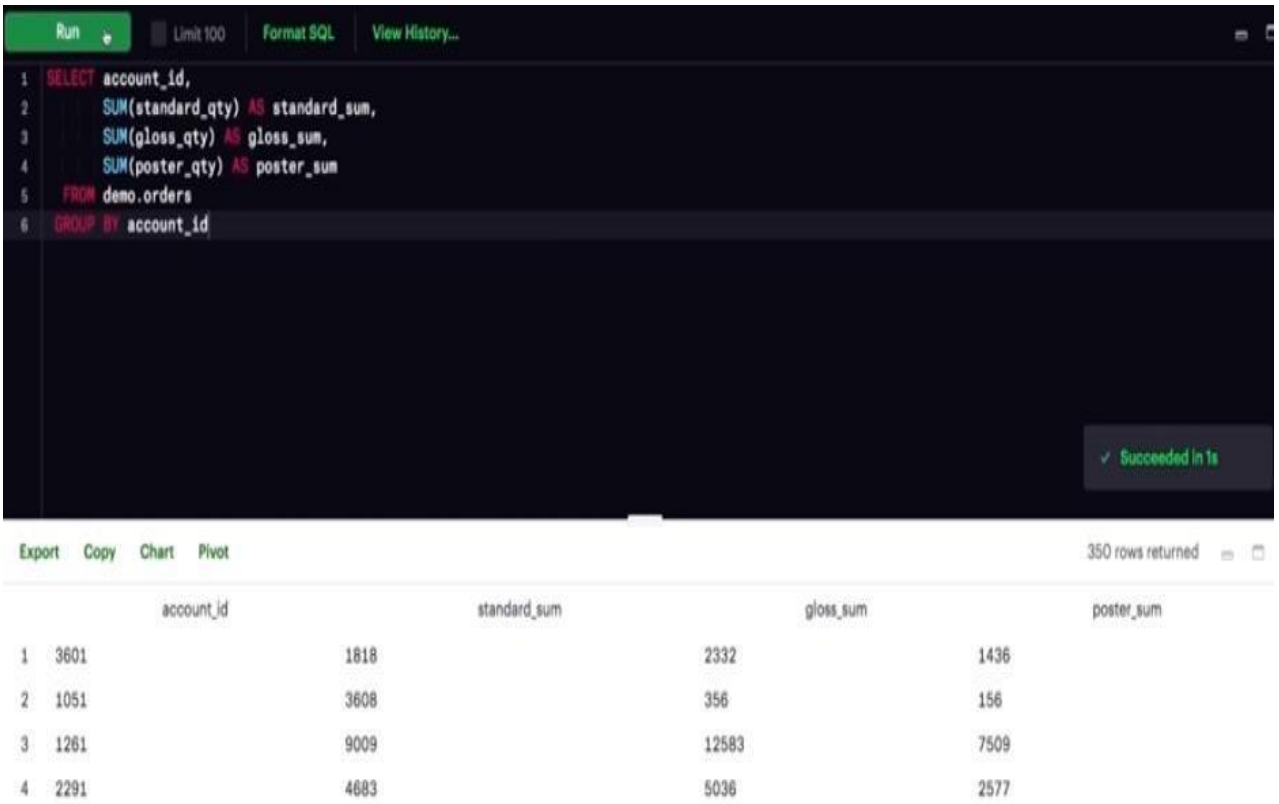
Now, we want to create a separate set of sums of each account_id. So, let's add the account_id column into our SELECT statement.



As you can see, this returns an error. We're effectively collapsing the number of rows returned.

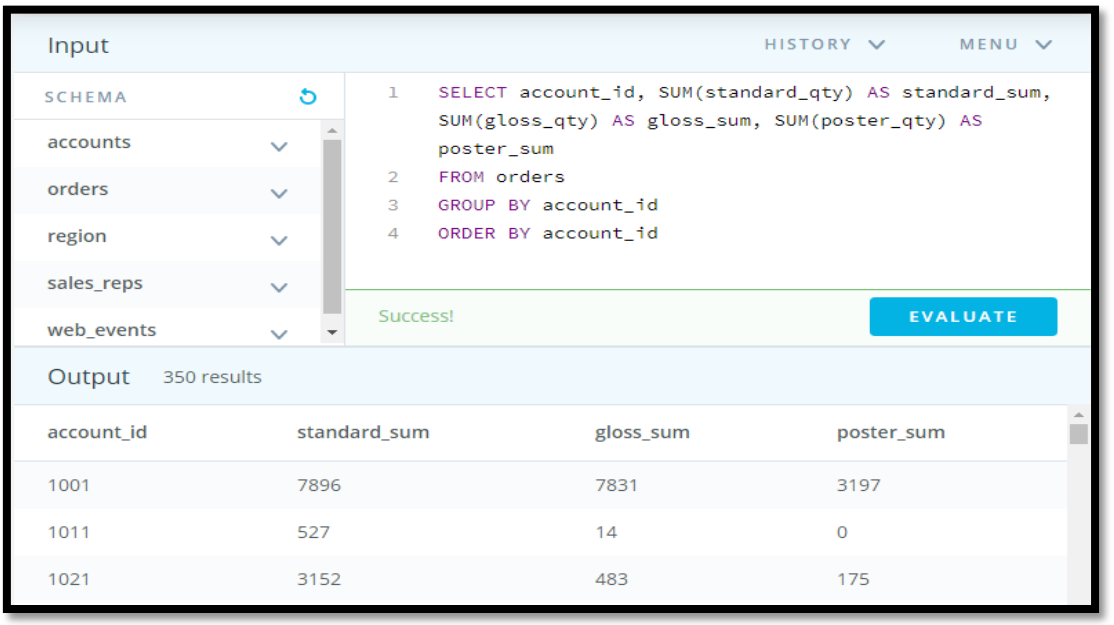
In our previous query, we collapsed all the way down to a single row. The reason we're getting an error here, is the we've included the account_id column but this column isn't being collapsed like the columns that are being aggregated. The query isn't sure whether to sum the account_id as well or whether to make it into a grouping. We have to be explicit about this.

We want to tell the query to aggregate into segments where each segments is one of the values in the account id column. We'll do this using the **GROUP BY** clause.



	account_id	standard_sum	gloss_sum	poster_sum
1	3601	1818	2332	1436
2	1051	3608	356	156
3	1261	9009	12583	7509
4	2291	4683	5036	2577

You can see that this isn't ordered intuitively, so let's fix that. The **GROUP BY** clause allows goes between the **WHERE** clause if there is one and the **ORDER BY** clause.



Whenever there's a filed in the **SELECT** statement that's not being aggregated, the query expects it to be in the **GROUP BY** clause. A column that's not aggregated and not in the **GROUP BY** will return the error we saw before. This is pretty important and it's the key to correctly using **GROUP BY** statements. Again, you should always see any columns in the SELECT statement that are not being aggregated on, in the GROUP BY statement.

GROUP BY II

The key takeaways here:

- **GROUP BY** can be used to aggregate data within subsets of the data. For example, grouping for different accounts, different regions, or different sales representatives.
- Any column in the **SELECT** statement that is not within an aggregator must be in the **GROUP BY** clause.
- The **GROUP BY** always goes between **WHERE** and **ORDER BY**.
- **ORDER BY** works like **SORT** in spreadsheet software.

GROUP BY - Expert Tip

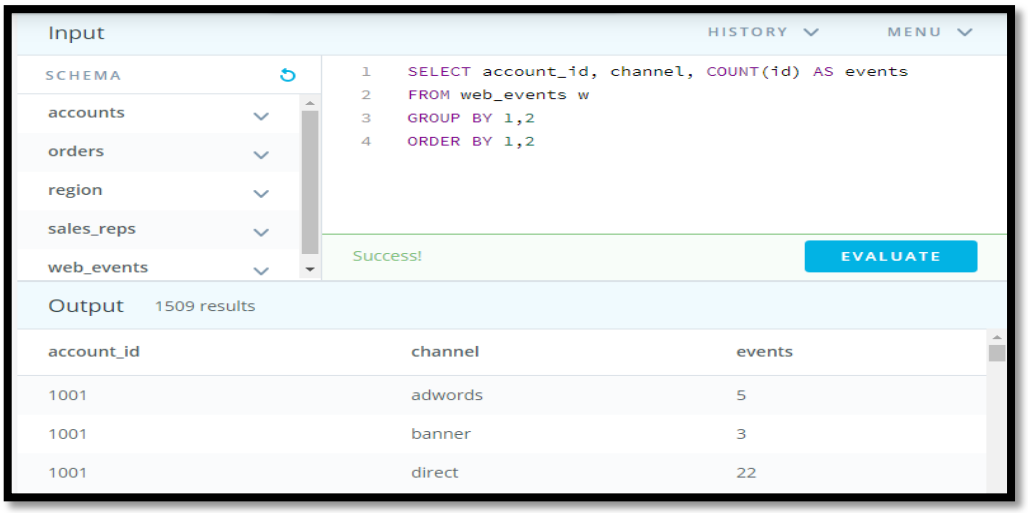
Before we dive deeper into aggregations using **GROUP BY** statements, it is worth noting that SQL evaluates the aggregations before the **LIMIT** clause. If you don't group by any columns, you'll get a 1-row result—no problem there. If you group by a column with enough unique values that it exceeds the **LIMIT** number, the aggregates will be calculated, and then some rows will simply be omitted from the results.

This is actually a nice way to do things because you know you're going to get the correct aggregates. If SQL cuts the table down to 100 rows, then performed the aggregations, your results would be substantially different. The above query's results exceed 100 rows, so it's a perfect example. In the next concept, use the SQL environment to try removing the **LIMIT** and running it again to see what changes.

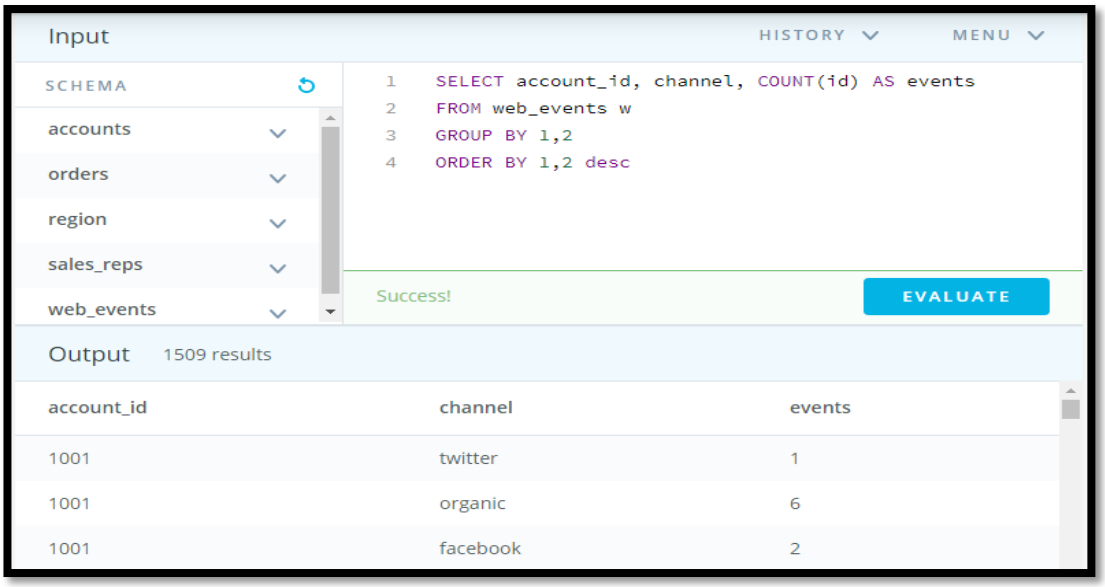
If we want to segment our data into even more granular chunks, we can GROUP BY multiple columns. Imagine yourself as a marketing manager at Parch & Posey, trying to understand how each account interacted with various advertising channels.

Which channels are driving traffic and leading to purchases? Are we investing in channels that aren't worth the cost ? how much traffic are we obtaining from each channel ?

One way we might begin to look into these questions, is to count up all of the events for each channel, for each account id.



It looks like the really useful information here is in the events column. Let's reorder this to highlight the highest volume channels for each account.



You'll notice that account id comes before events in the order by clause. Here in the results, you can see that these are ordered first by account ID, then by events within the account ID.

Quiz: GROUP BY

GROUP BY Note

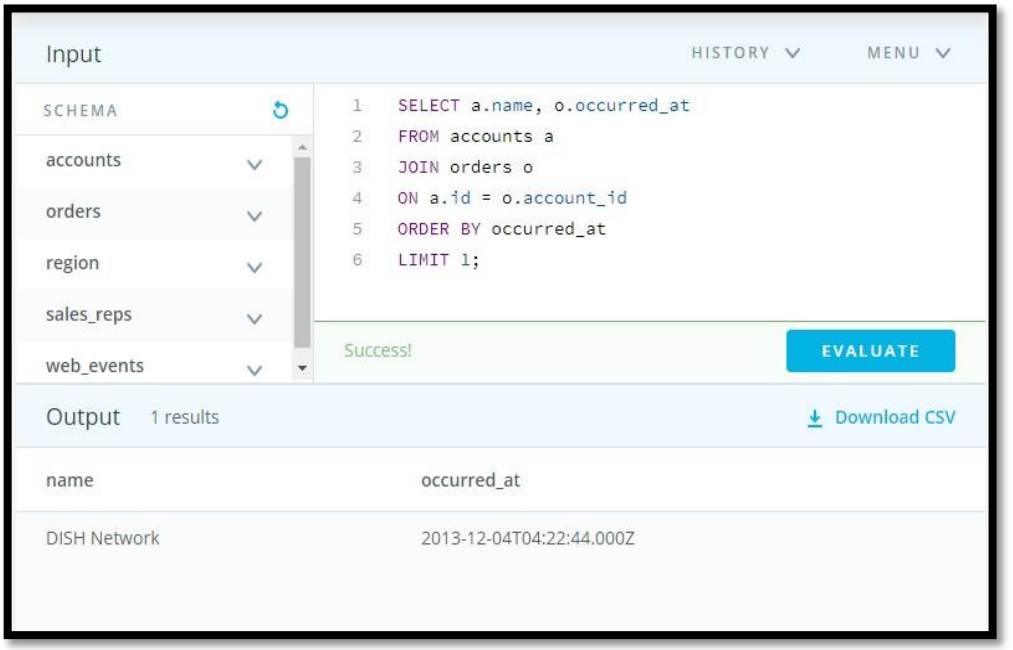
Now that you have been introduced to **JOIN**s, **GROUP BY**, and aggregate functions, the real power of **SQL** starts to come to life. Try some of the below to put your skills to the test!

Questions: GROUP BY

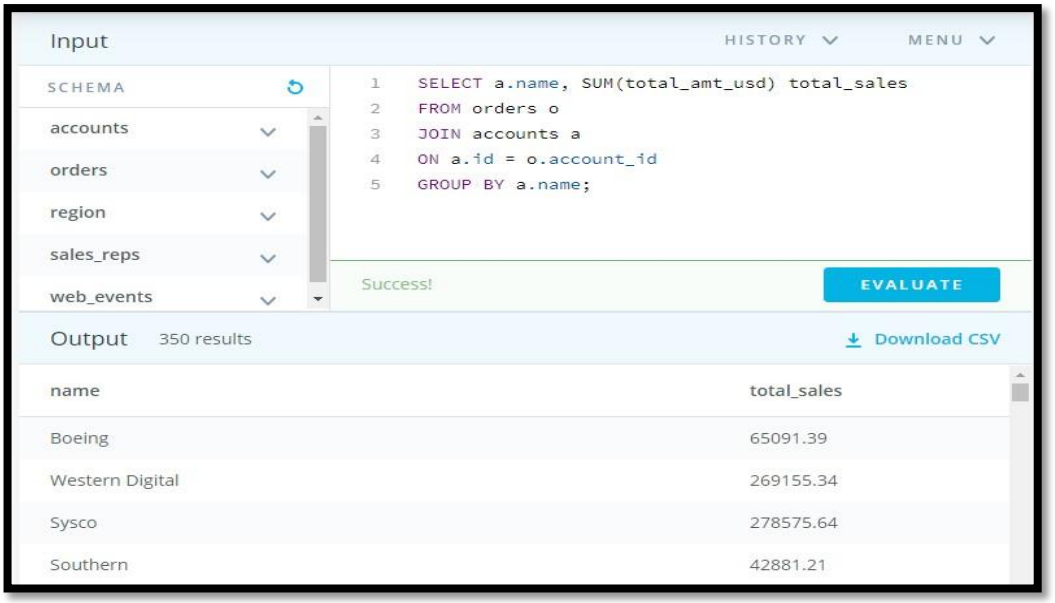
Use the **SQL** environment below to assist with answering the following questions. Whether you get stuck or you just want to double check your solutions, my answers can be found at the top of the next concept.

One part that can be difficult to recognize is when it might be easiest to use an aggregate or one of the other SQL functionalities. Try some of the below to see if you can differentiate to find the easiest solution.

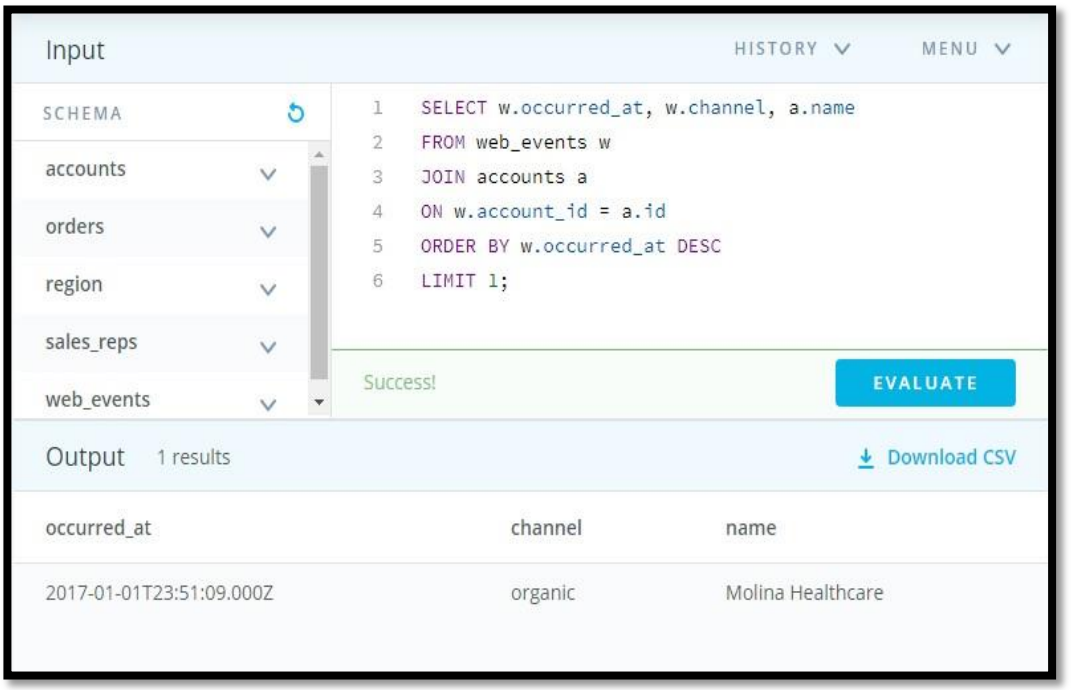
1. Which **account** (by name) placed the earliest order? Your solution should have the **account name** and the **date** of the order.



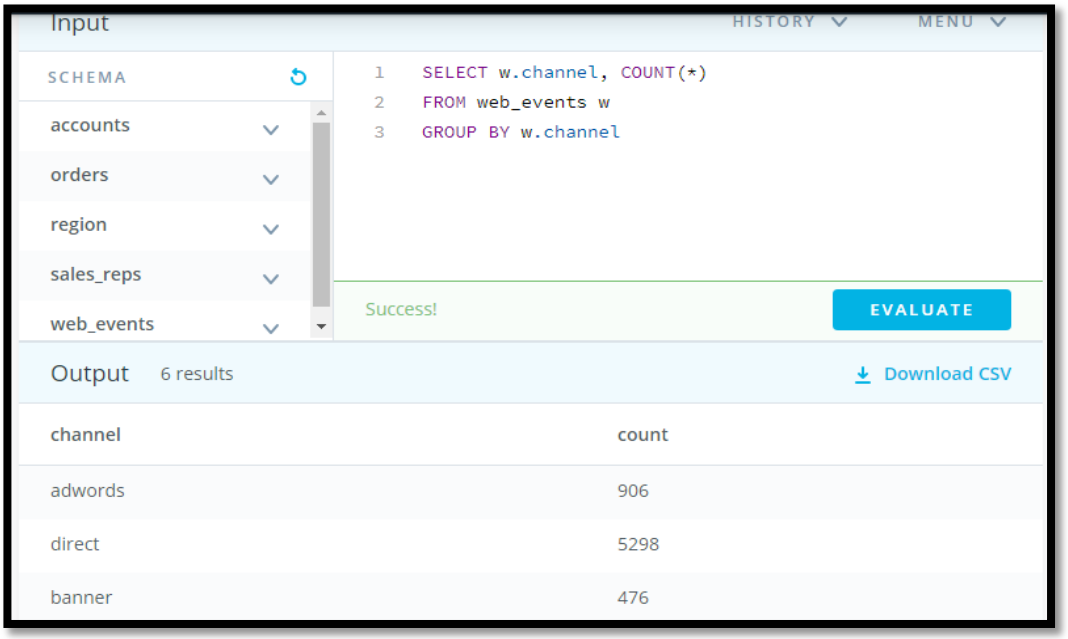
2. Find the total sales in **usd** for each account. You should include two columns - the total sales for each company's orders in **usd** and the company **name**.



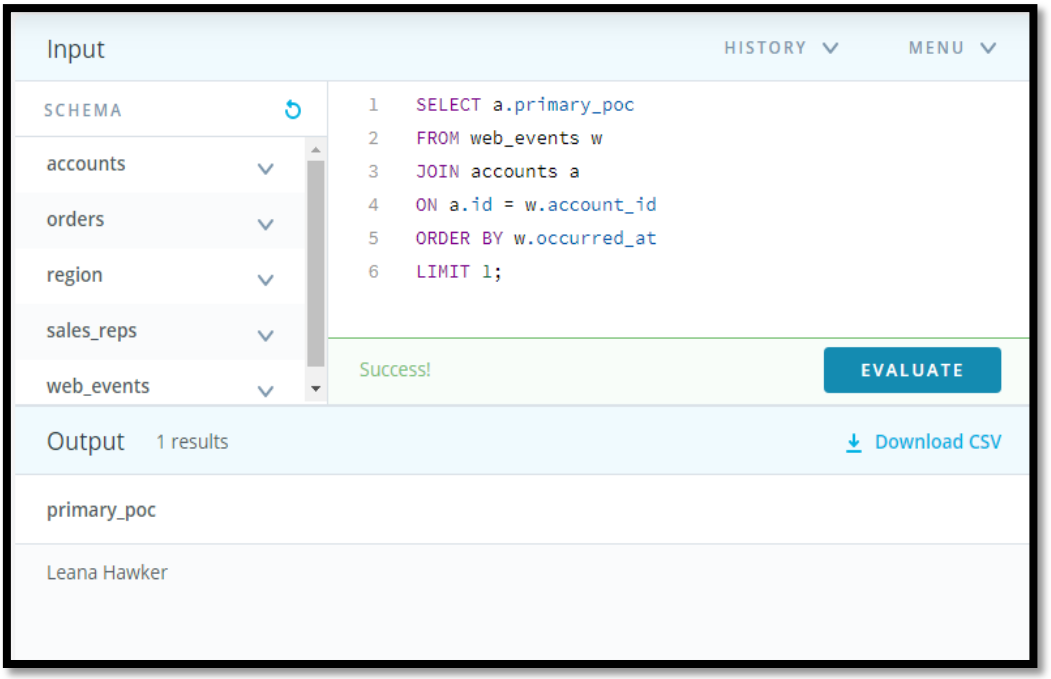
3. Via what **channel** did the most recent (latest) **web_event** occur, which **account** was associated with this **web_event**? Your query should return only three values - the **date**, **channel**, and **account name**.



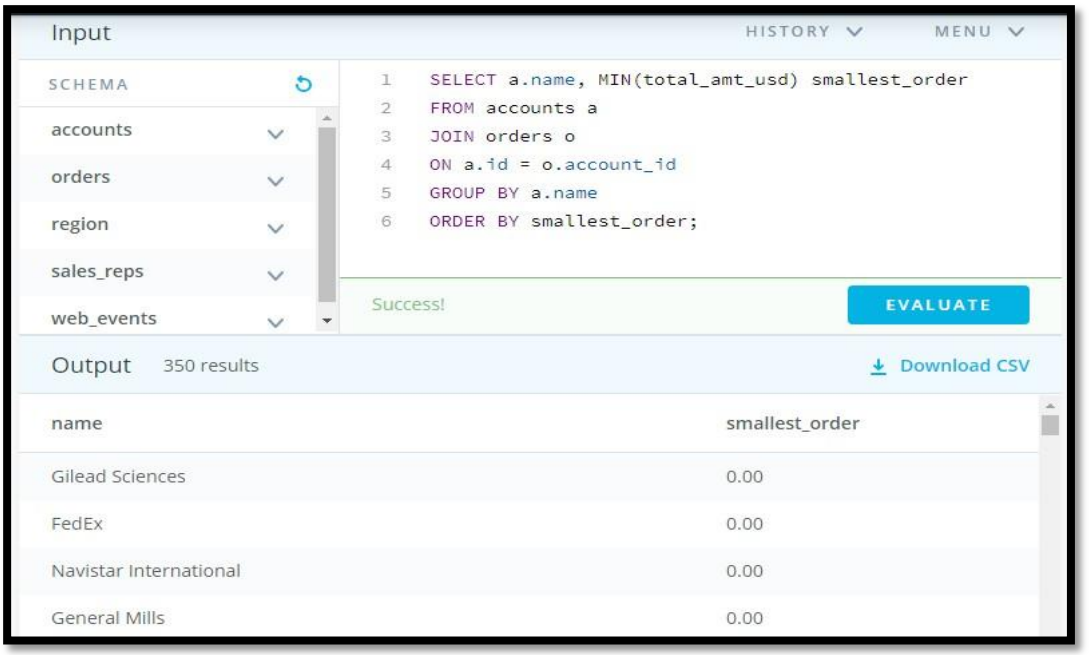
4. Find the total number of times each type of **channel** from the **web_events** was used. Your final table should have two columns - the **channel** and the number of times the channel was used.



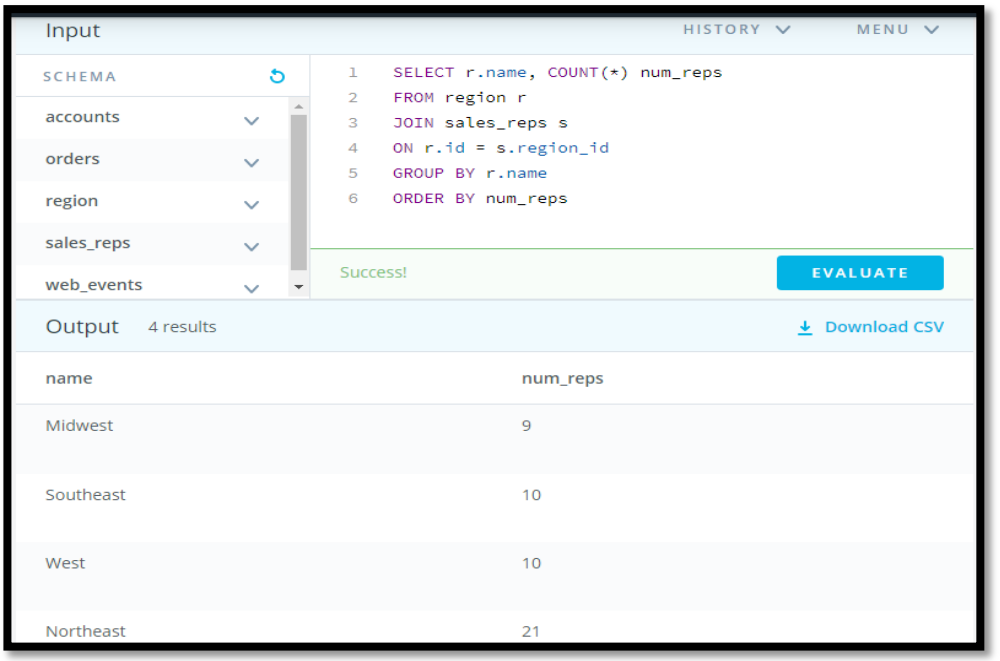
5. Who was the **primary contact** associated with the earliest **web_event**?



6. What was the smallest order placed by each **account** in terms of **total usd**. Provide only two columns - the account **name** and the **total usd**. Order from smallest dollar amounts to largest.



7. Find the number of **sales reps** in each region. Your final table should have two columns - the **region** and the number of **sales_reps**. Order from fewest reps to most reps.

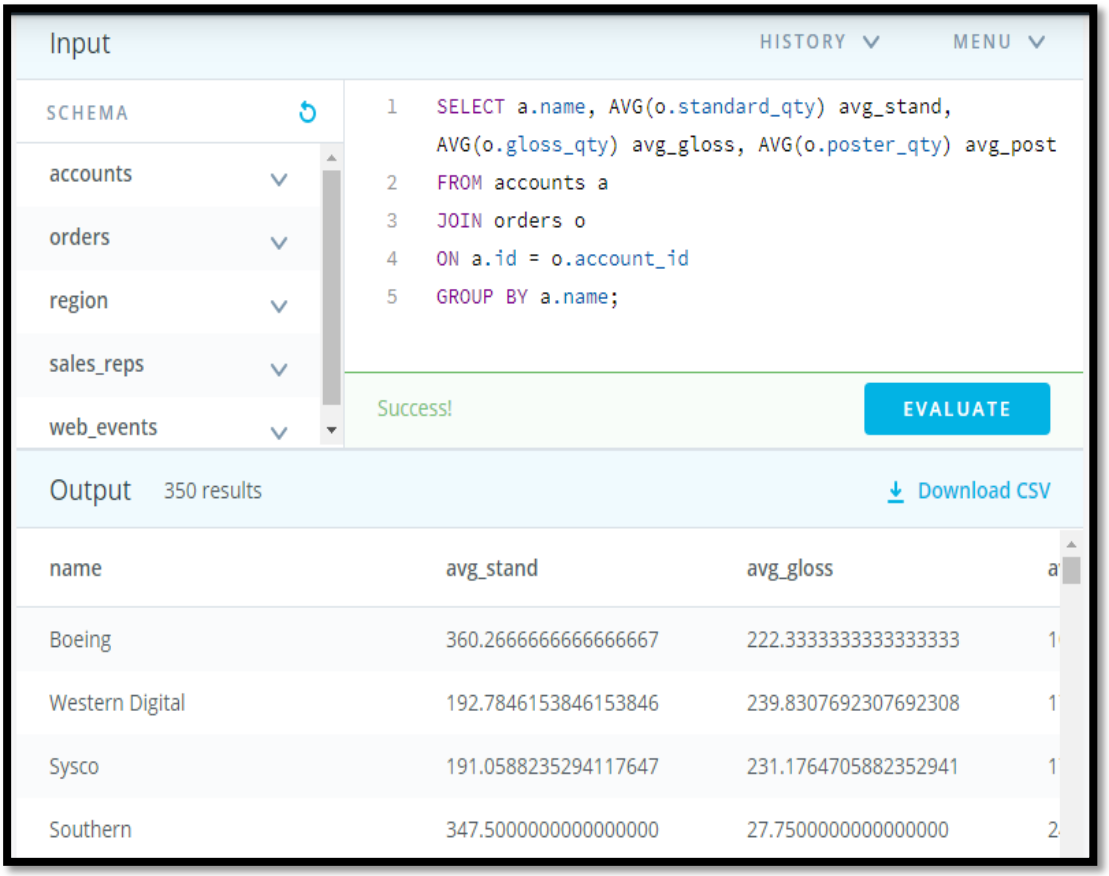


Quiz: GROUP BY Part II

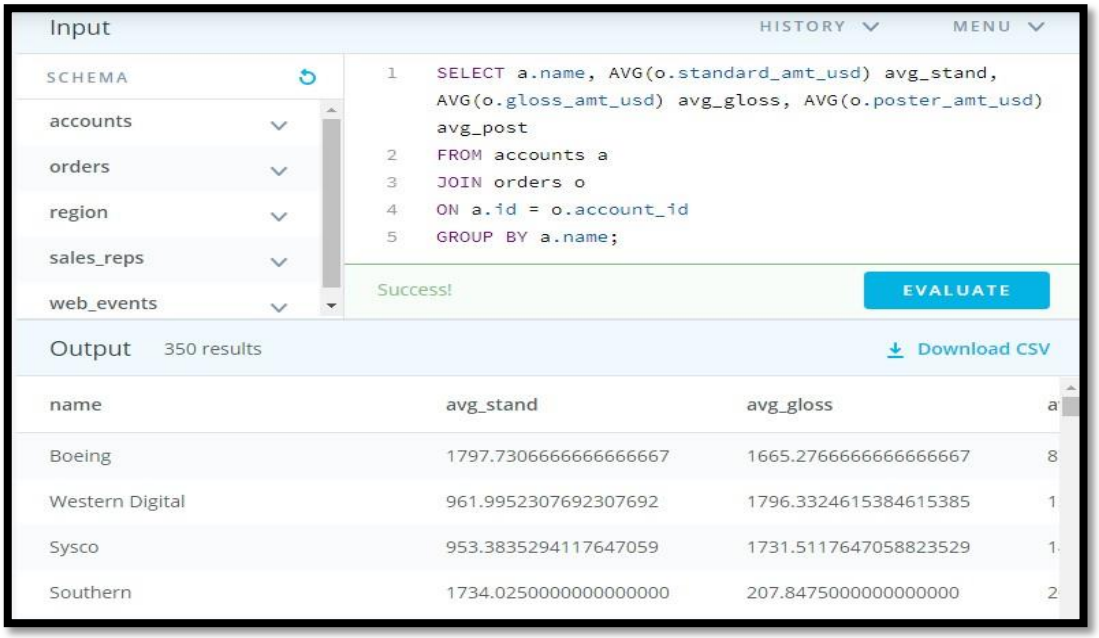
Questions: GROUP BY Part II

Use the **SQL** environment below to assist with answering the following questions. Whether you get stuck or you just want to double check your solutions, my answers can be found at the top of the next concept.

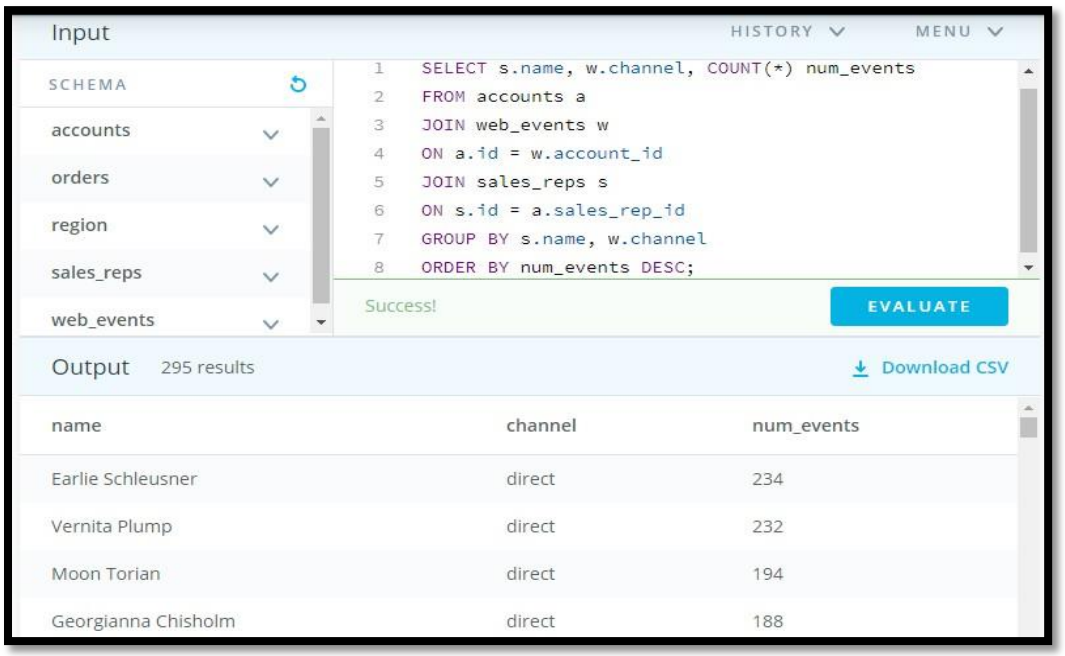
1. For each account, determine the average amount of each type of paper they purchased across their orders. Your result should have four columns - one for the account **name** and one for the average quantity purchased for each of the paper types for each account.



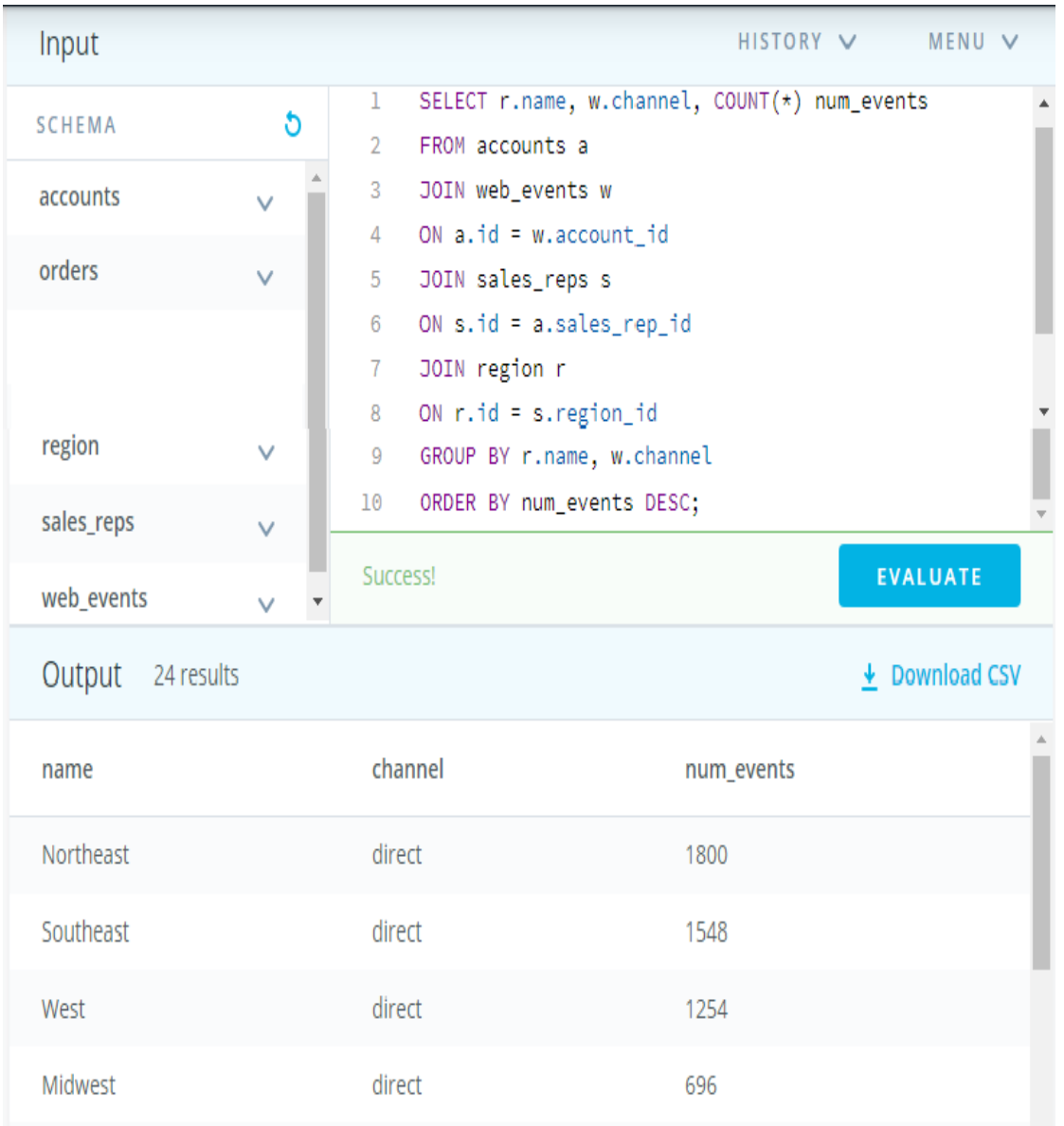
2. For each account, determine the average amount spent per order on each paper type. Your result should have four columns - one for the account **name** and one for the average amount spent on each paper type.



3. Determine the number of times a particular **channel** was used in the **web_events** table for each **sales rep**. Your final table should have three columns - the **name of the sales rep**, the **channel**, and the number of occurrences. Order your table with the highest number of occurrences first.



4. Determine the number of times a particular **channel** was used in the **web_events** table for each **region**. Your final table should have three columns - the **region name**, the **channel**, and the number of occurrences. Order your table with the highest number of occurrences first.



DISTINCT

DISTINCT is always used in **SELECT** statements, and it provides the unique rows for all columns written in the **SELECT** statement. Therefore, you only use **DISTINCT** once in any particular **SELECT** statement.
You could write:

```
SELECT DISTINCT column1, column2, column3
FROM table1;
```

which would return the unique (or **DISTINCT**) rows across all three columns.
You would **not** write:

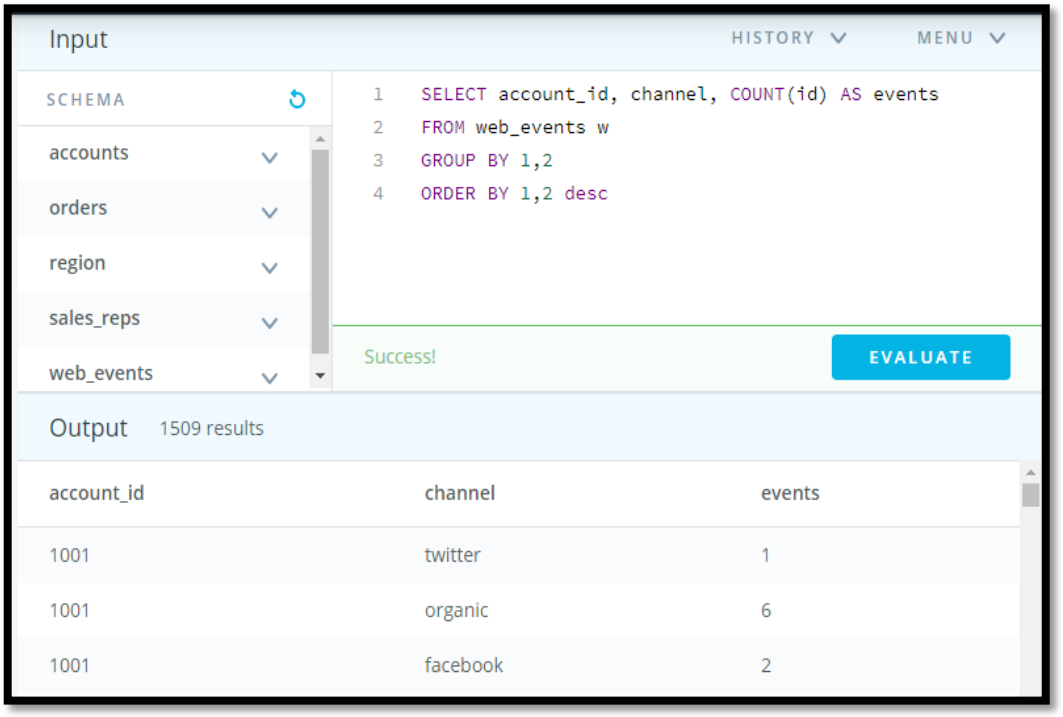
```
SELECT DISTINCT column1, DISTINCT column2, DISTINCT column3
FROM table1;
```

You can think of **DISTINCT** the same way you might think of the statement "unique".

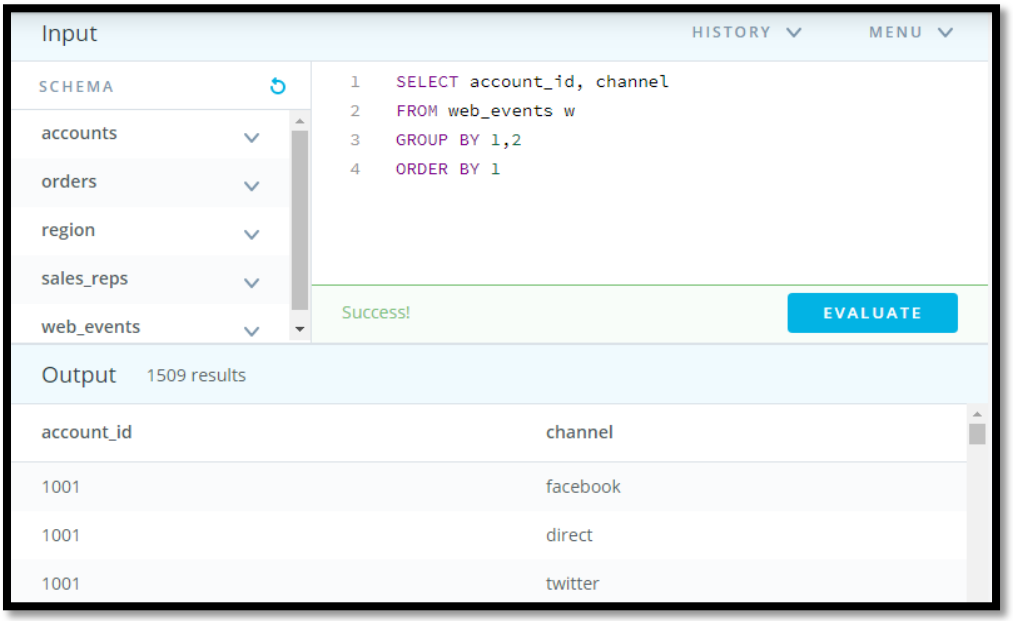
DISTINCT - Expert Tip

It's worth noting that using **DISTINCT**, particularly in aggregations, can slow your queries down quite a bit.

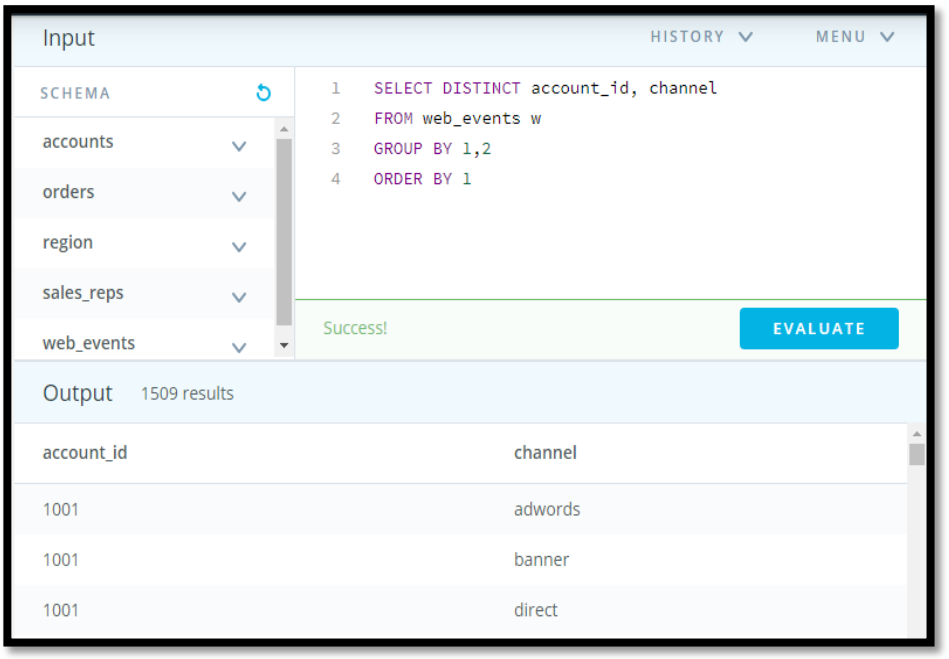
You can think of **DISTINICT** this way, if you want to GROUP BY some columns but you don't necessary want to include any aggregations, you can used **DISTINICT** instead.
As an example, let's revisit the count of events by channel by account we looked at in the GROUP BY segment.



Notice the row count, 1509. If we get rid of the events column, you can see that the resulting query returns basically the same results with the same row count.



And if we run that query again with **DISTANICT**, you can see that the results are the same.



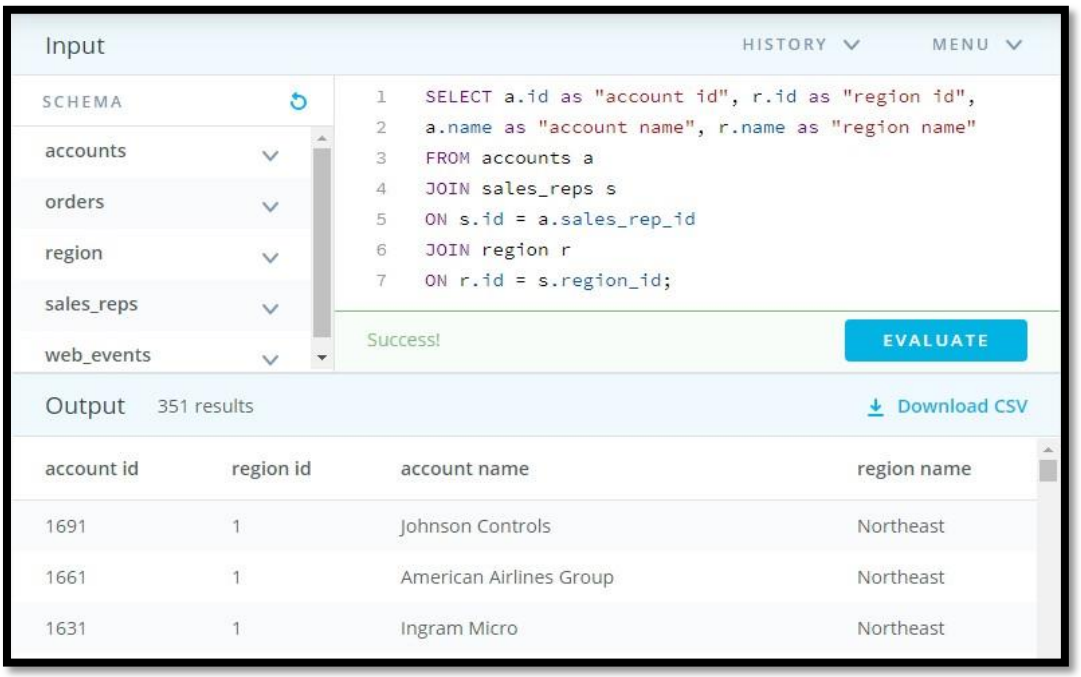
Of course, they don't include the COUNT anymore because we've removed it from our select statement. Since it's no longer in our select statement, and since we're no longer doing an aggregation, we don't need to GROUP BY clause either.

Quiz: DISTINCT

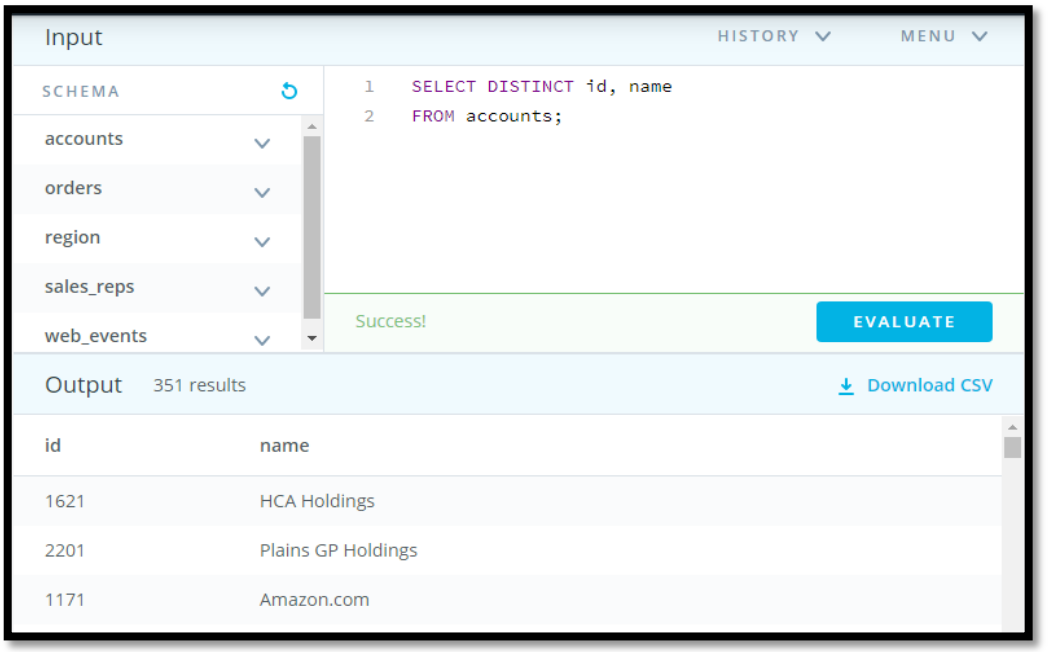
Questions: DISTINCT

Use the **SQL** environment below to assist with answering the following questions. Whether you get stuck or you just want to double check your solutions, my answers can be found at the top of the next concept.

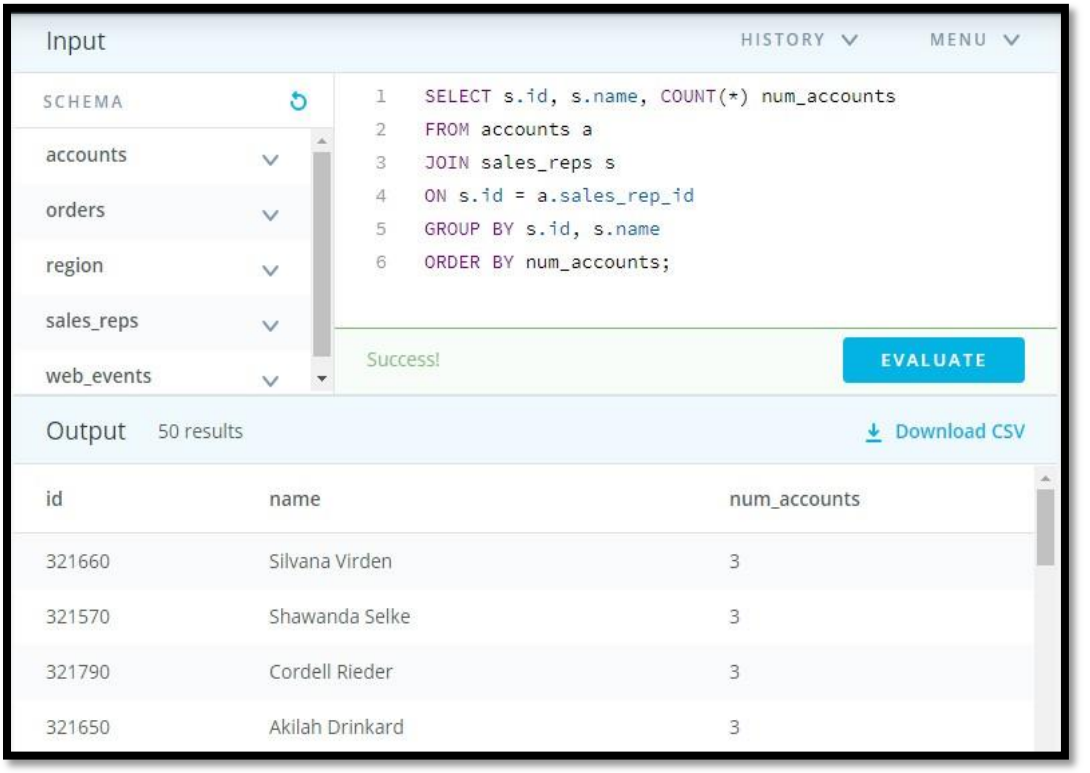
1. Use **DISTINCT** to test if there are any accounts associated with more than one region.



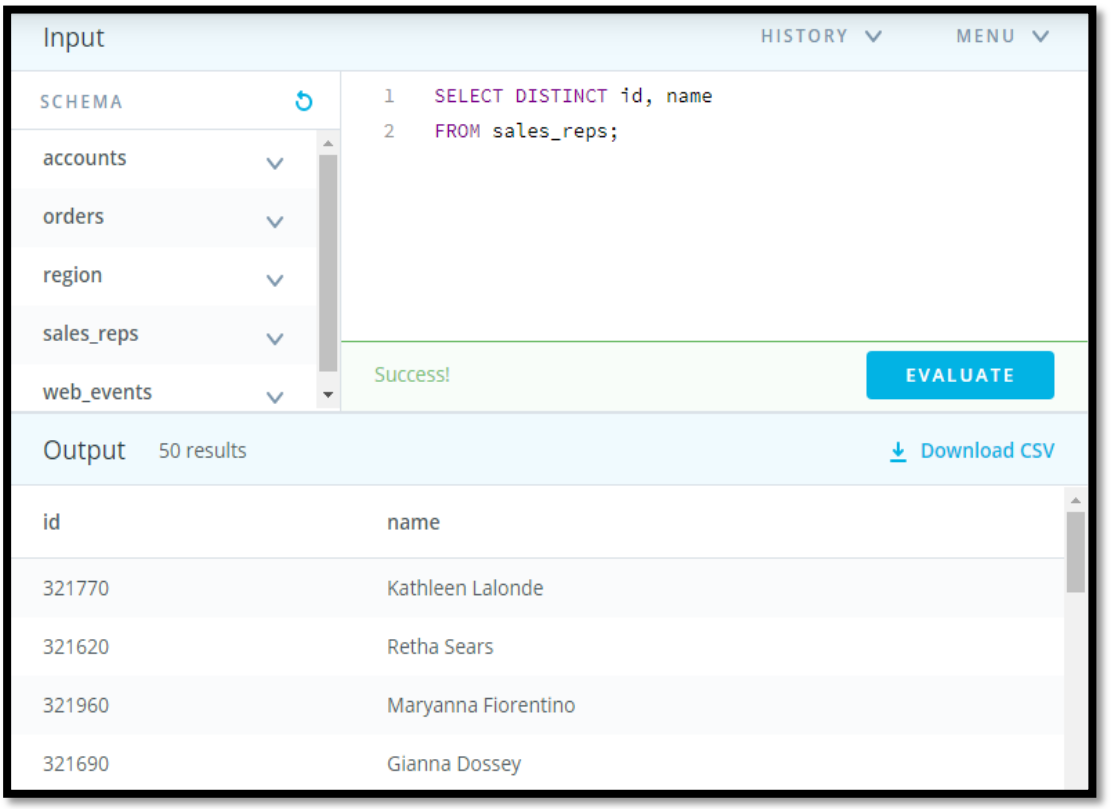
In another way



2. Have any **sales reps** worked on more than one account?



In another way



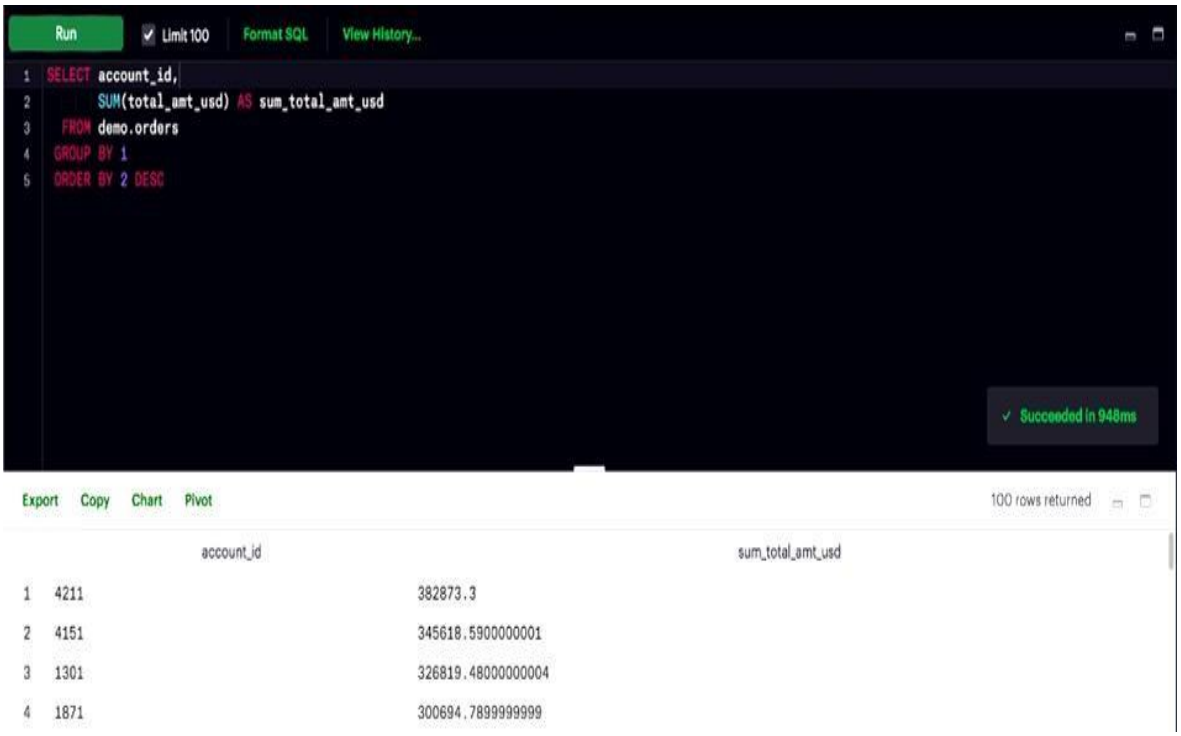
HAVING

HAVING - Expert Tip

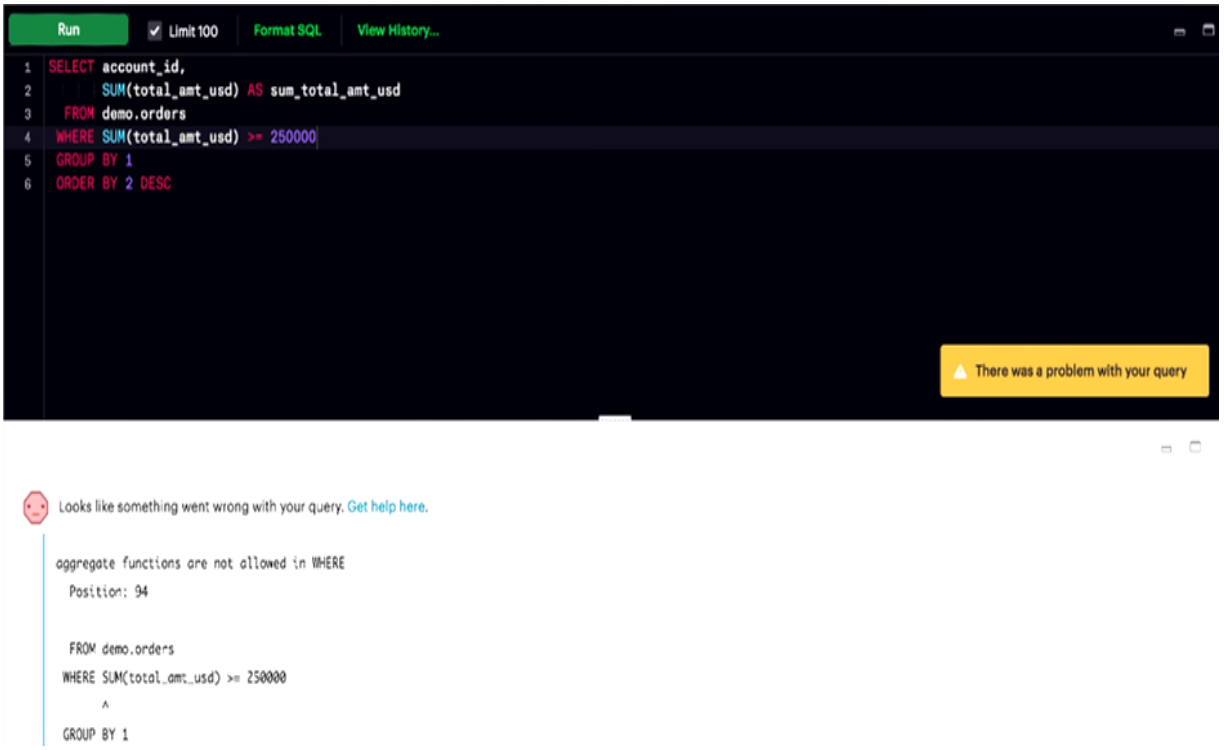
HAVING is the –clean‖ way to filter a query that has been aggregated, but this is also commonly done using a **subquery**. Essentially, any time you want to perform a **WHERE** on an element of your query that was created by an aggregate, you need to use **HAVING** instead.

Example :

imagine yourself as an account manager at Parch &Posey, working with the company's largest accounts. you might want to identify the total sales in US dollars for account with over $250,000 in sales, to better understand the profession of revenue that comes from these large accounts. To get this list, let's first get the sum sales for each account. We'll order them in descending order so that we can see all the highest value accounts at the top of the result set.

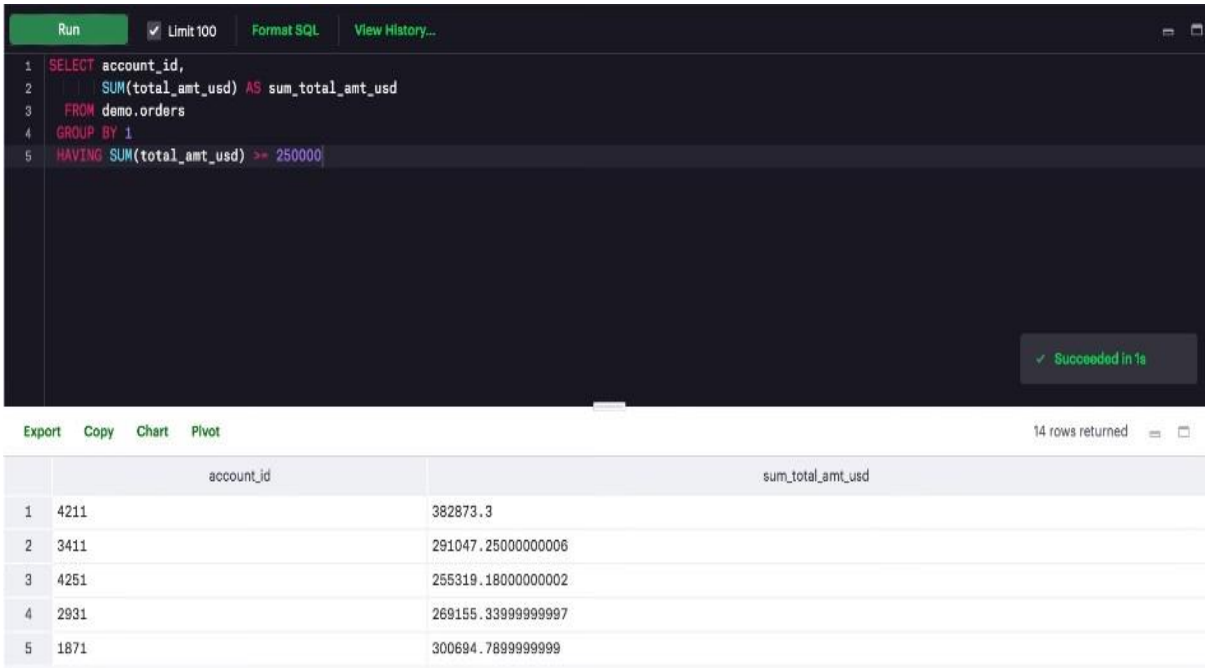


Now let's filter down to just the accounts with more than $250,000 in sales.



You'll notice that the WHER clause isn't exactly built for this. The WHERE clause won't work for this because it doesn't allow you to filter on aggregate columns. that's where the having clause comes in.

This filter query down to just the account IDs with more than $250,000 in total sales.



	account_id	sum_total_amt_usd
1	4211	382873.3
2	3411	291047.25000000006
3	4251	255319.18000000002
4	2931	269155.33999999997
5	1871	300694.7899999999

One thing to keep in mind is that this is really only useful when grouping by one or more columns. if you're aggregating across the entire dataset, the output is only one line anyway, so there's no need to filter beyond that.

Quiz : HAVING

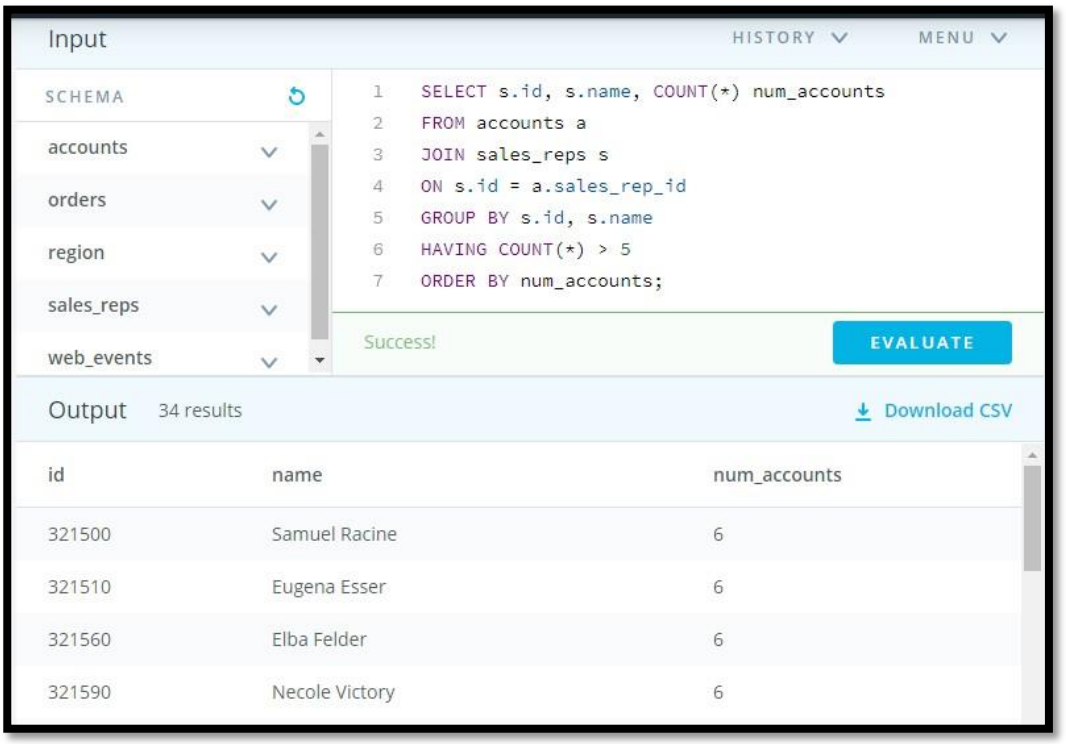
Often there is confusion about the difference between **WHERE** and **HAVING**. Select all the statements that are true regarding **HAVING** and **WHERE** statements.

- [x] **WHERE** subsets the returned data based on a logical condition.
- [x] **WHERE** appears after the **FROM**, **JOIN**, and **ON** clauses, but before **GROUP BY**.
- [x] **HAVING** appears after the **GROUP BY** clause, but before the **ORDER BY** clause.
- [x] **HAVING** is like **WHERE**, but it works on logical statements involving aggregations.

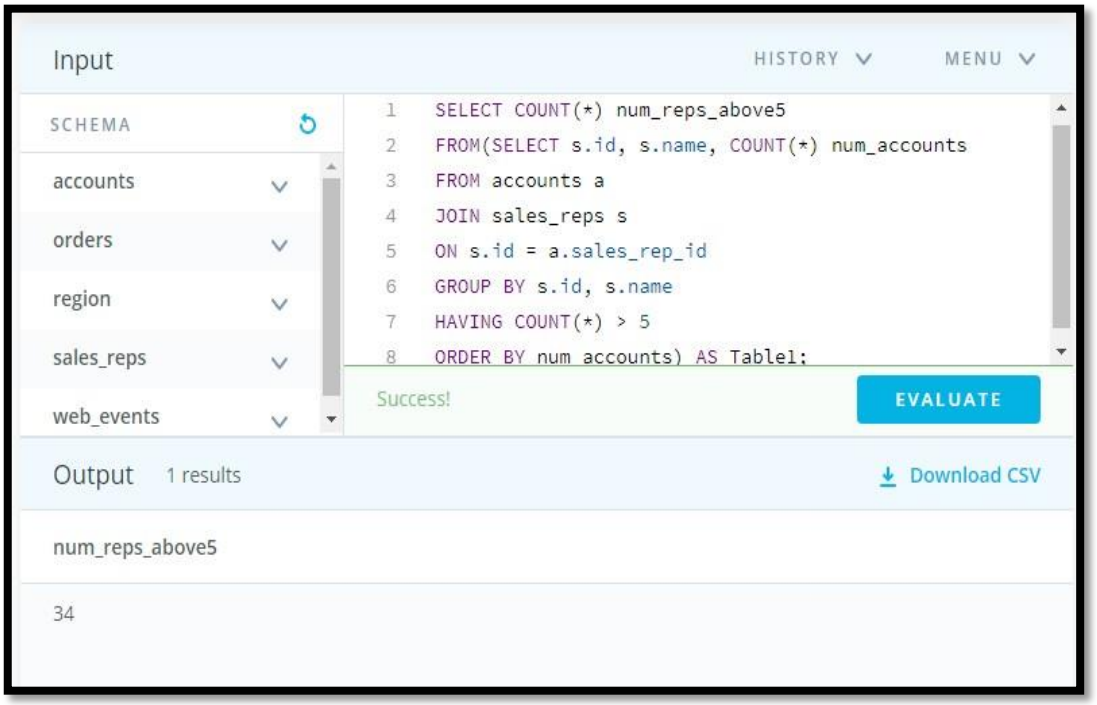
Questions: HAVING

Use the **SQL** environment below to assist with answering the following questions. Whether you get stuck or you just want to double check your solutions, my answers can be found at the top of the next concept.

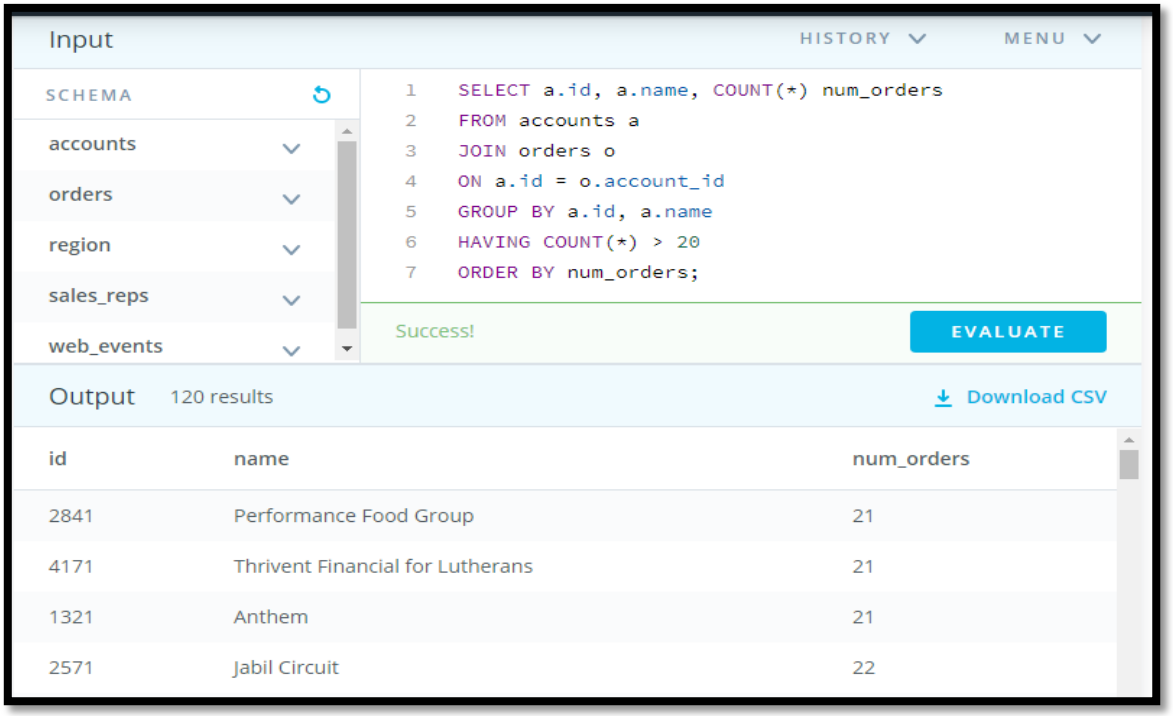
1. How many of the **sales reps** have more than 5 accounts that they manage?



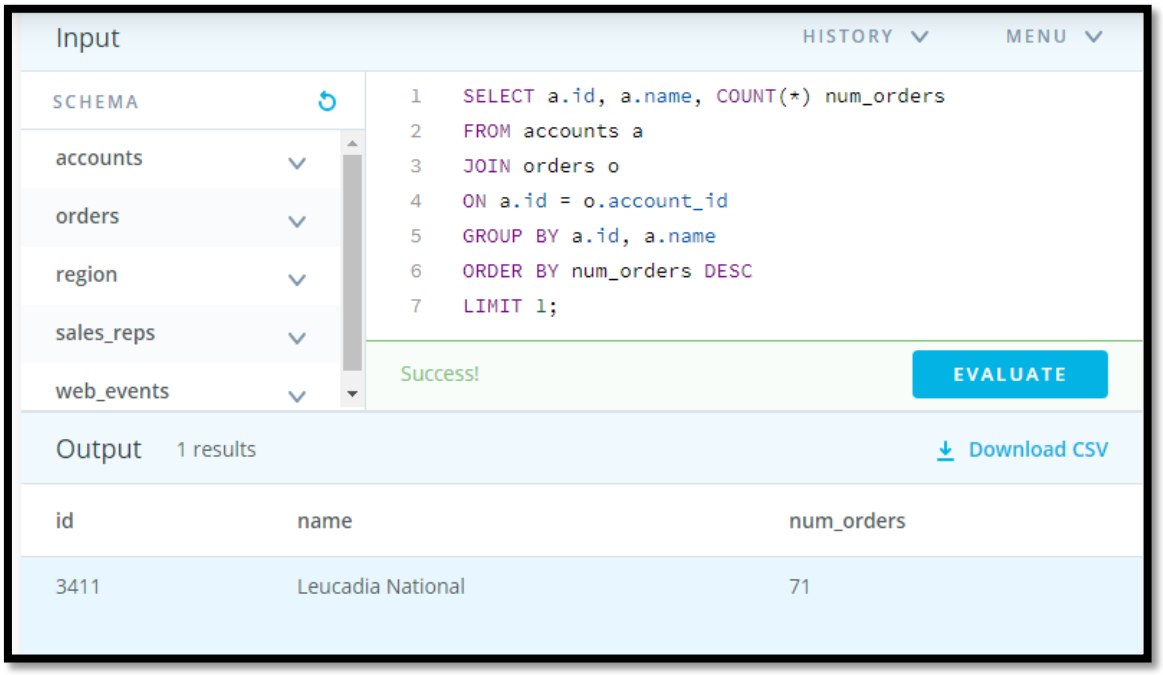
and technically, we can get this using a **SUBQUERY** as shown below. This same logic can be used for the other queries, but this will not be shown.



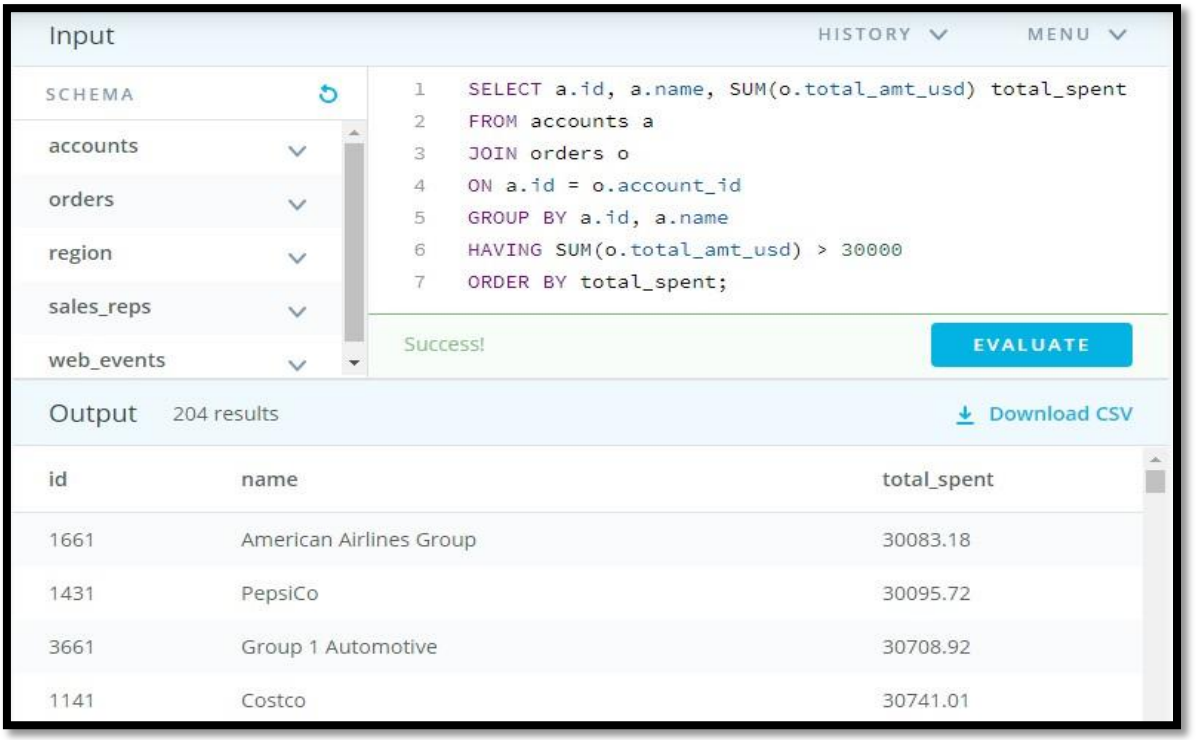
2. How many **accounts** have more than 20 orders?



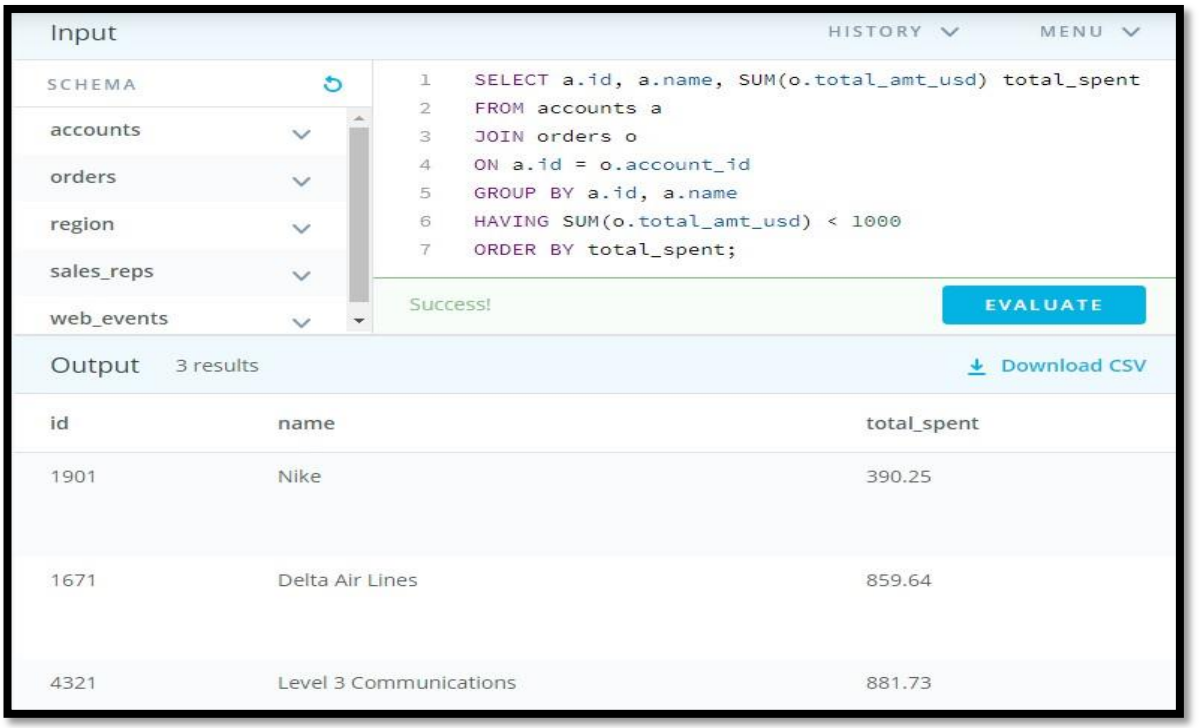
3. Which account has the most orders?



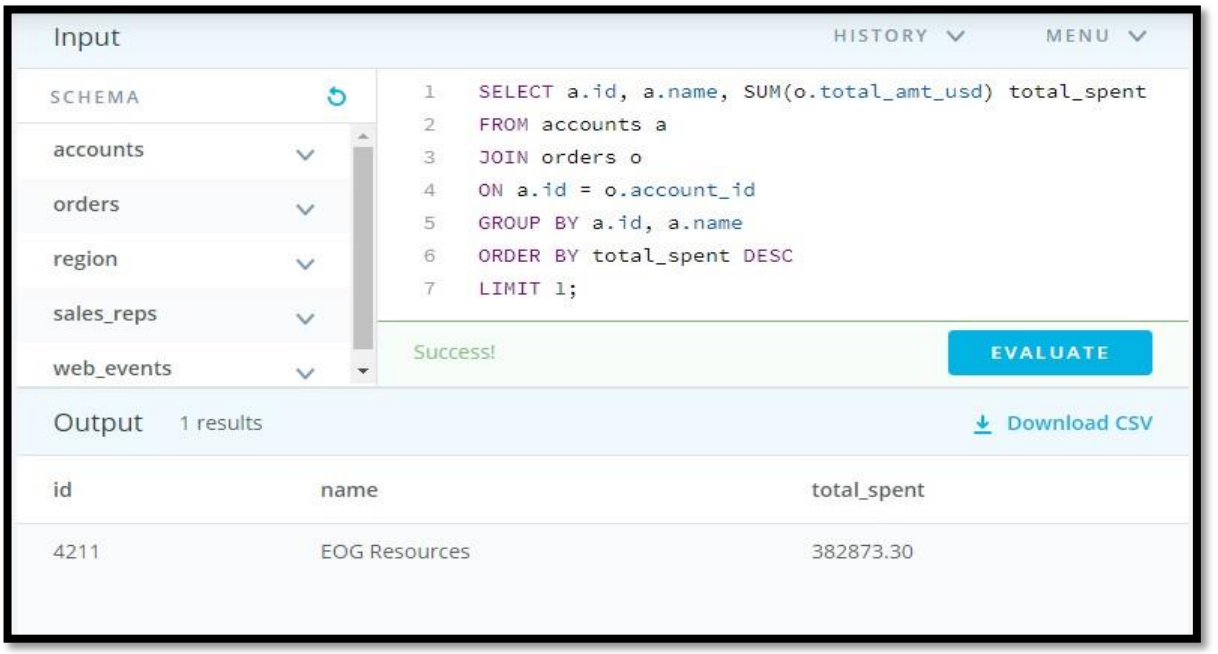
4. How many accounts spent more than 30,000 usd total across all orders?



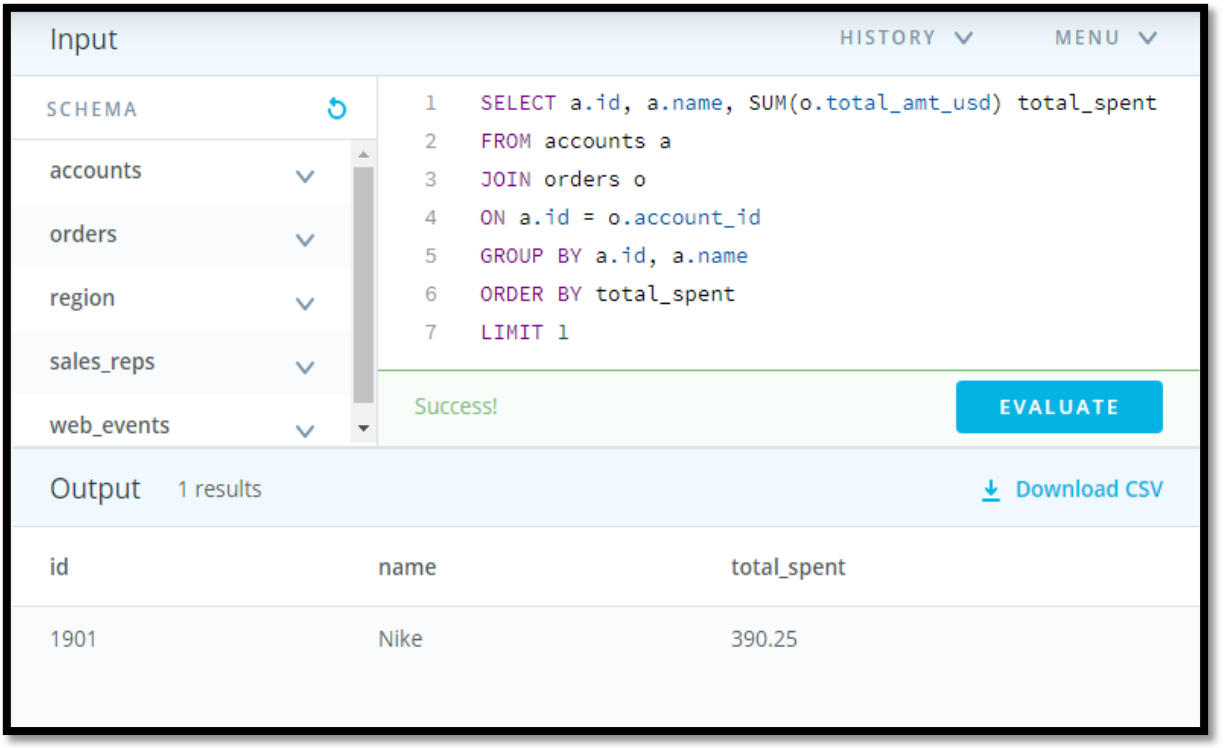
5. How many accounts spent less than 1,000 usd total across all orders?



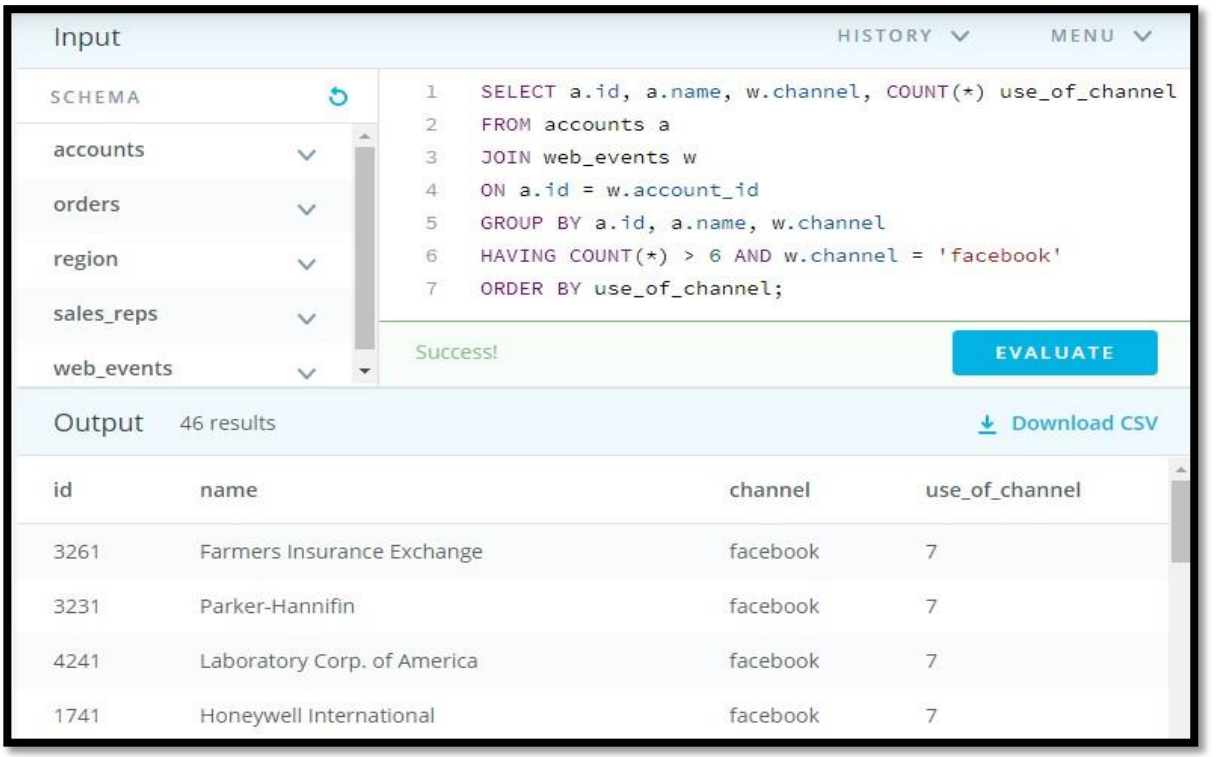
6. Which account has spent the most with us?



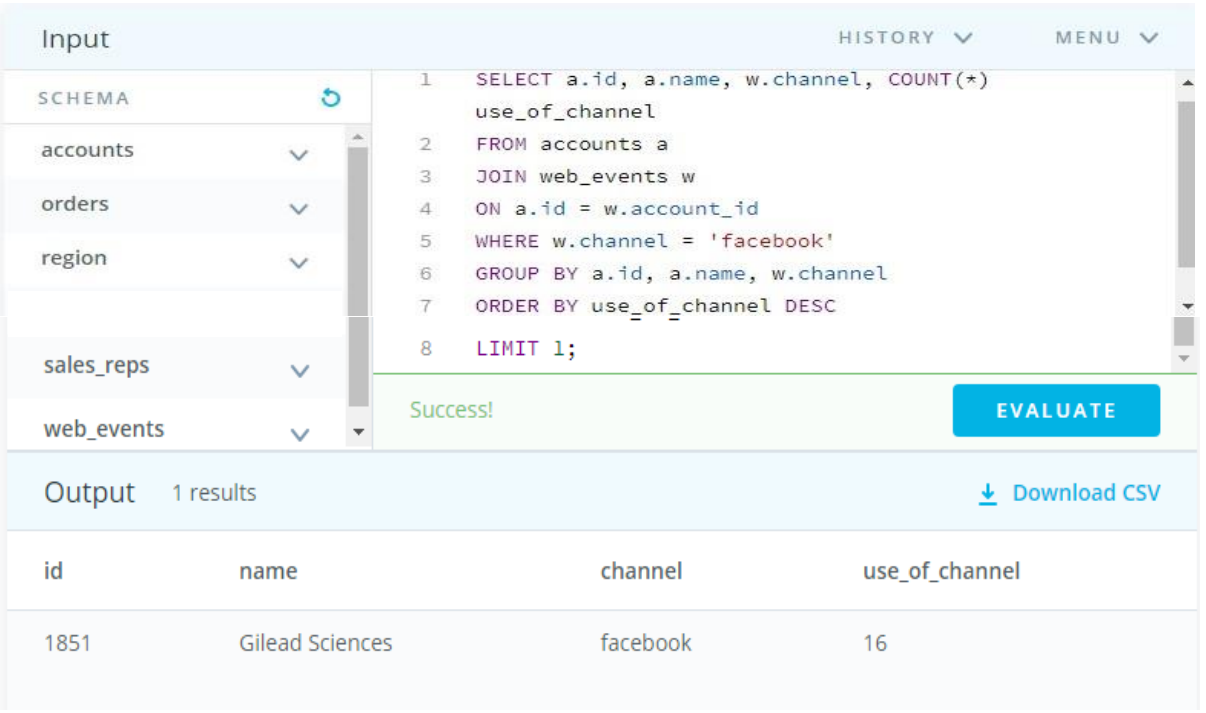
7. Which account has spent the least with us?



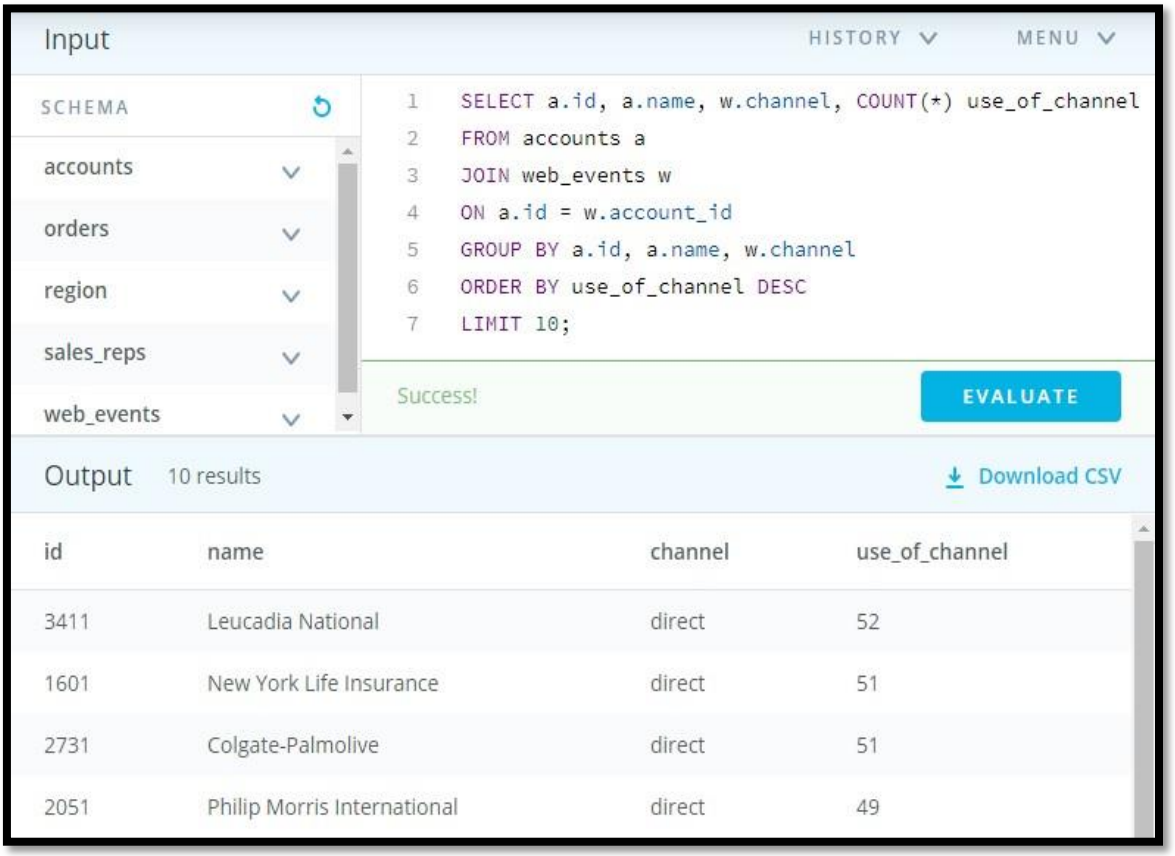
8. Which accounts used Facebook as a **channel** to contact customers more than 6 times?



9. Which account used facebook most as a **channel**?



10. Which channel was most frequently used by most accounts?



DATE Functions

GROUPing **BY** a date column is not usually very useful in SQL, as these columns tend to have transaction data down to a second. Keeping date information at such a granular data is both a blessing and a curse, as it gives really precise information (a blessing), but it makes grouping information together directly difficult (a curse).

Lucky for us, there are a number of built in SQL functions that are aimed at helping us improve our experience in working with dates.

Here we saw that dates are stored in year, month, day, hour, minute, second, which helps us in truncating. In the next concept, you will see a number of functions we can use in SQL to take advantage of this functionality.

By now, you might have noticed that dates are a bit hard to work with. Aggregating by data fields, in particular, doesn't work in a practical way. It treats each time stamp is unique. When it would be more practical to round to the nearest day, week or month and aggregate across that period. Take for example, this sum of standard paper quantities by time period.

you can see in the results here that this really isn't any more useful than booking at the row data. This aggregates from six 6,912 rows in the row data down to 6,908. Almost all of the date in this table are unique. good news though, there are plenty of special functions to make dates easier to work with. But before we dig into date and time functions, let's take a look at how data stored.

If you live in the United States, you're probably used to seeing dates formatted as month-month, day-day, year-year-year-year, or similar month first format. This isn't necessarily better or rows, it's just different. Databases do it yet another way. Ordering from least to most granular part of the data, year- year- year, month-month, day-day. This is a very specific utility, and bases of that date stored alphabetically are also in chronological order. In another wards, date ordering is the same whether you think of them as dates or as bits of text.

With this example here, we can see that the year first way the databases store dates, is ideal for sorting the way we'll want to retrieve this information in the future. Whether we want the most recent or oldest information, day first and month first date formats sort of funny ways that don't make a tone of sense. Another benefit is the date can easily be truncated in order to group them for analyses.

Take the date, **2017 - 04 - 01 12:15:01**. If we want to group that with other events that occurred on the same day, we can't do that with the date in it's current format. Grouping now or group by every event that occurred in April 1st, at 12:15 and 1 second. That won't help us very much.

DATE Functions II

The first function you are introduced to in working with dates is **DATE_TRUNC**.

DATE_TRUNC allows you to truncate your date to a particular part of your date-time column. Common trunctions are day, month, and year.

DATE_PART can be useful for pulling a specific portion of a date, but notice pulling month or day of the week (dow) means that you are no longer keeping the years in order. Rather you are grouping for certain components regardless of which year they belonged in.

For additional functions you can use with dates, but the **DATE_TRUNC** and **DATE_PART** functions definitely give you a great start!

You can reference the columns in your select statement in **GROUP BY** and **ORDER BY** clauses with numbers that follow the order they appear in the select statement. For example

```
SELECT standard_qty, COUNT(*)
FROM orders
```

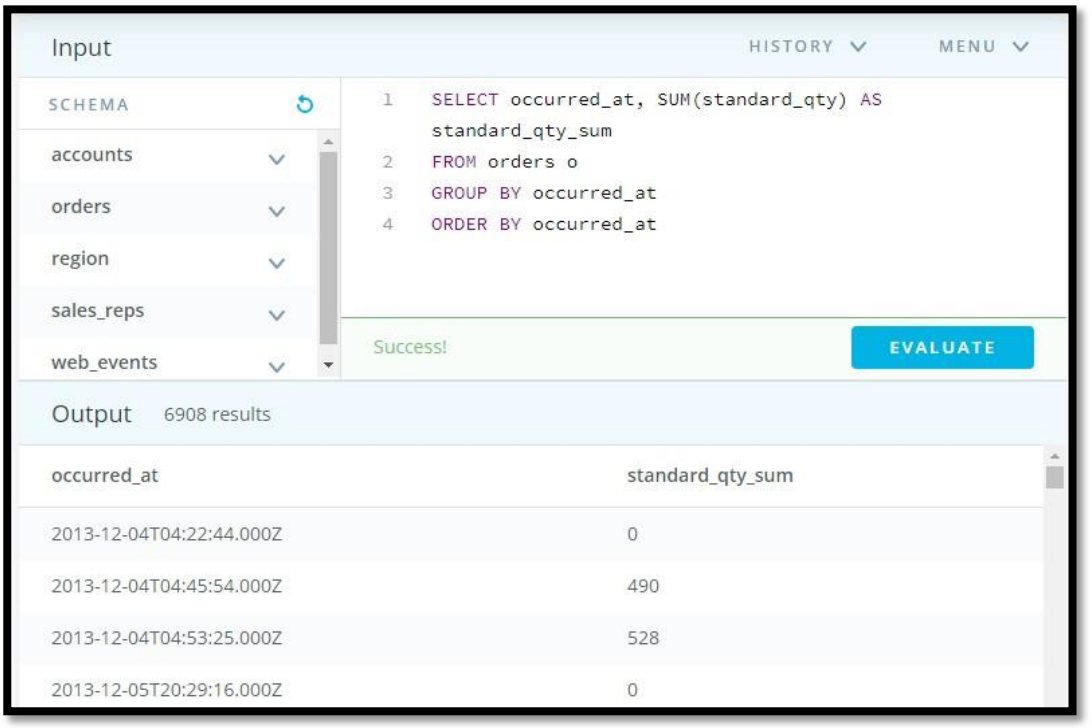
GROUP BY 1 (this 1 refers to standard_qty since it is the first of the columns included in the select statement)

ORDER BY 1 (this 1 refers to standard_qty since it is the first of the columns included in the select statement)

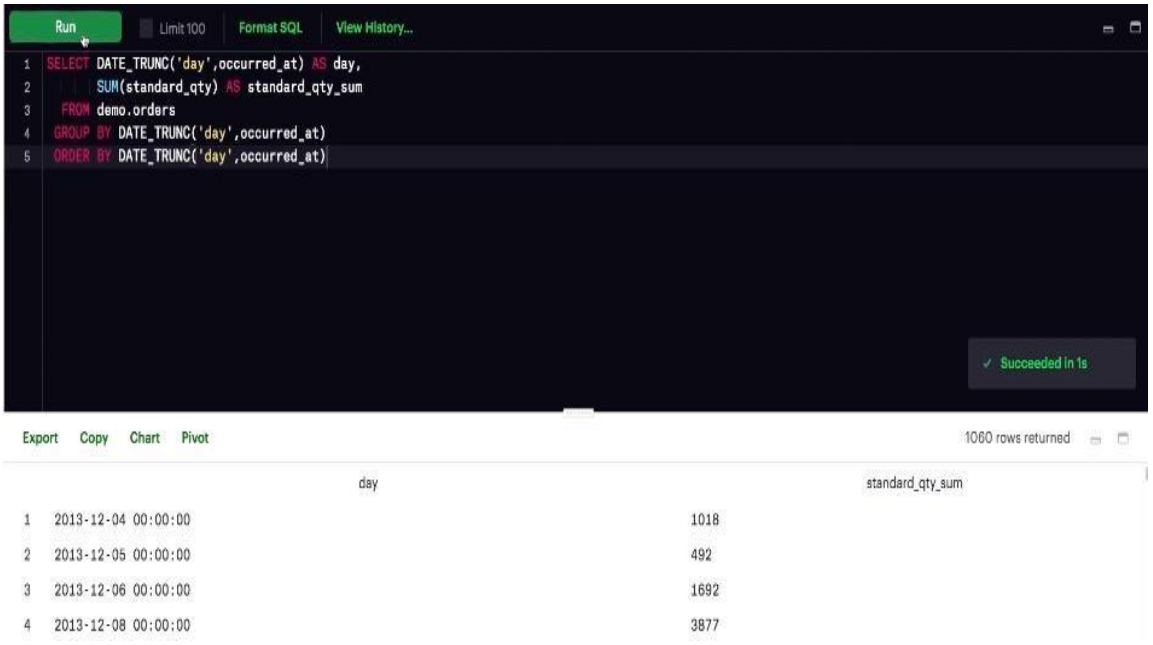
For example, in order to group by day, we'll need to adjust all the times on April 1st 2017 to read: 2017- 04 - 01 00:00:00. That way when we group by date, we get every event that occurred for all hours, minutes and seconds of April 1st. they'll all be grouped together into the same grouping.

We can do this using the date trunc function.

Let's start by grouping by occurred at without any truncation.



As you can see, this doesn't help us much at all. If we replace each instance of occurred at with a truncated version, we'll get result set that sums the quantities of standard paper by day. It's important to group by the same metric that's included in the select statement to assure your results are consisting. In some databases, it's possible to group by a year but truncate on day in the select statement.



This is something most people only do by accident as a provider results that are confusing and in many cases wrong for the type of question you're trying to answer. the easiest way to make sure you correctly is to use column numbers instead of retyping the exact functions. Date trunc can be used to aggregate at very granular levels like second. That might be useful if you're working with server logs or many events happen in give second.

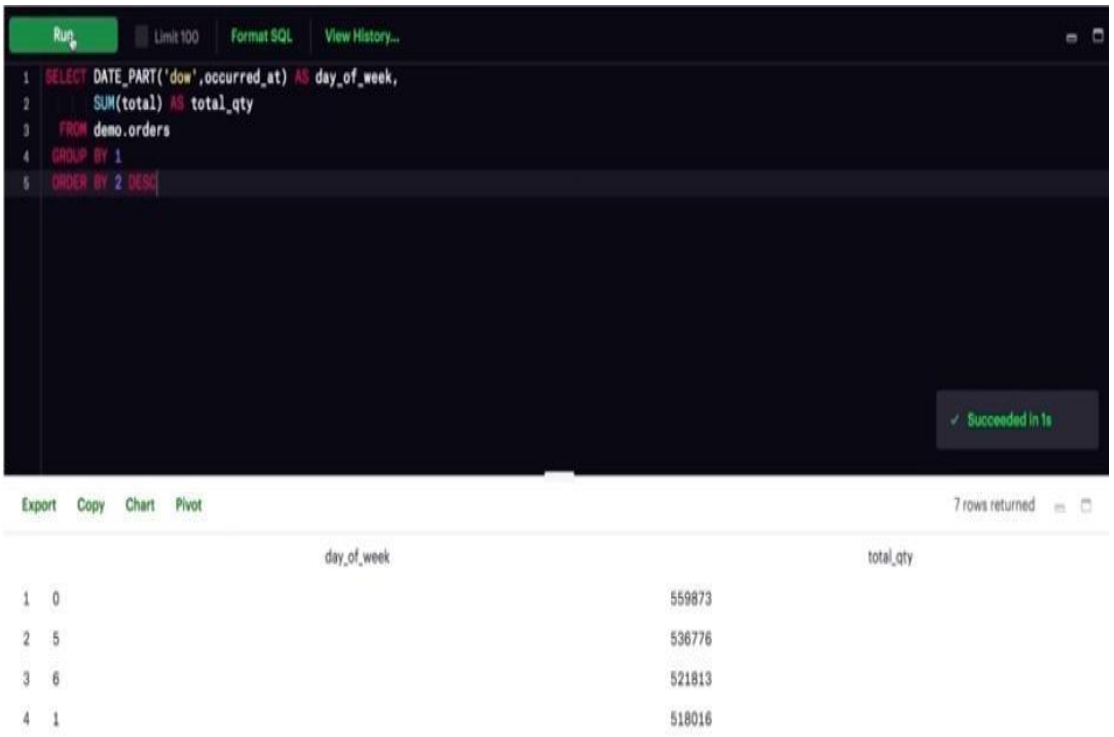
most of the time though, you'll use this to aggregate at intervals that make sense from a business perspective: day, week, month, quarter and year. Here, we can see different truncations. Notice that if our month or day are 01, There's no change made to these values, as you can see.

there are some cases where you might want to just pull out again part of the day. for example if you want to know what day of the week Parch & Posey's website sees the most traffic, you wouldn't want to use date trunk. To get the way of week, you'd have to use date part. Date part allows you to pull the part of the date that you're interested in. but notice that regardless of year, a date part would provide the same month for an event that happens in April 2016 and April 2017 where a date trunk would differentiate these events.

Let's explore this example using Parch & Posey's date. On what day of the week are the most sales made?

Let's start by figuring out the day of the week for each one. DOW stands for day of week and returns a value from zero to six, where zero is Sunday and six is Saturday.

Now that we have this column, we can aggregate to figure out the day with the most reams of paper sold. We'll order it by the sum descending order so that the day with the most sales will be at the top of the result set.



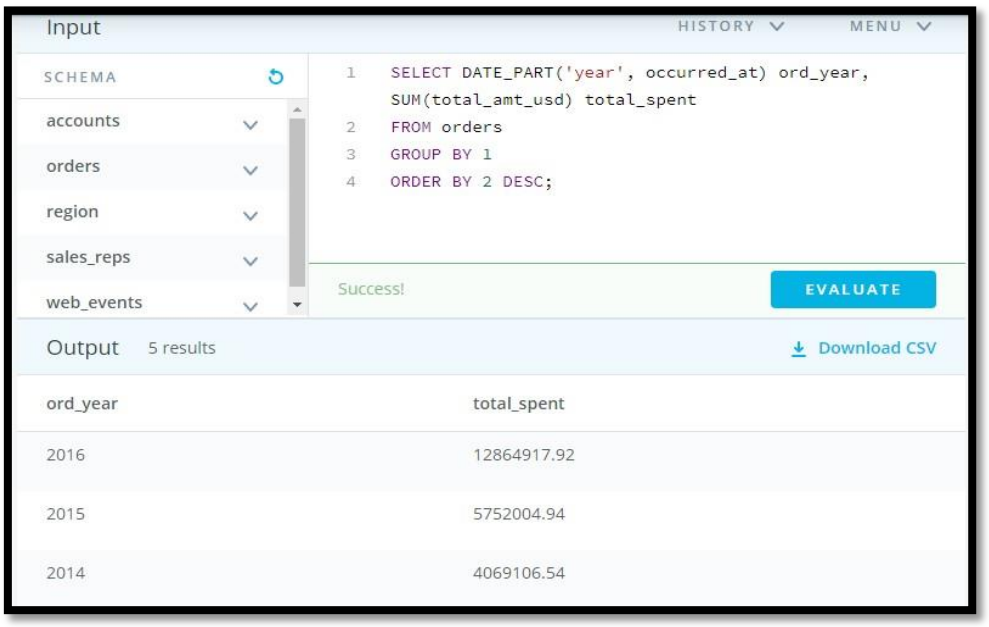
Look like the most paper was ordered on Sunday and the last paper was ordered on Thursday.

Quiz: DATE Functions

Questions: Working With DATEs

Use the **SQL** environment below to assist with answering the following questions. Whether you get stuck or you just want to double check your solutions, my answers can be found at the top of the next concept.

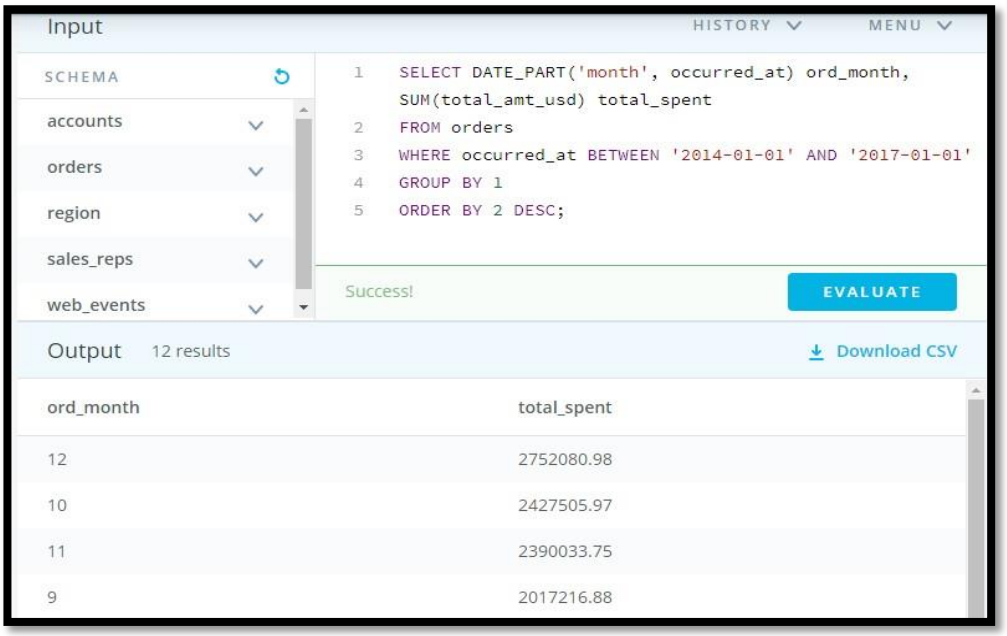
1. Find the sales in terms of total dollars for all orders in each year, ordered from greatest to least. Do you notice any trends in the yearly sales totals?



When we look at the yearly totals, you might notice that 2013 and 2017 have much smaller totals than all other years. If we look further at the monthly data, we see that for 2013 and 2017 there is only one month of sales for each of these years (12 for 2013 and 1 for 2017). Therefore, neither of these are evenly represented. Sales have been increasing year over year, with 2016 being the largest sales to date. At this rate, we might expect 2017 to have the largest sales.

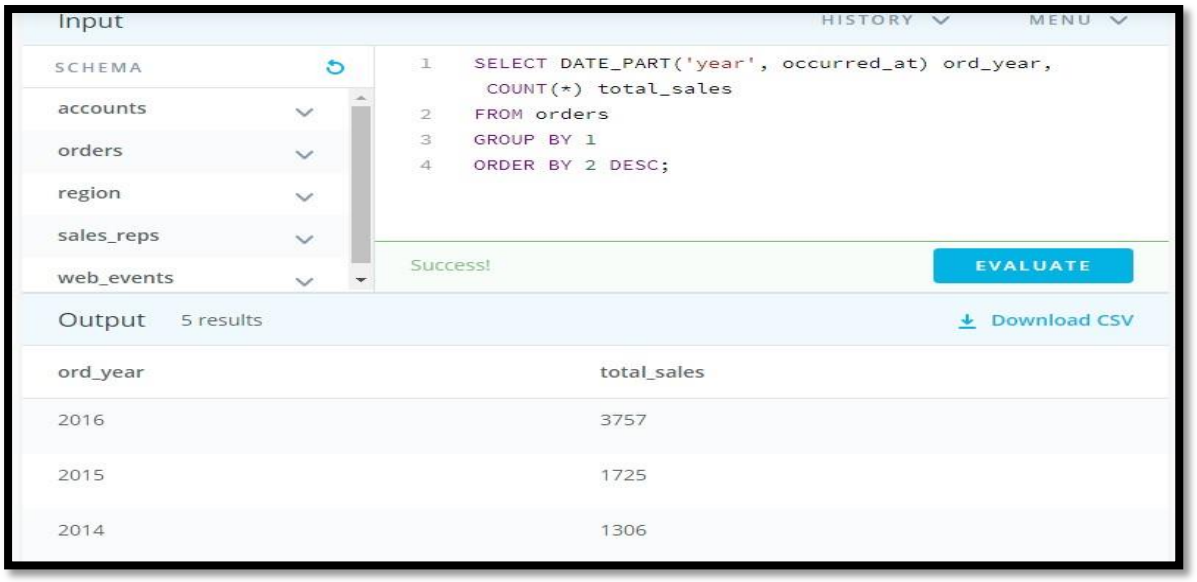
2. Which **month** did Parch & Posey have the greatest sales in terms of total dollars? Are all months evenly represented by the dataset?

   In order for this to be 'fair', we should remove the sales from 2013 and 2017. For the same reasons as discussed above.



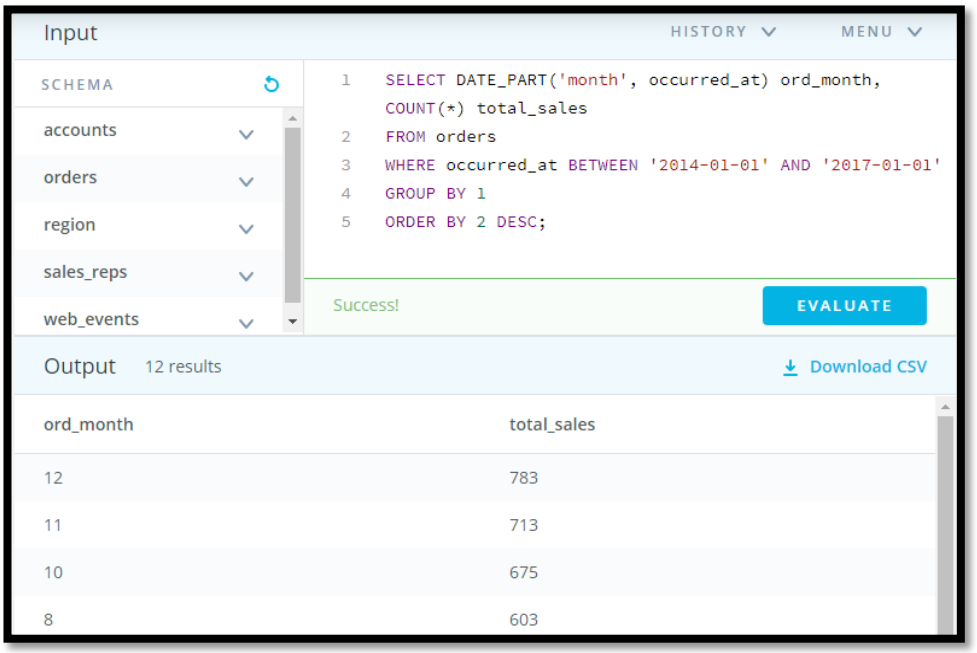
The greatest sales amounts occur in December (12).

3. Which **year** did Parch & Posey have the greatest sales in terms of total number of orders? Are all years evenly represented by the dataset?



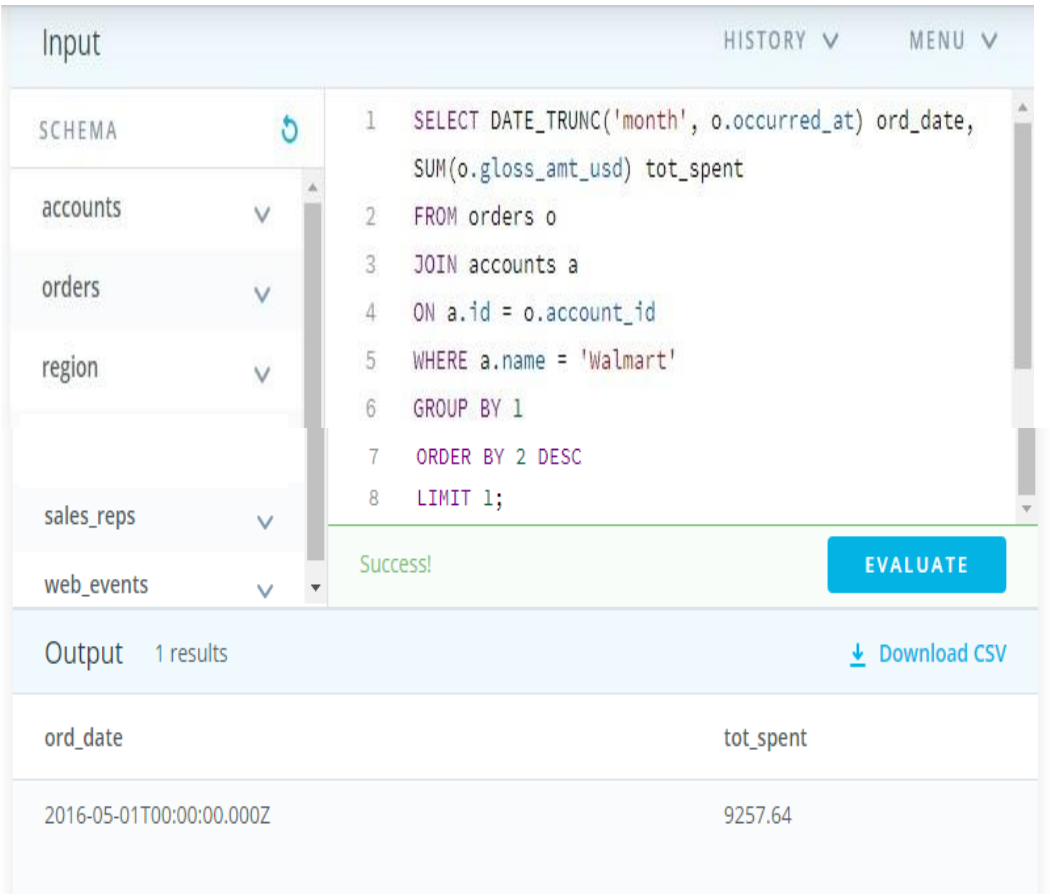
Again, 2016 by far has the most amount of orders, but again 2013 and 2017 are not evenly represented to the other years in the dataset.

4. Which **month** did Parch & Posey have the greatest sales in terms of total number of orders? Are all months evenly represented by the dataset?



December still has the most sales, but interestingly, November has the second most sales (but not the most dollar sales. To make a fair comparison from one month to another 2017 and 2013 data were removed.

5. In which **month** of which **year** did Walmart spend the most on gloss paper in terms of dollars?



May 2016 was when Walmart spent the most on gloss paper.

CASE Statements

CASE - Expert Tip

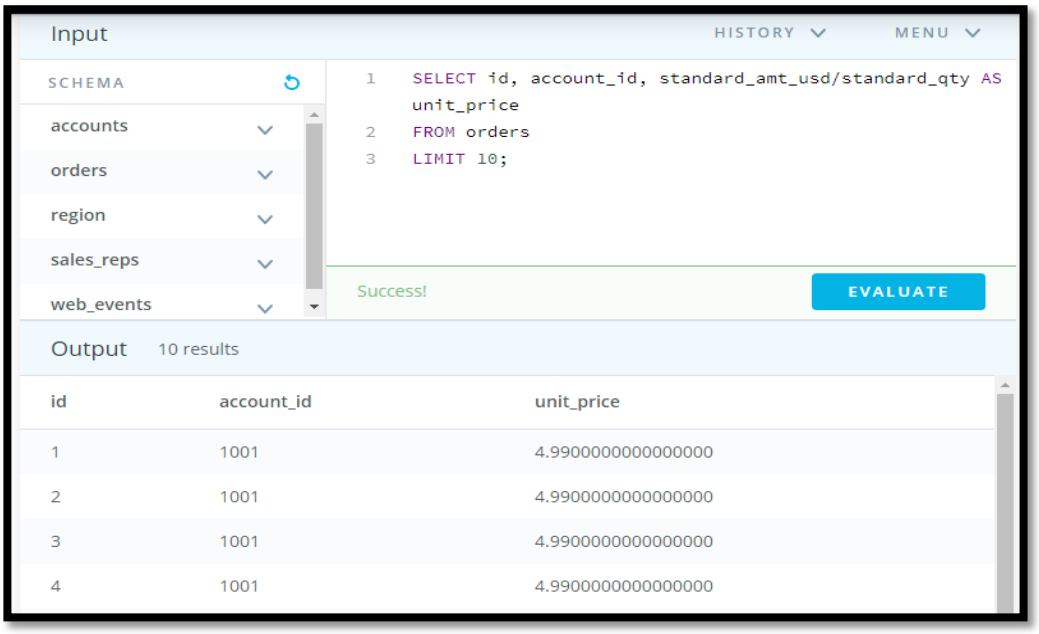
- The CASE statement always goes in the SELECT clause.
- CASE must include the following components: WHEN, THEN, and END. ELSE is an optional component to catch cases that didn't meet any of the other previous CASE conditions.
- You can make any conditional statement using any conditional operator (like **WHERE**) between WHEN and THEN. This includes stringing together multiple conditional statements using AND and OR.
- You can include multiple WHEN statements, as well as an ELSE statement again, to deal with any unaddressed conditions.

Example

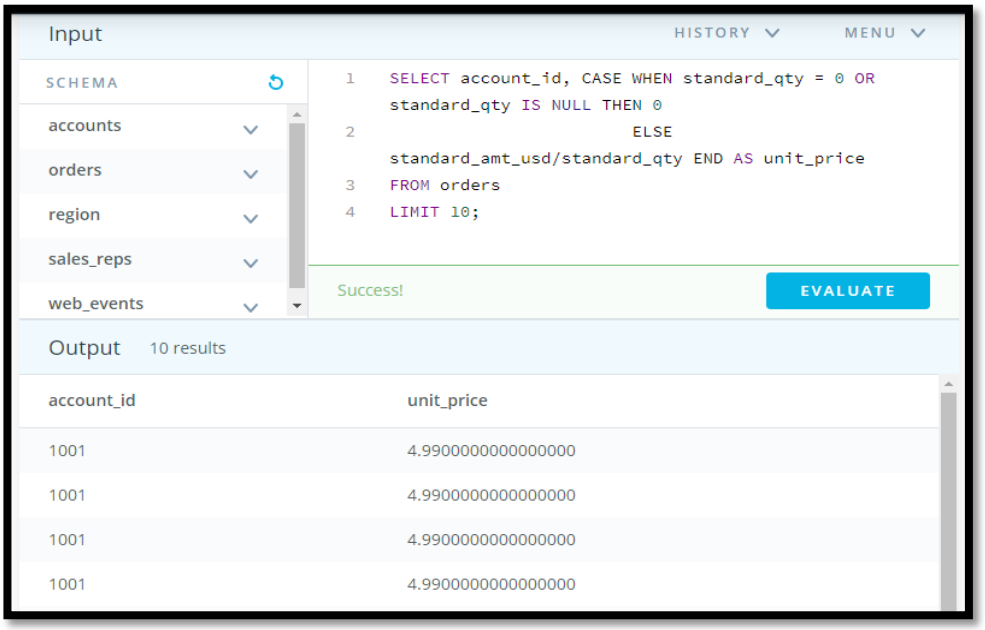
In a quiz question in the previous Basic SQL lesson, you saw this question:

1. Create a column that divides the standard_amt_usd by the standard_qty to find the unit price for standard paper for each order. Limit the results to the first 10 orders, and include the id and account_id fields. **NOTE - you will be thrown an error with the correct solution to this question. This is for a division by zero. You will learn how to get a solution without an error to this query when you learn about CASE statements in a later section.**

Let's see how we can use the **CASE** statement to get around this error.



Now, let's use a **CASE** statement. This way any time the **standard_qty** is zero, we will return 0, and otherwise we will return the **unit_price**.



Let's see how we can use the **CASE** statement to get around this error.
Now the first part of the statement will catch any of those division by zero values that were causing the error, and the other components will compute the division as necessary. You will notice, we essentially charge all of our accounts 4.99 for standard paper. It makes sense this doesn't fluctuate, and it is more accurate than adding 1 in the denominator like our quick fix might have been in the earlier lesson. We will see that.

CASE & Aggregations

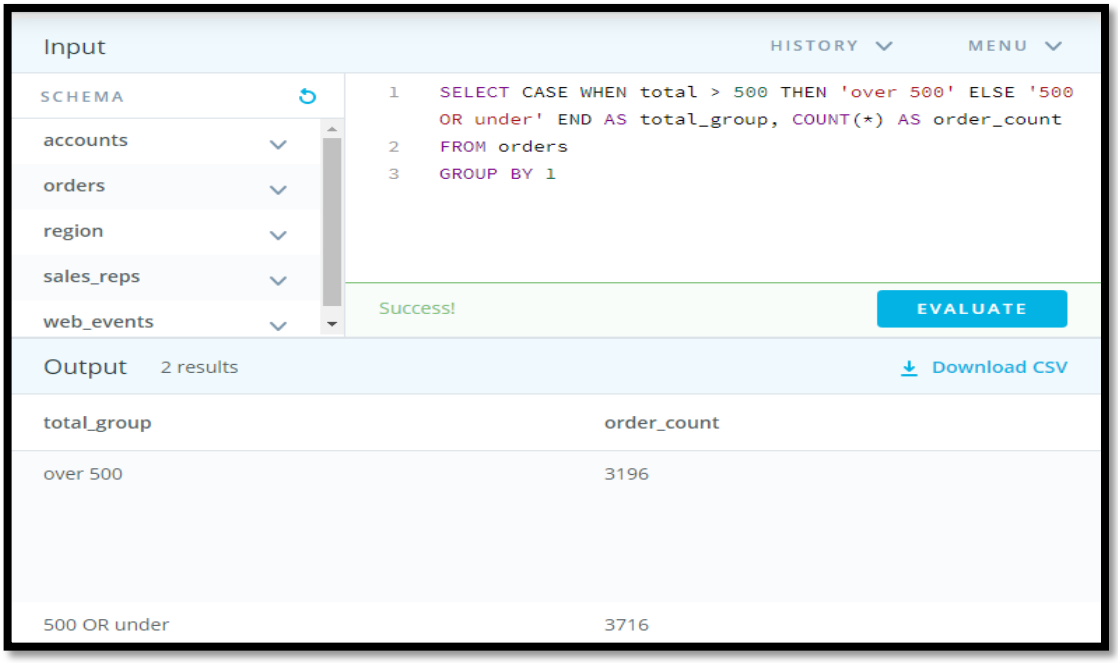
This one is pretty tricky. Try running the query yourself to make sure you understand what is happening. The next concept will give you some practice writing **CASE** statements on your own. In this section, we showed that getting the same information using a **WHERE** clause means only being able to get one set of data from the **CASE** at a time.

There are some advantages to separating data into separate columns like this depending on what you want to do, but often this level of separation might be easier to do in another programming language - rather than with SQL.

Example:

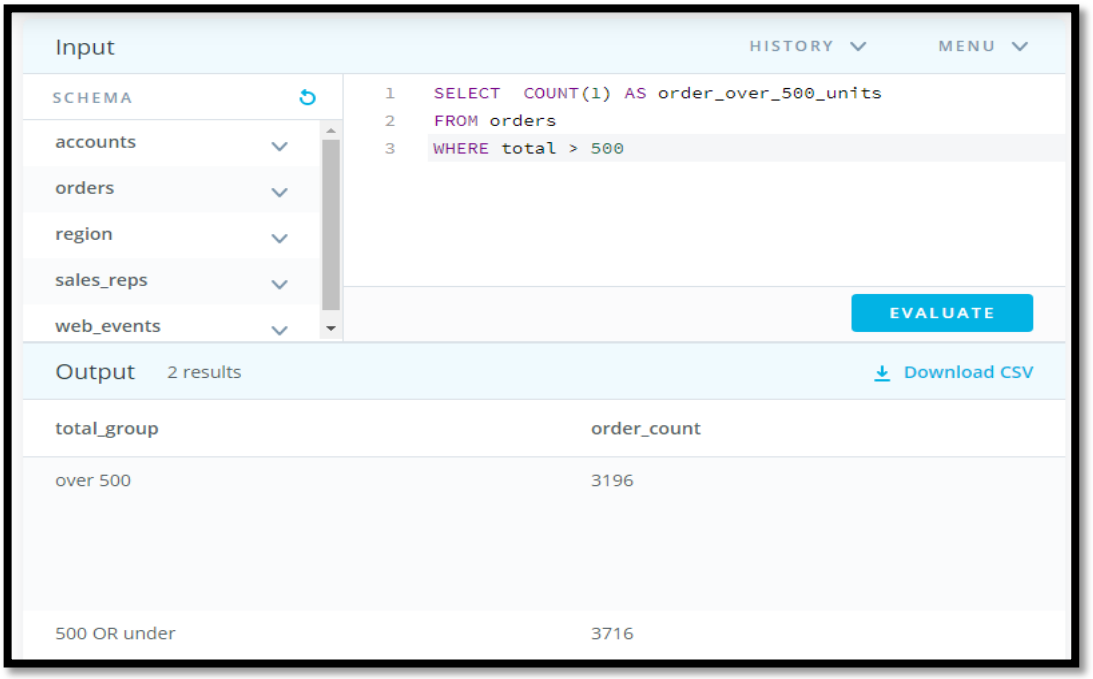
As a sales a manager at Parch & Posey, calssifying orders into general groups is a helpful exercise. But is much more useful if you can let it count up all the orders in each group. Aggregating based on these new will make it easier to report back to company leaders and take action. The easiers way to count all the members of a group is to create a column that group the way you want it to, then create another column to count by that group.

Here, we're using CASE to group orders into those with total quantity sold over 500, and those with 500 or less.



This query is an excellent place to use numbers instead of full column calculation in the GROUP BY clause, because repeating the CASE statement in the GROUP BY would make the query obnoxiously long. Now, you might be thinking, why woudn't I just use a WHERE clause to filter out rows I don’t want to count?

You could do that, and it would look like this.



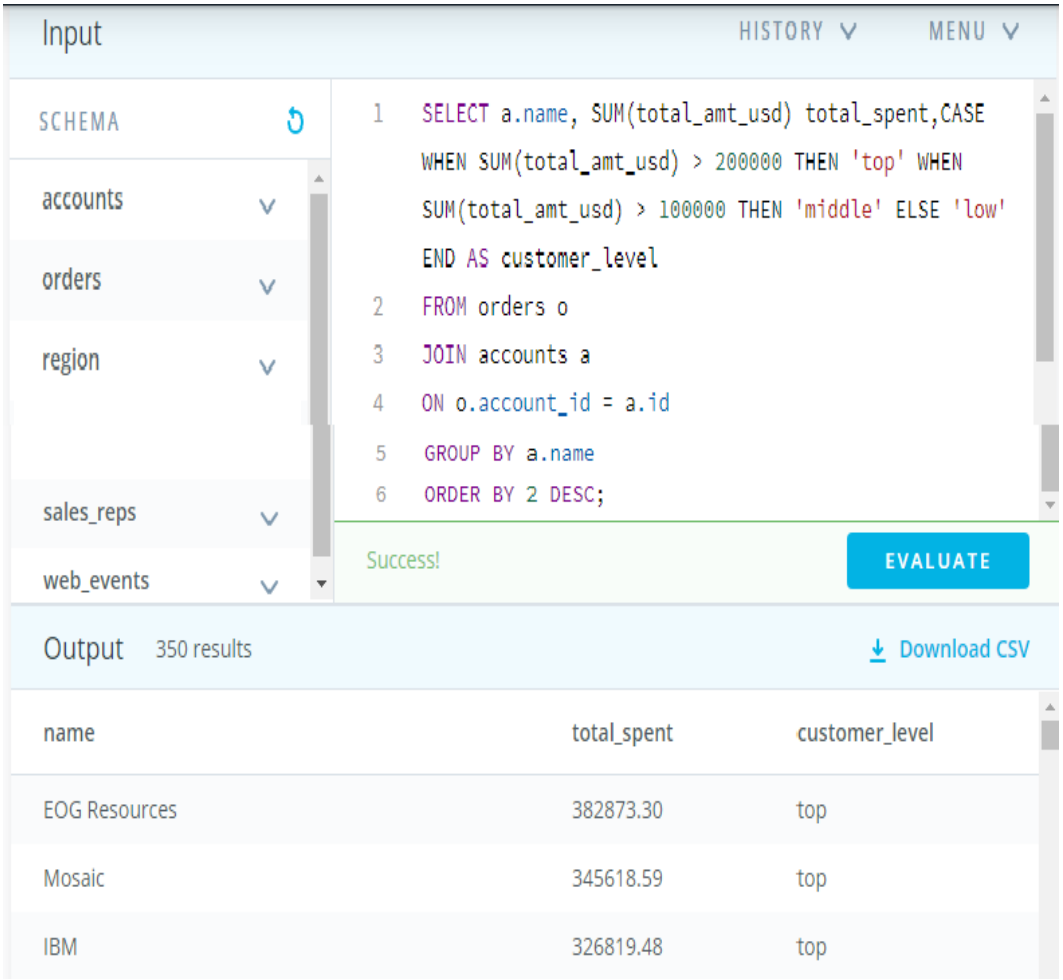
Unfortunately, using the WHERE clause only allows you to count one condition at a time. This would be tedious if we had a number of different cases. We would need a separate query for each one.

Quiz: CASE

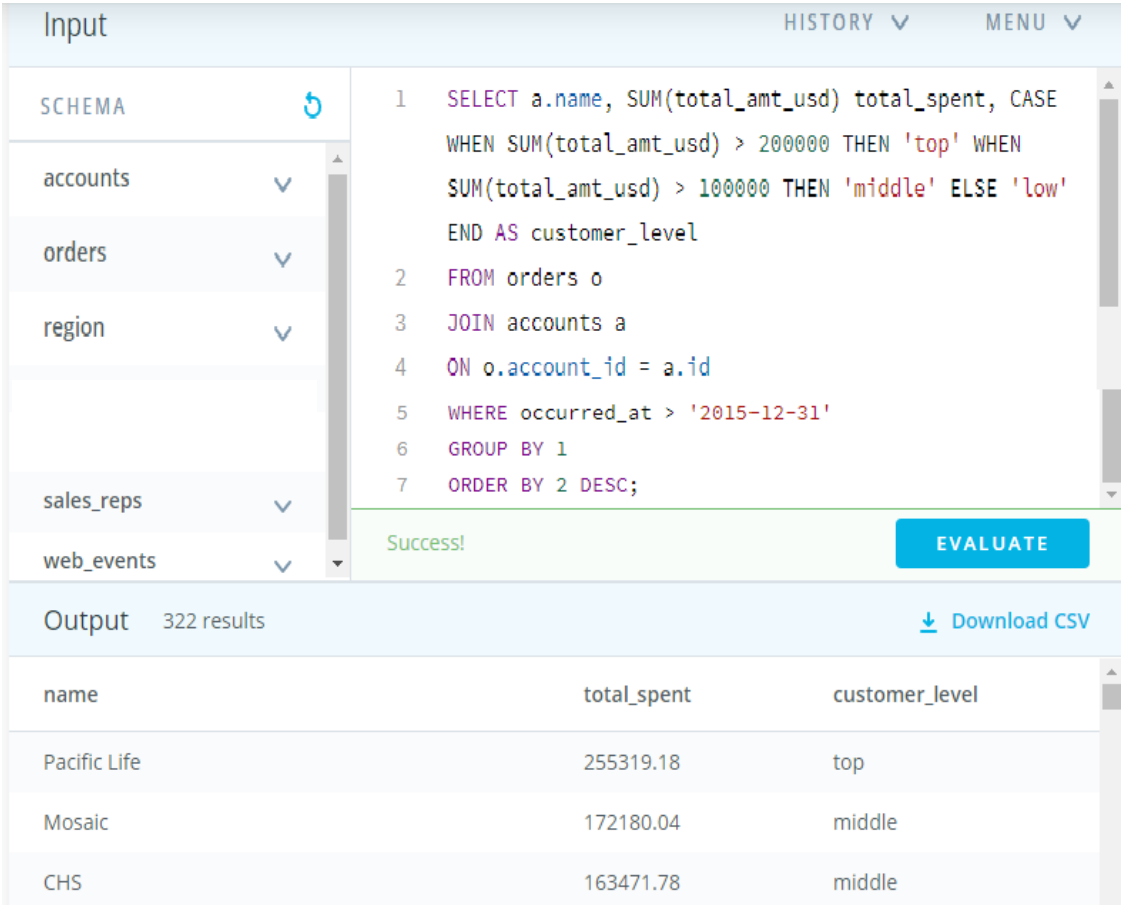
Questions: CASE

Use the **SQL** environment below to assist with answering the following questions.

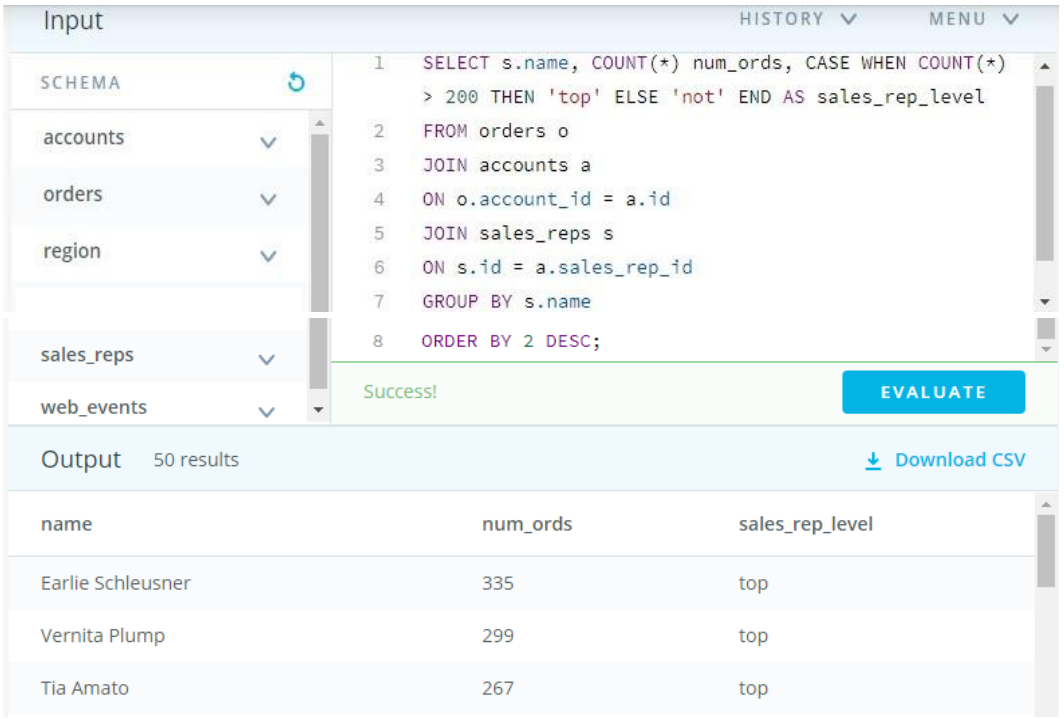
1. We would like to understand 3 different levels of customers based on the amount associated with their purchases. The top branch includes anyone with a Lifetime Value (total sales of all orders) greater than 200,000 usd. The second branch is between 200,000 and 100,000 usd. The lowest branch is anyone under 100,000 usd. Provide a table that includes the **level** associated with each **account**. You should provide the **account name**, the **total sales of all orders** for the customer, and the **level**. Order with the top spending customers listed first.



2. We would now like to perform a similar calculation to the first, but we want to obtain the total amount spent by customers only in 2016 and 2017. Keep the same **level**s as in the previous question. Order with the top spending customers listed first.

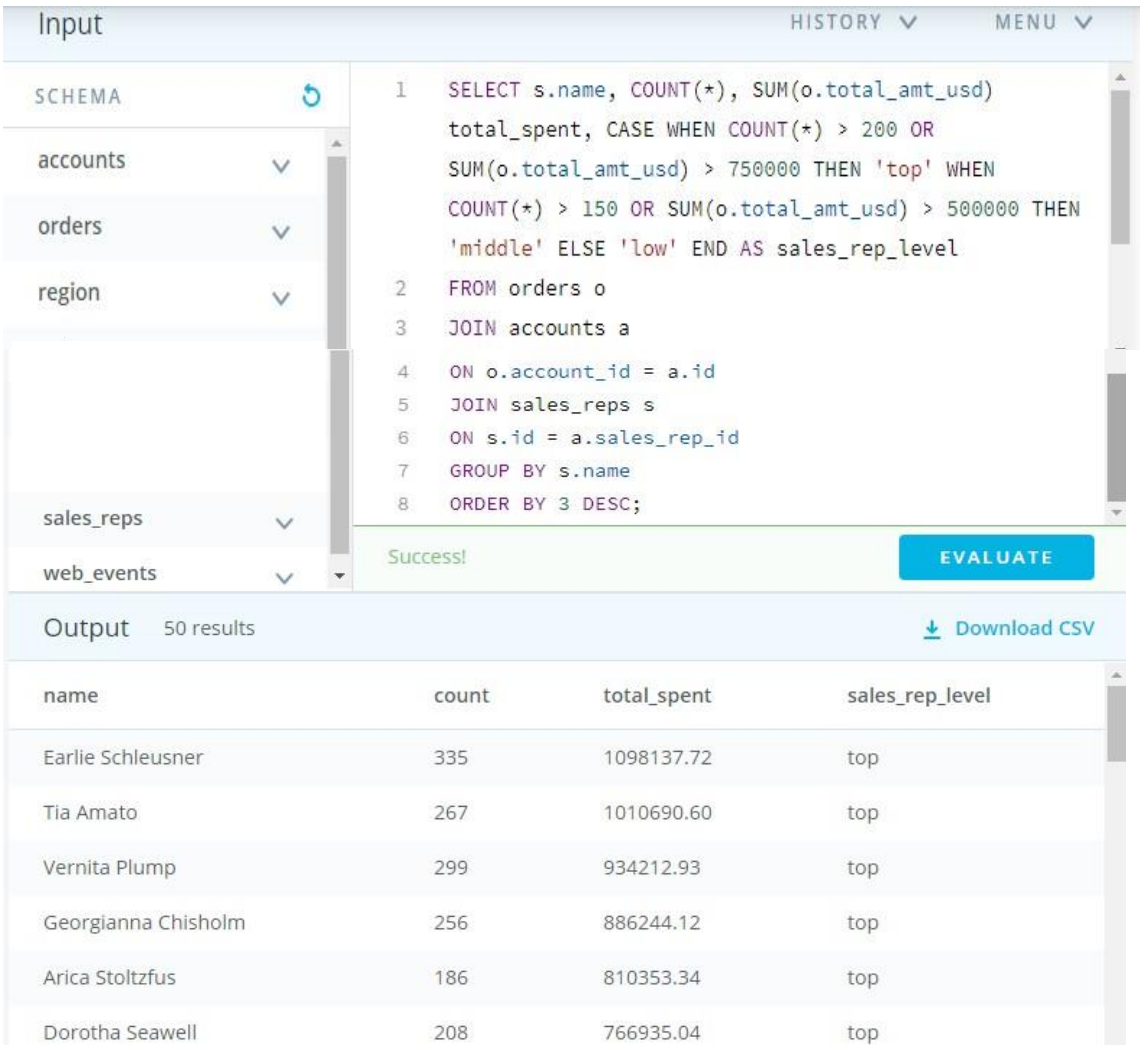


3. We would like to identify top performing **sales reps**, which are sales reps associated with more than 200 orders. Create a table with the **sales rep name**, the total number of orders, and a column with top or not depending on if they have more than 200 orders. Place the top sales people first in your final table.



It is worth mentioning that this assumes each name is unique - which has been done a few times. We otherwise would want to break by the name and the id of the table.

4. The previous didn't account for the middle, nor the dollar amount associated with the sales. Management decides they want to see these characteristics represented as well. We would like to identify top performing **sales reps**, which are sales reps associated with more than 200 orders or more than 750000 in total sales. The middle group has any **rep** with more than 150 orders or 500000 in sales. Create a table with the **sales rep name**, the total number of orders, total sales across all orders, and a column with top, middle, or low depending on this criteria. Place the top sales people based on dollar amount of sales first in your final table. You might see a few upset sales people by this criteria!



You might see a few upset sales people by this criteria !

Recap

Each of the sections has been labeled to assist if you need to revisit a particular topic. Intentionally, the solutions for a particular section are actually not in the labeled section, because my hope is this will force you to practice if you have a question about a particular topic we covered.

You have now gained a ton of useful skills associated with **SQL**. The combination of **JOINs** and **Aggregations** are one of the reasons **SQL** is such a powerful tool.
If there was a particular topic you struggled with, I suggest coming back and revisiting the questions with a fresh mind. The more you practice the better, but you also don't want to get stuck on the same problem for an extended period of time!

LESSON 29

Introduction

What this lesson is about...

Up to this point you have learned a lot about working with data using SQL. **This lesson will focus on three topics:**

1. Subqueries
2. Table Expressions
3. Persistent Derived Tables

Both **subqueries** and **table expressions** are methods for being able to write a query that creates a table, and then write a query that interacts with this newly created table. Sometimes the question you are trying to answer doesn't have an answer when working directly with existing tables in database.
However, if we were able to create new tables from the existing tables, we know we could query these new tables to answer our question. This is where the queries of this lesson come to the rescue.

If you can't yet think of a question that might require such a query, don't worry because you are about to see a whole bunch of them!

Introduction to Subqueries

Whenever we need to use existing tables to create a new table that we then want to query again, this is an indication that we will need to use some sort of **subquery**. In the next couple of concepts, we will walk through an example together. Then you will get some practice tackling some additional problems on your own.

Some queries, also known as inner queries or nested queries, are a tool for berforming operations in multiple steps.

For example, let's put our marketing manager hats back on. We'd like to know which channels send the most traffic per day on average to Parch & Posey.

In order to do that, we'll need to aggregate events by channel by day, then we need to take those and average them.

Write Your First Subquery

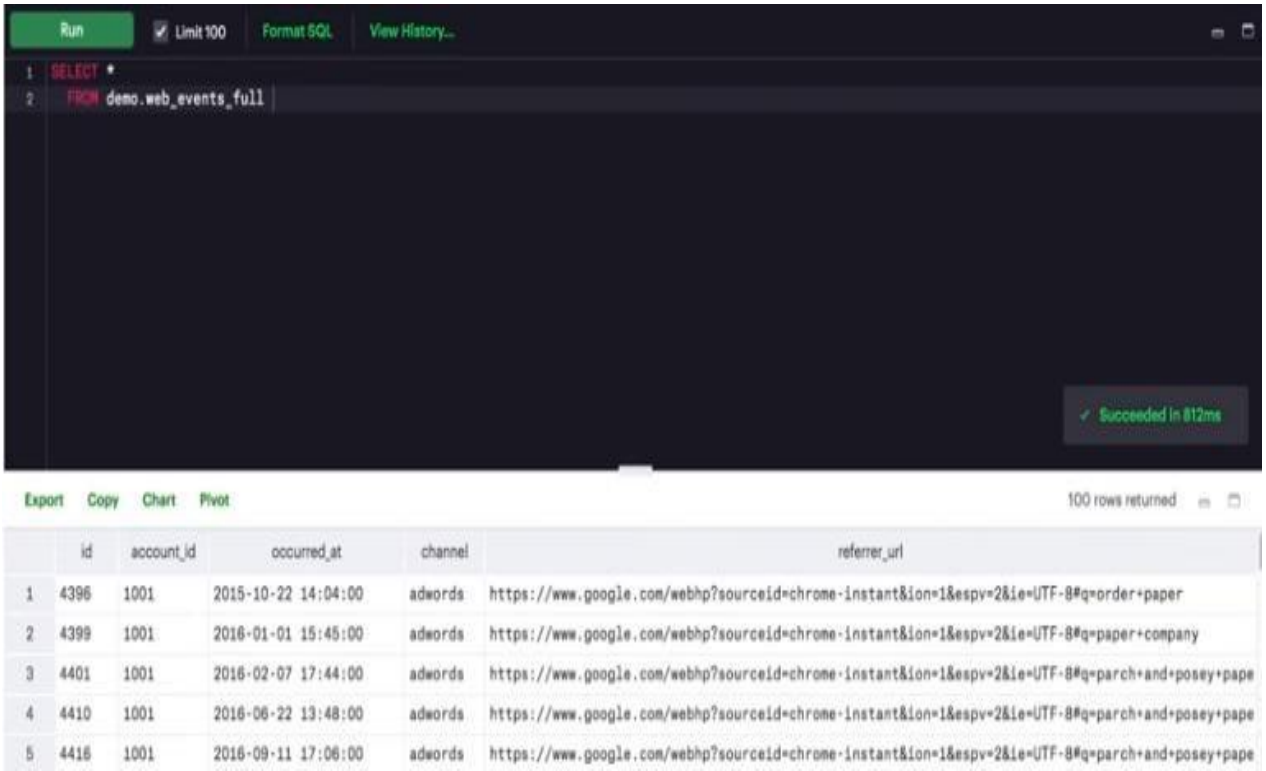
Your First Subquery

The first time you write a subquery it might seem really complex. Let's try breaking it down into its different parts.

We want to find the average number of events for each day for each channel. The first table will provide us the number of events for each day and channel, and then we will need to average these values together using a second query.

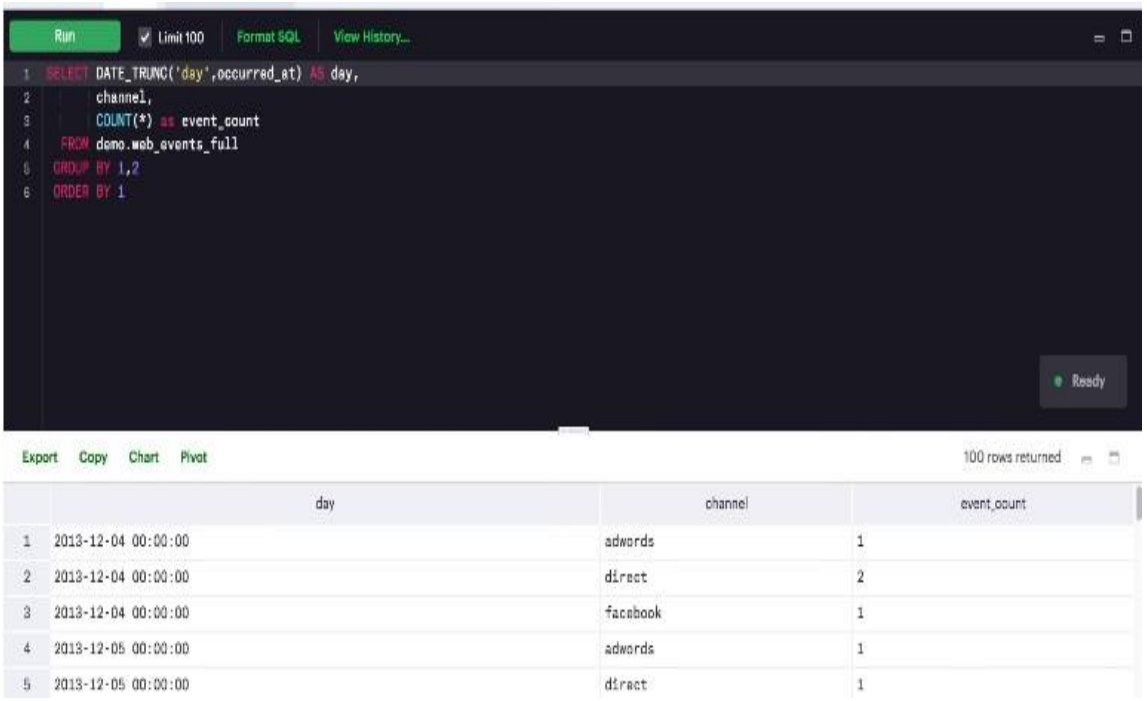
Example :

First, we'll start by querying the underlying table to make sure that date makes sense for what we're trying to do.



	id	account_id	occurred_at	channel	referrer_url
1	4396	1001	2015-10-22 14:04:00	adwords	https://www.google.com/webhp?sourceid=chrome-instant&ion=1&espv=2&ie=UTF-8#q=order+paper
2	4399	1001	2016-01-01 15:45:00	adwords	https://www.google.com/webhp?sourceid=chrome-instant&ion=1&espv=2&ie=UTF-8#q=paper+company
3	4401	1001	2016-02-07 17:44:00	adwords	https://www.google.com/webhp?sourceid=chrome-instant&ion=1&espv=2&ie=UTF-8#q=parch+and+posey+pape
4	4410	1001	2016-06-22 13:48:00	adwords	https://www.google.com/webhp?sourceid=chrome-instant&ion=1&espv=2&ie=UTF-8#q=parch+and+posey+pape
5	4416	1001	2016-09-11 17:06:00	adwords	https://www.google.com/webhp?sourceid=chrome-instant&ion=1&espv=2&ie=UTF-8#q=parch+and+posey+pape

Next, we'll count up all the events in each channel, in each day.



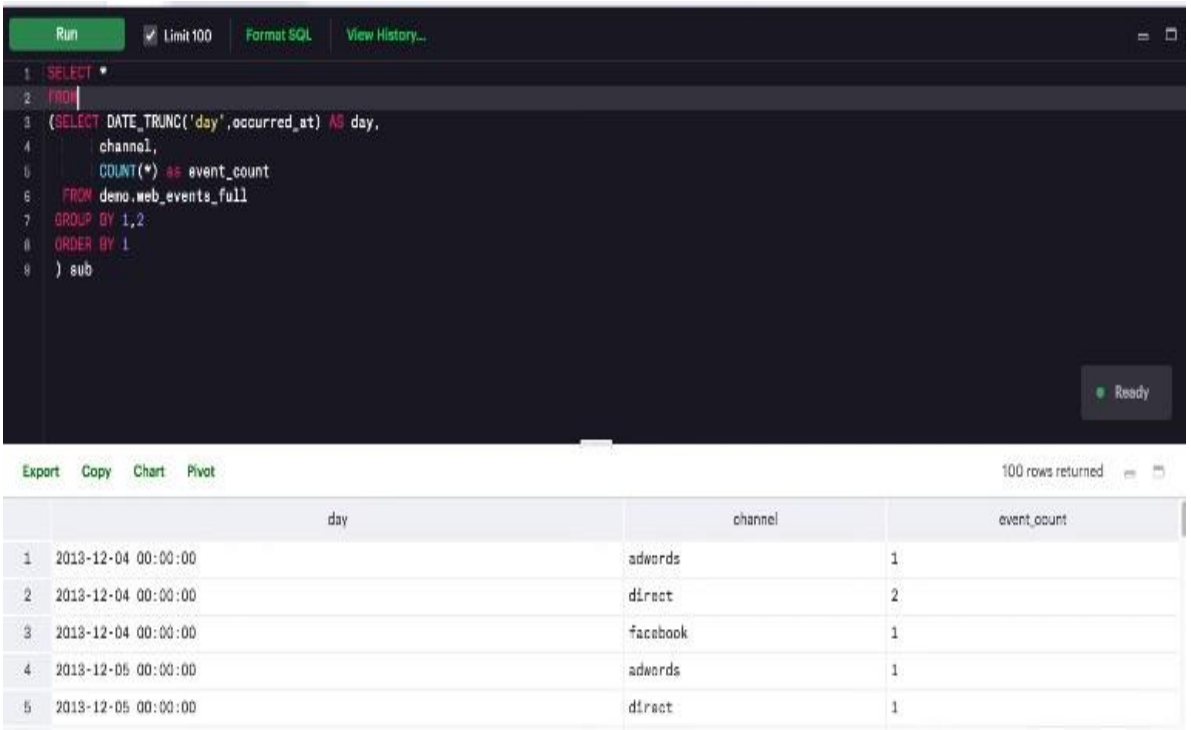
```
SELECT DATE_TRUNC('day',occurred_at) AS day,
       channel,
       COUNT(*) as event_count
  FROM demo.web_events_full
GROUP BY 1,2
ORDER BY 1
```

Run | Limit 100 | Format SQL | View History... | Ready

Export | Copy | Chart | Pivot | 100 rows returned

	day	channel	event_count
1	2013-12-04 00:00:00	adwords	1
2	2013-12-04 00:00:00	direct	2
3	2013-12-04 00:00:00	facebook	1
4	2013-12-05 00:00:00	adwords	1
5	2013-12-05 00:00:00	direct	1

The last step is that we want to average across the events column we've created. In order to do that, we'll want to query against the results from this query. We can do that by wrapping the query in parentheses and using it in the FROM clause of the next query that we write.



```
SELECT *
FROM
(SELECT DATE_TRUNC('day',occurred_at) AS day,
       channel,
       COUNT(*) as event_count
  FROM demo.web_events_full
GROUP BY 1,2
ORDER BY 1
) sub
```

Run | Limit 100 | Format SQL | View History... | Ready

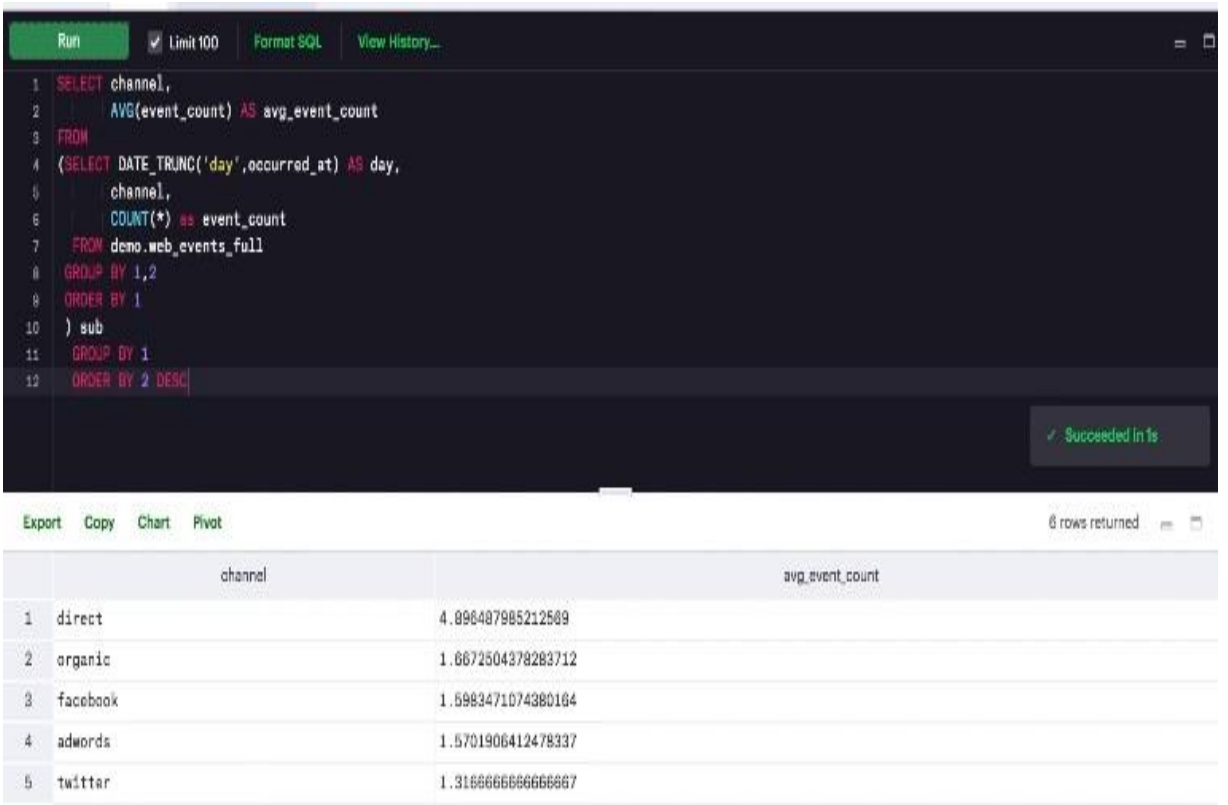
Export | Copy | Chart | Pivot | 100 rows returned

	day	channel	event_count
1	2013-12-04 00:00:00	adwords	1
2	2013-12-04 00:00:00	direct	2
3	2013-12-04 00:00:00	facebook	1
4	2013-12-05 00:00:00	adwords	1
5	2013-12-05 00:00:00	direct	1

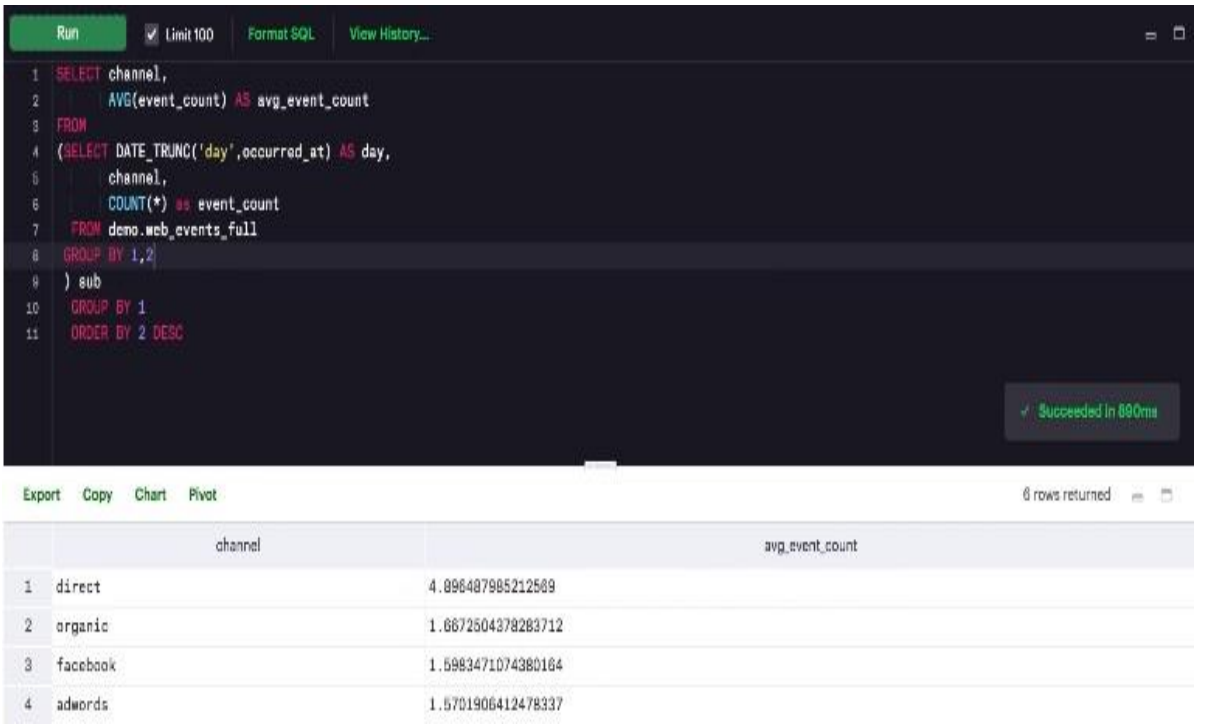
It is now a query within a query, also known as a subquery.

Subqueries are required to have aliases, which are added after the parentheses, the same way you would add an alias to a table. Here, we're just selecting all of the data from the subquery.

Let's go the last mile and average for each channel. Since the subquery acts like one table in the FROM clause, we'll put a GROUP BY clause after the subquery.



Since we're now reordering based on this new aggregation, we no longer need the ORDER BY statement in the subquery. So let's take that out to keep things clean.



Just to make it super clear, let's break down how this new query runs.

First, your inner query will run. It might sound like a no-brainer, but it's important. Your inner query must actually run on its own as the database will treat it as an independent query. Once your inner query is complete, the rest of the query also known as the outer query will run across the result set created by the inner query.

A nice feature that mode and many other SQL editors share is the ability to highlight a portion of the query and run only that portion. This is especially helpful when making changes to an inner query. You can make the change, than quickly check the inner queries output to make sure it looks correct before running the outer query again.

Quiz: Write Your First Subquery

Quiz 1

On which day-channel pair did the most events occur. (Mark all that are true)

- [x] January 1, 2017; direct
- [ ] December 31, 2016; facebook
- [ ] November 3, 2016; direct
- [x] December 21, 2016; direct

Quiz 2

Mark all of the below that are true regarding writing your subquery.

- [x] The original query goes in the **FROM** statement.
- [ ] The original query goes in the **SELECT** statement.
- [x] An * is used in the **SELECT** statement to pull all of the data from the original query.
- [x] You **MUST** use an alias for the table you nest within the outer query.

QUESTION 3 OF 3

Match each channel to its corresponding average number of events per day.

Submit to check your answer choices!

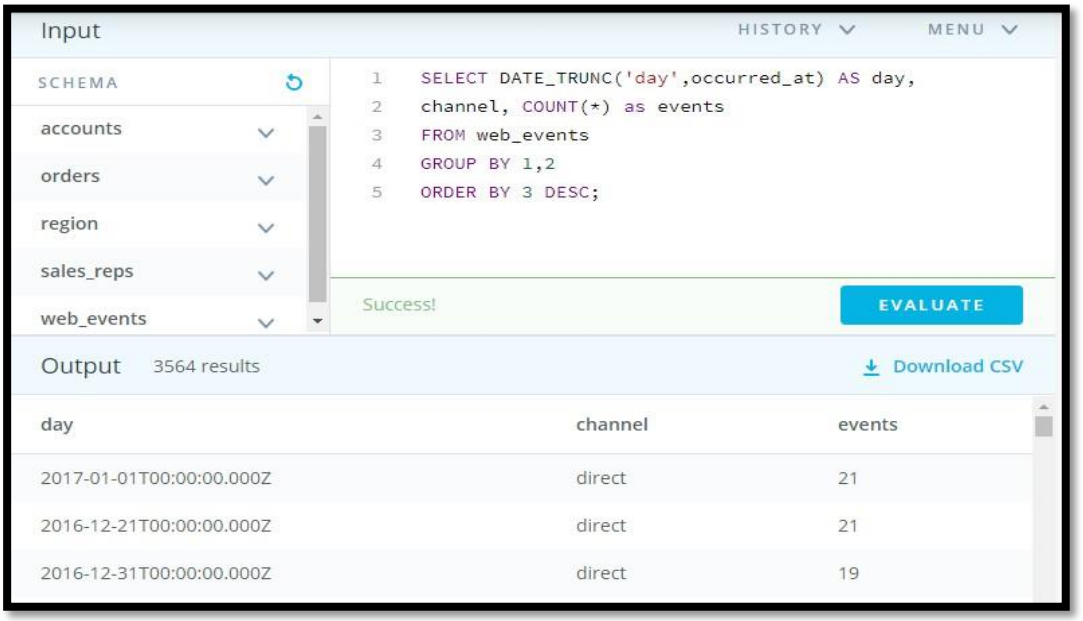
CHANNEL	AVERAGE NUMBER OF EVENTS/DAY
Direct	4.90
Facebook	1.60
Organic	1.67
Twitter	1.32

You try solving this yourself.

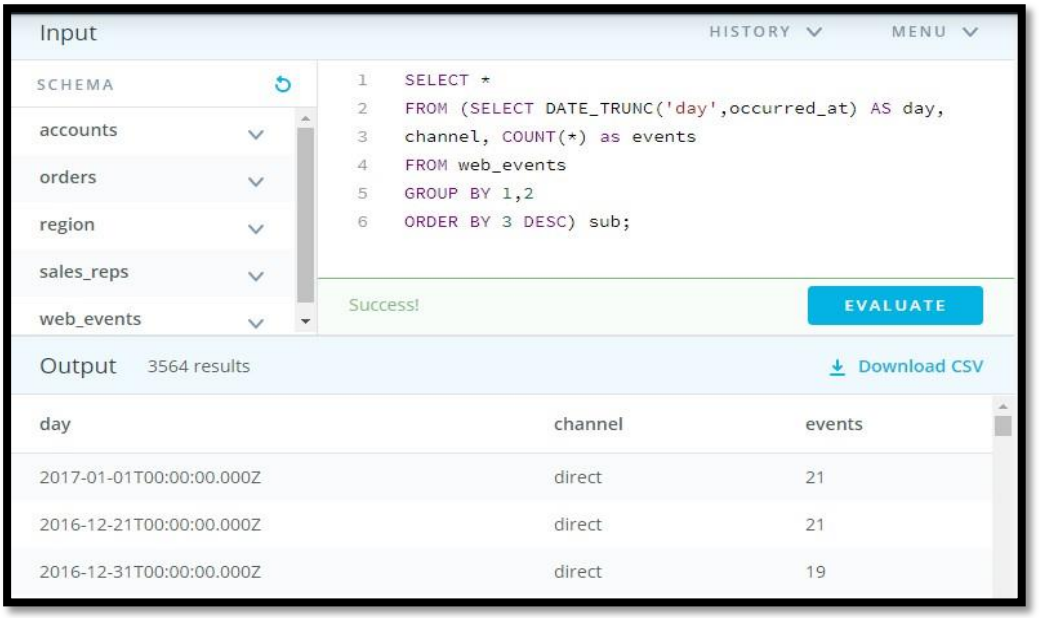
Tasks to complete.

Task List

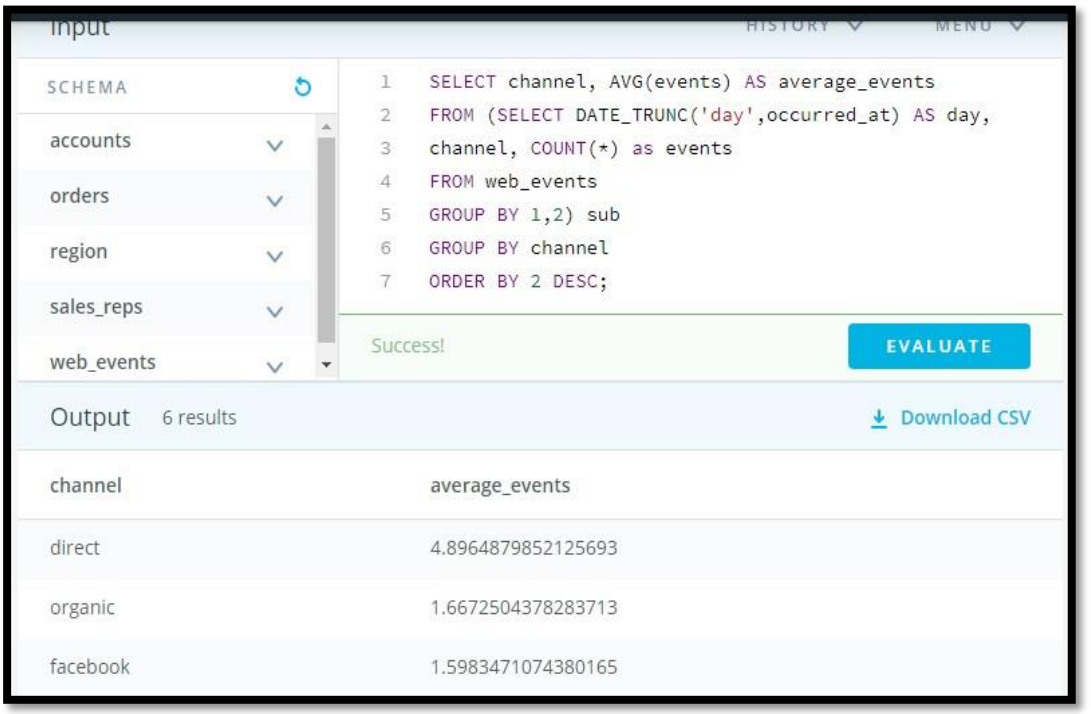
1. Use the test environment below to find the number of events that occur for each day for each channel.



2. Now create a subquery that simply provides all of the data from your first query.



3. Now find the average number of events for each channel. Since you broke out by day earlier, this is giving you an average per day.



Subquery Formatting

Subquery Formatting

When writing **Subqueries**, it is easy for your query to look incredibly complex. In order to assist your reader, which is often just yourself at a future date, formatting SQL will help with understanding your code.

The important thing to remember when using subqueries is to provide some way for the reader to easily determine which parts of the query will be executed together. Most people do this by indenting the subquery in some way - you saw this with the solution blocks in the previous concept.

The examples in this class are indented quite far—all the way to the parentheses. This isn't practical if you nest many subqueries, but in general, be thinking about how to write your queries in a readable way. Examples of the same query written multiple different ways is provided below. You will see that some are much easier to read than others.

Badly Formatted Queries

Though these poorly formatted examples will execute the same way as the well formatted examples, they just aren't very friendly for understanding what is happening!

Here is the first, where it is impossible to decipher what is going on:

```
SELECT * FROM (SELECT DATE_TRUNC('day',occurred_at) AS day, channel,
COUNT(*) as events FROM web_events GROUP BY 1,2 ORDER BY 3 DESC) sub;
```

This second version, which includes some helpful line breaks, is easier to read than that previous version, but it is still not as easy to read as the queries in the **Well Formatted Query** section.

```
SELECT *
FROM (
SELECT DATE_TRUNC('day',occurred_at) AS day,
channel, COUNT(*) as events
FROM web_events
GROUP BY 1,2
ORDER BY 3 DESC) sub;
```

Well Formatted Query

Now for a well formatted example, you can see the table we are pulling from much easier than in the previous queries.

```
SELECT *
FROM (SELECT DATE_TRUNC('day',occurred_at) AS day,
            channel, COUNT(*) as events
    FROM web_events
    GROUP BY 1,2
    ORDER BY 3 DESC) sub;
```

Additionally, if we have a **GROUP BY**, **ORDER BY**, **WHERE**, **HAVING**, or any other statement following our subquery, we would then indent it at the same level as our outer query.
The query below is similar to the above, but it is applying additional statements to the outer query, so you can see there are **GROUP BY** and **ORDER BY** statements used on the output are not tabbed. The inner query **GROUP BY** and **ORDER BY** statements are indented to match the inner table.

```
SELECT *
FROM (SELECT DATE_TRUNC('day',occurred_at) AS day,
            channel, COUNT(*) as events
     FROM web_events
     GROUP BY 1,2
     ORDER BY 3 DESC) sub
GROUP BY channel
ORDER BY 2 DESC;
```

These final two queries are so much easier to read!

More On Subqueries

Subqueries Part II

In the first subquery you wrote, you created a table that you could then query again in the **FROM** statement. However, if you are only returning a single value, you might use that value in a logical statement like **WHERE**, **HAVING**, or even **SELECT**- the value could be nested within a **CASE** statement.

On the next concept, we will work through this example, and then you will get some practice on answering some questions on your own.

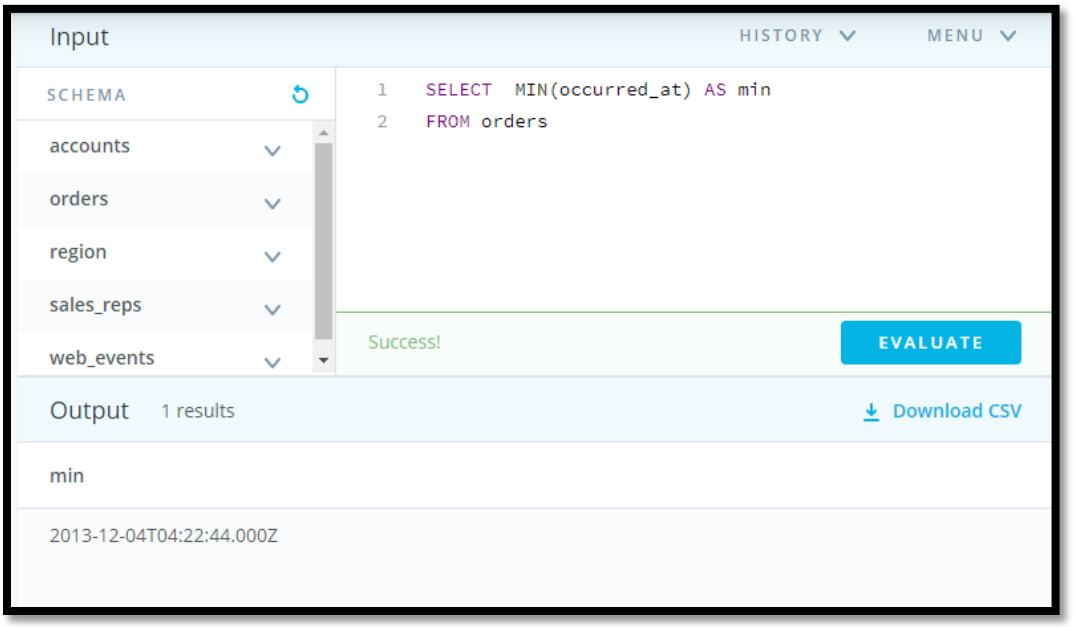
Expert Tip

Note that you should not include an alias when you write a subquery in a conditional statement. This is because the subquery is treated as an individual value (or set of values in the **IN** case) rather than as a table.

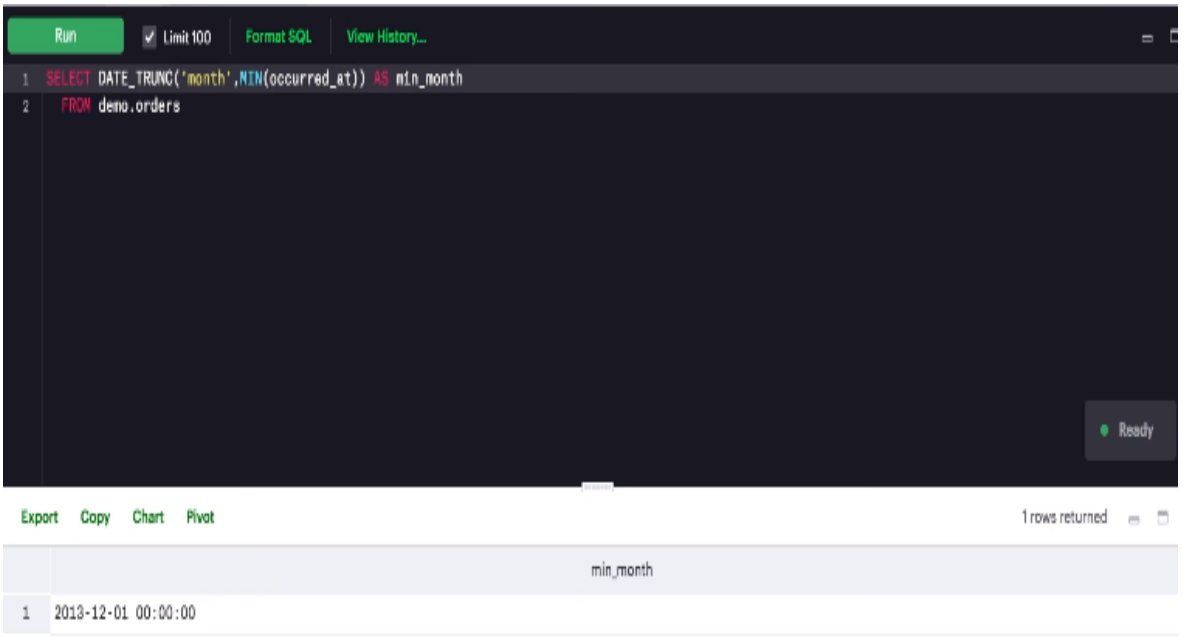
Also, notice the query here compared a single value. If we returned an entire column **IN** would need to be used to perform a logical argument. If we are returning an entire table, then we must use an **ALIAS** for the table, and perform additional logic on the entire table.

Subqueries can be used in several places within a query. It can really be used anywhere you might use a table name or even a column name or an individual value. They're especially useful in conditional logic, in conjunction with where or clauses, or in the when portion of a case statement.

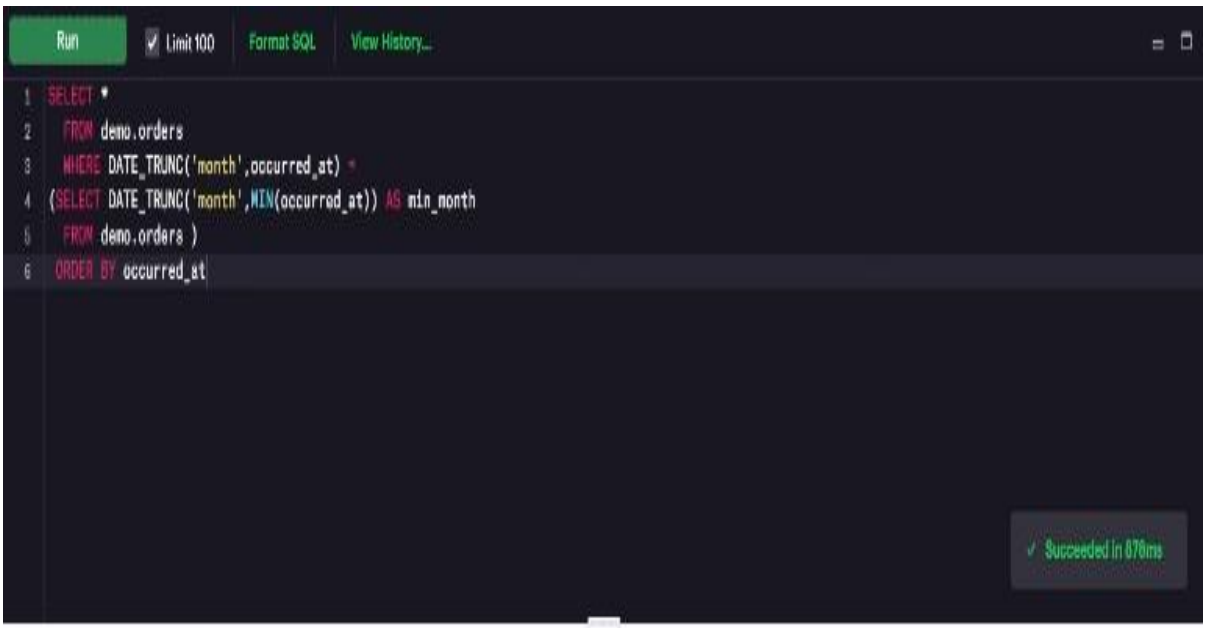
For example, you might want to return only orders the occurred in the same month as Parch & Poseies first order ever. To get the date of the first order, you can write a subquery with a min function.



Let's add a date_trunc function to get the month.



Finally, let's write an outer query that uses this to filter the orders table and sorts by the occurred at column.



Export Copy Chart Pivot 99 rows returned

	id	account_id	occurred_at	standard_qty	gloss_qty	poster_qty	total	standard_amt_usd	gloss_amt_usd	poster_amt_usd	total_amt_usd
1	5786	2861	2013-12-04 04:22:44	0	48	33	81	0	359.52	267.96	627.48
2	2415	2861	2013-12-04 04:45:54	490	15	11	516	2445.1	112.35	89.32	2648.77
3	4108	4311	2013-12-04 04:53:25	528	10	0	538	2634.72	74.9	0	2709.62
4	4489	1281	2013-12-05 20:29:16	0	37	0	37	0	277.13	0	277.13

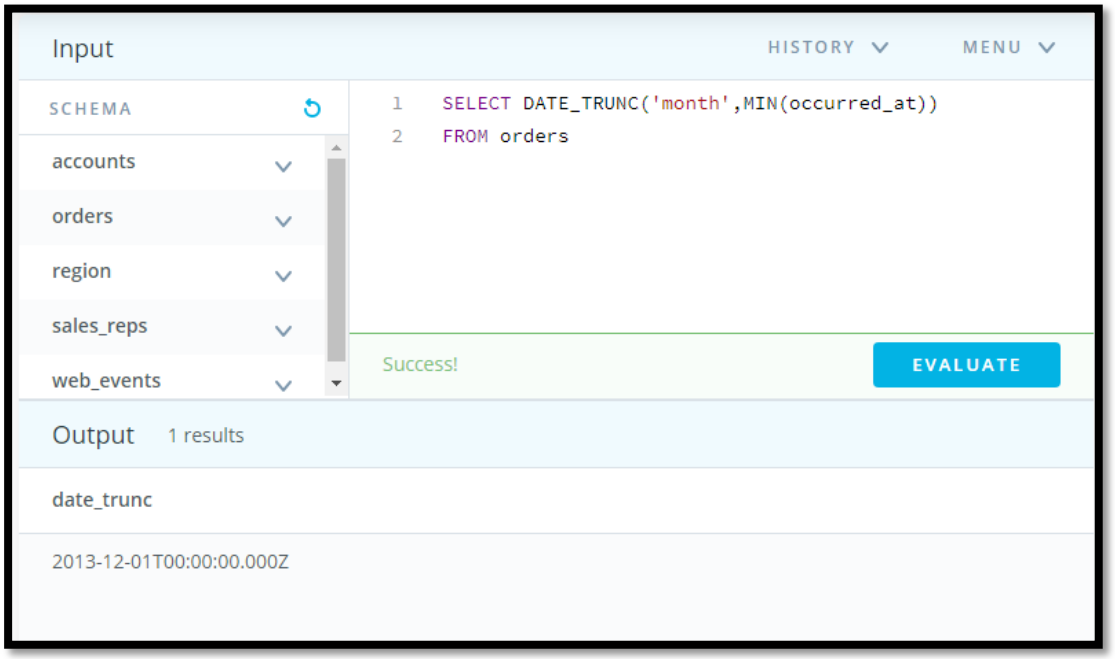
You can see that all of the results took place in December 2013, the same month as the first order. This query works because the result of the subquery is only one cell. Most conditional logic will work with subqueries containing one-cell results. But IN is the only type of conditional logic that will work when the inner query contains multiple results.

Quiz: More On Subqueries

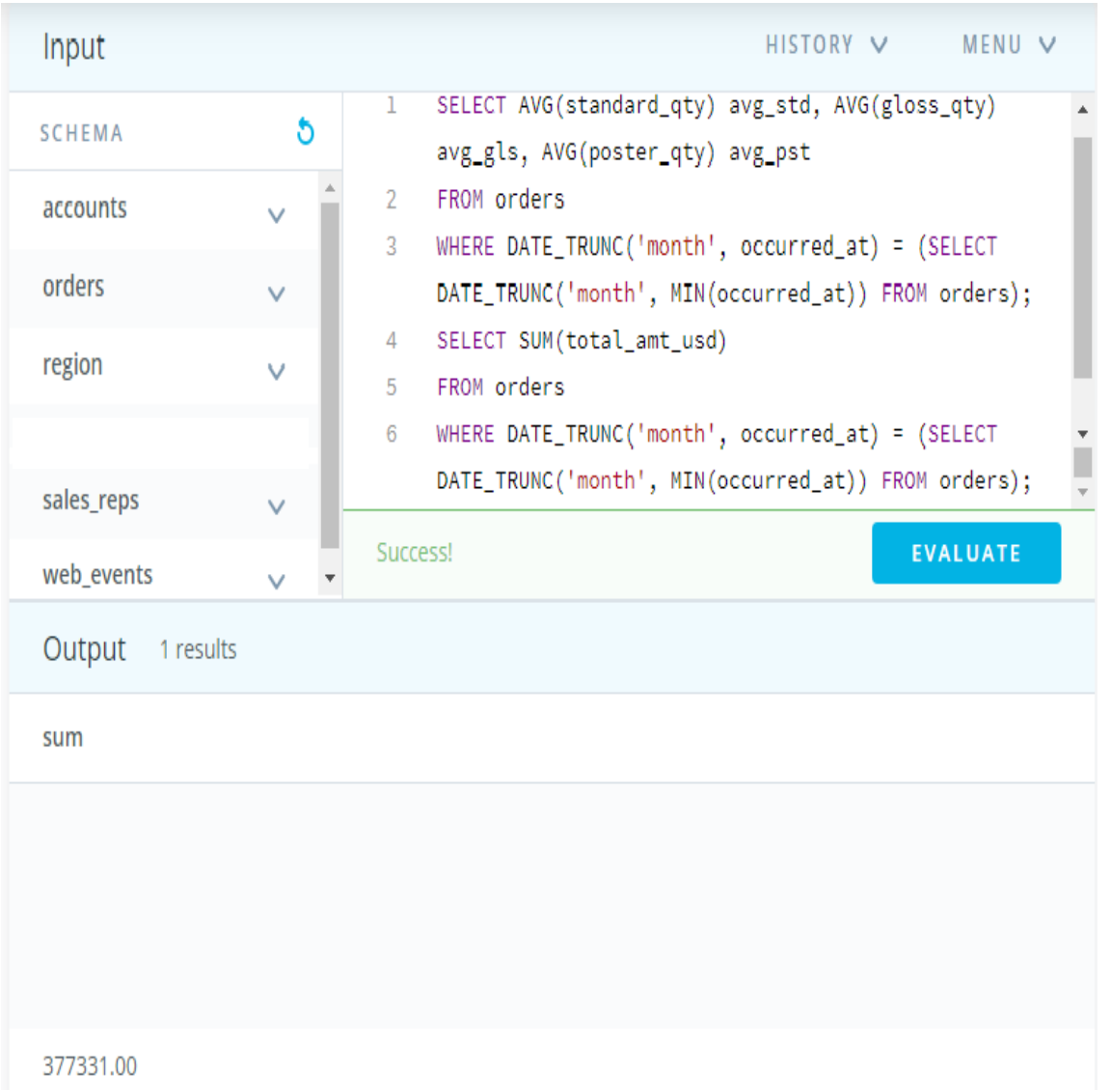
Use the task list below to work through the previous example.

Tasks to complete:

Use **DATE_TRUNC** to pull month level information about the first order ever placed in the **orders** table.



Use the result of the previous query to find only the orders that took place in the same month and year as the first order, and then pull the average for each type of paper qty in this month.



QUESTION 1 OF 2

What was the month/year combo for the first order placed?

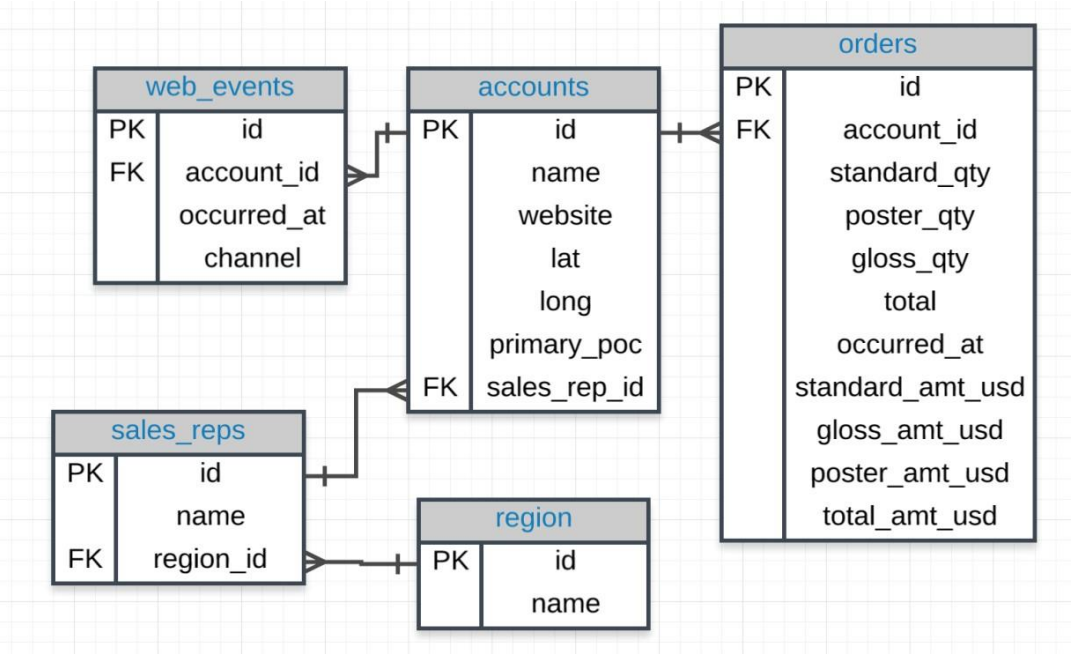
- October 2011
- October 2012
- December 2013
- November 2013

QUESTION 2 OF 2

Match each value to the corresponding description.

DESCRIPTION	VALUE
The average amount of standard paper sold on the first month that any order was placed in the **orders** table (in terms of quantity).	268
The average amount of gloss paper sold on the first month that any order was placed in the **orders** table (in terms of quantity).	209
The average amount of poster paper sold on the first month that any order was placed in the **orders** table (in terms of quantity).	112
The total amount spent on all orders on the first month that any order was placed in the **orders** table (in terms of usd).	377331

Quiz: Subquery Mania

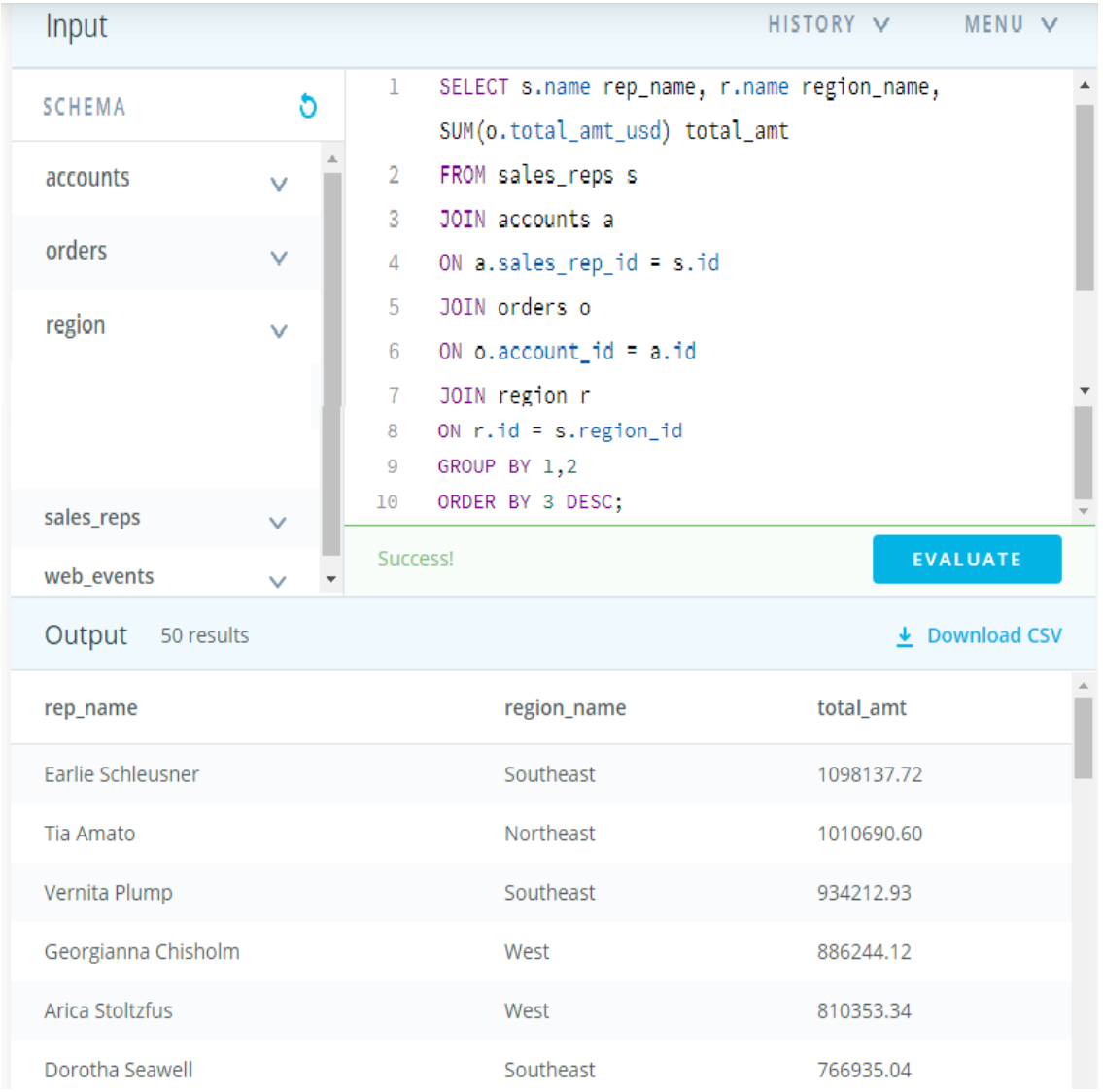


More Subqueries Quizzes

Above is the ERD for the database again - it might come in handy as you tackle the quizzes below. You should write your solution as a subquery, not by finding one solution and copying the output. The importance of this is that it allows your query to be dynamic in answering the question - even if the data changes, you still arrive at the right answer.

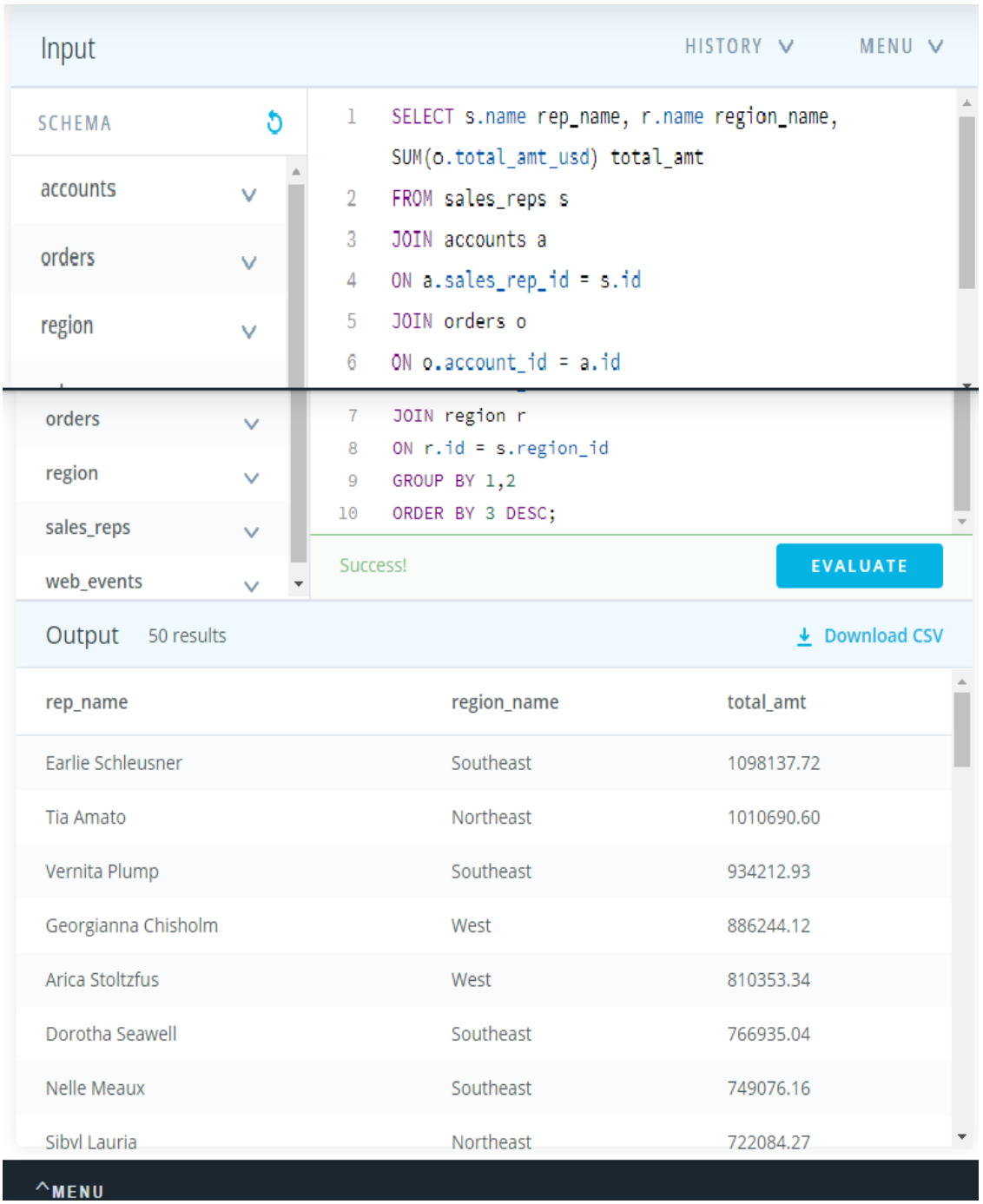
1. Provide the **name** of the **sales_rep** in each **region** with the largest amount of **total_amt_usd** sales.

First, I wanted to find the **total_amt_usd** totals associated with each **sales rep**, and I also wanted the region in which they were located. The query below provided this information.



rep_name	region_name	total_amt
Earlie Schleusner	Southeast	1098137.72
Tia Amato	Northeast	1010690.60
Vernita Plump	Southeast	934212.93
Georgianna Chisholm	West	886244.12
Arica Stoltzfus	West	810353.34
Dorotha Seawell	Southeast	766935.04

Next, I pulled the max for each region, and then we can use this to pull those rows in our final result.



Essentially, this is a **JOIN** of these two tables, where the region and amount match.

Input HISTORY MENU

SCHEMA

accounts
orders
orders
region
region
sales_reps
web_events

```
SELECT t3.rep_name, t3.region_name, t3.total_amt
FROM(SELECT region_name, MAX(total_amt) total_amt
     FROM(SELECT s.name rep_name, r.name region_name,
SUM(o.total_amt_usd) total_amt
FROM sales_reps s
JOIN accounts a
ON a.sales_rep_id = s.id
JOIN orders o
ON o.account_id = a.id
JOIN region r
ON r.id = s.region_id
GROUP BY 1, 2) t1
GROUP BY 1) t2
JOIN (SELECT s.name rep_name, r.name region_name,
SUM(o.total_amt_usd) total_amt
FROM sales_reps s
JOIN accounts a
ON a.sales_rep_id = s.id
JOIN orders o
ON o.account_id = a.id
JOIN region r
ON r.id = s.region_id
GROUP BY 1,2
```

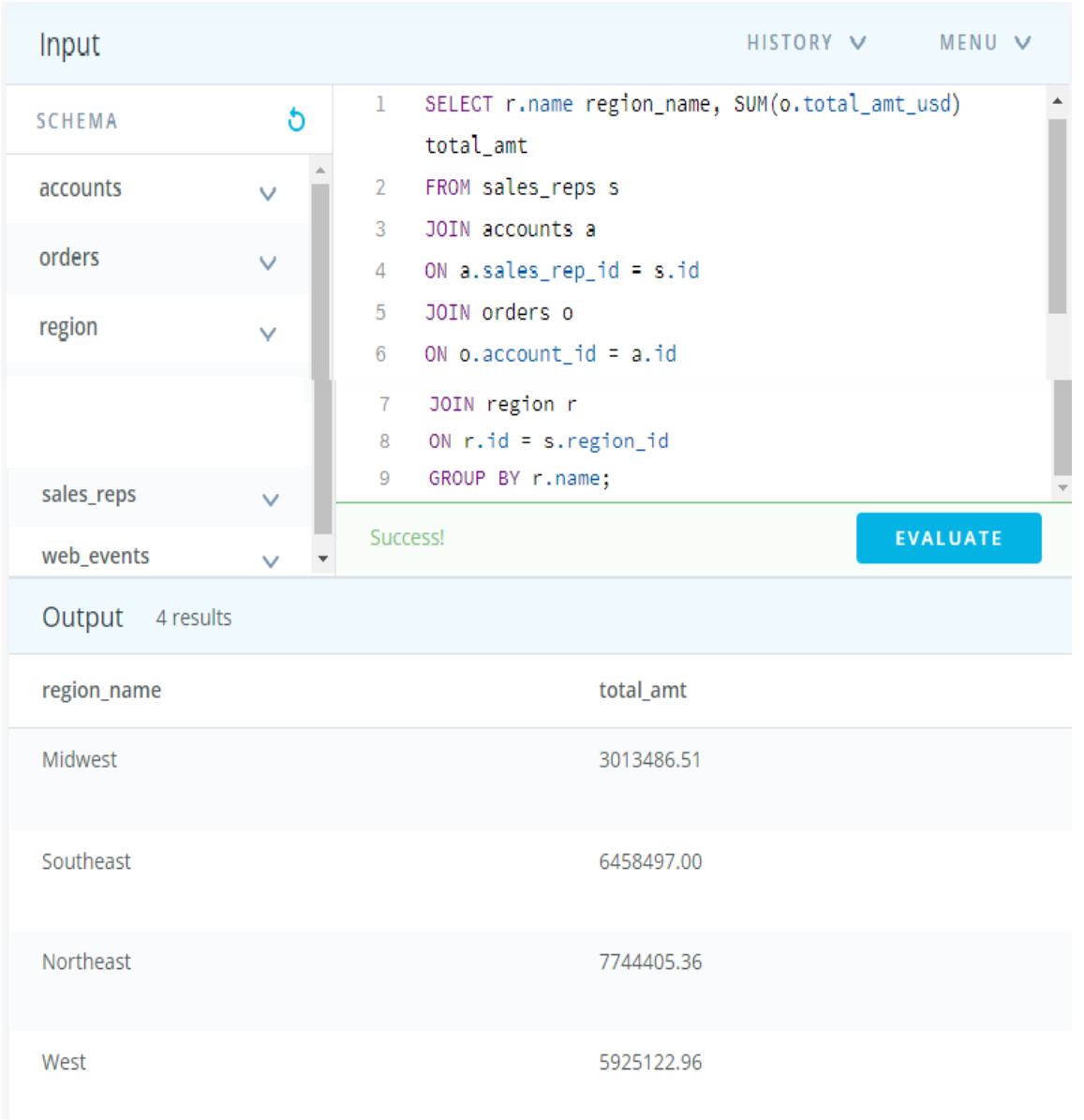
EVALUATE

Output 4 results Download CSV

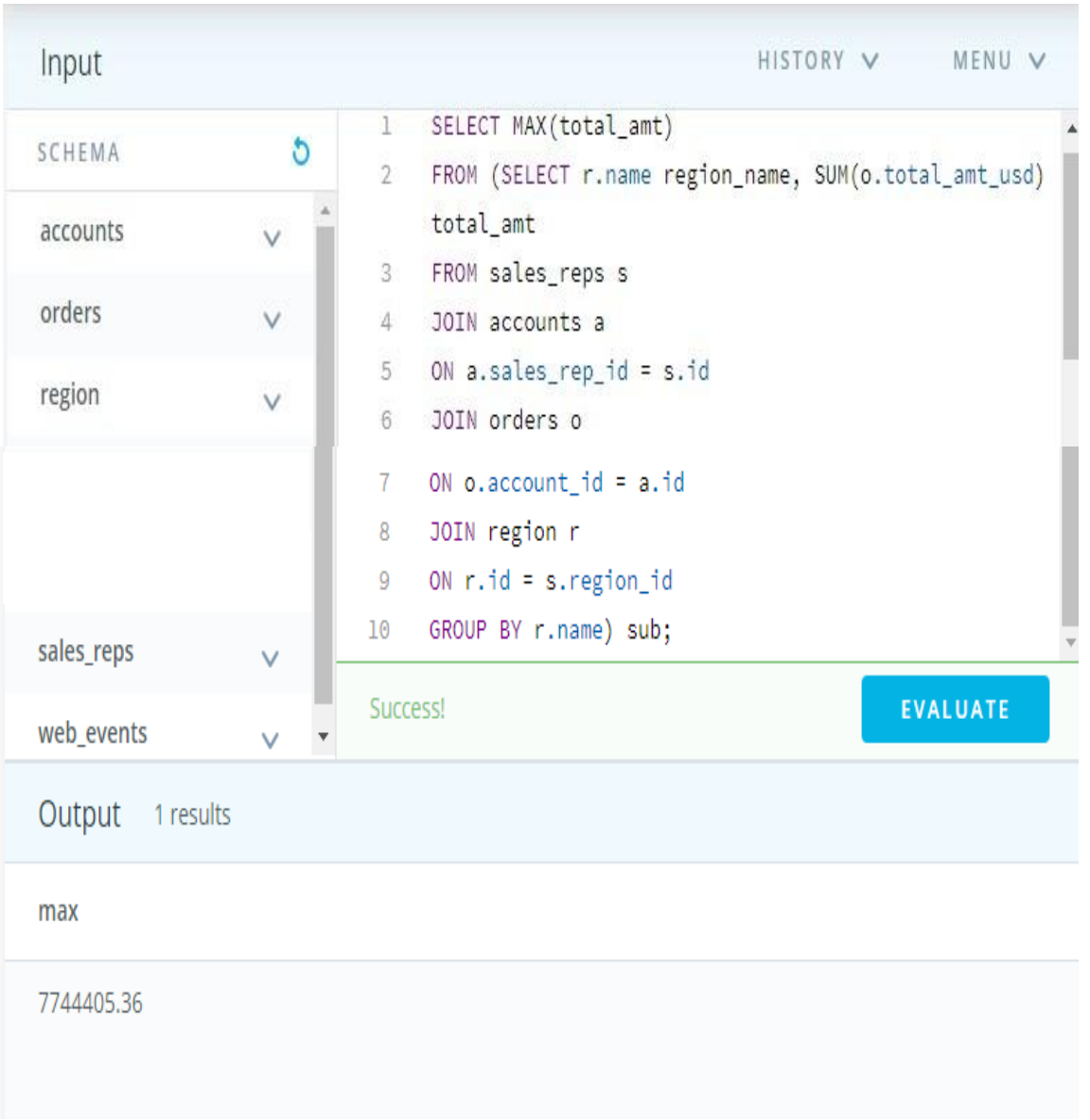
rep_name	region_name	total_amt
Earlie Schleusner	Southeast	1098137.72
Tia Amato	Northeast	1010690.60
Georgianna Chisholm	West	886244.12
Charles Bidwell	Midwest	675637.19

2. For the region with the largest (sum) of sales **total_amt_usd**, how many **total** (count) orders were placed?

The first query I wrote was to pull the **total_amt_usd** for each **region**.



Then we just want the region with the max amount from this table. There are two ways I considered getting this amount. One was to pull the max using a subquery. Another way is to order descending and just pull the top value.



Finally, we want to pull the total orders for the region with this amount:

Input HISTORY MENU

SCHEMA

accounts

orders

sales_reps

web_events

```
SELECT r.name, COUNT(o.total) total_orders
FROM sales_reps s
JOIN accounts a
ON a.sales_rep_id = s.id
JOIN orders o
ON o.account_id = a.id
JOIN region r
ON r.id = s.region_id
GROUP BY r.name
HAVING SUM(o.total_amt_usd) = (
      SELECT MAX(total_amt)
      FROM (SELECT r.name region_name,
SUM(o.total_amt_usd) total_amt
FROM sales_reps s
JOIN accounts a
ON a.sales_rep_id = s.id
JOIN orders o
ON o.account_id = a.id
JOIN region r
ON r.id = s.region_id
GROUP BY r.name) sub);
```

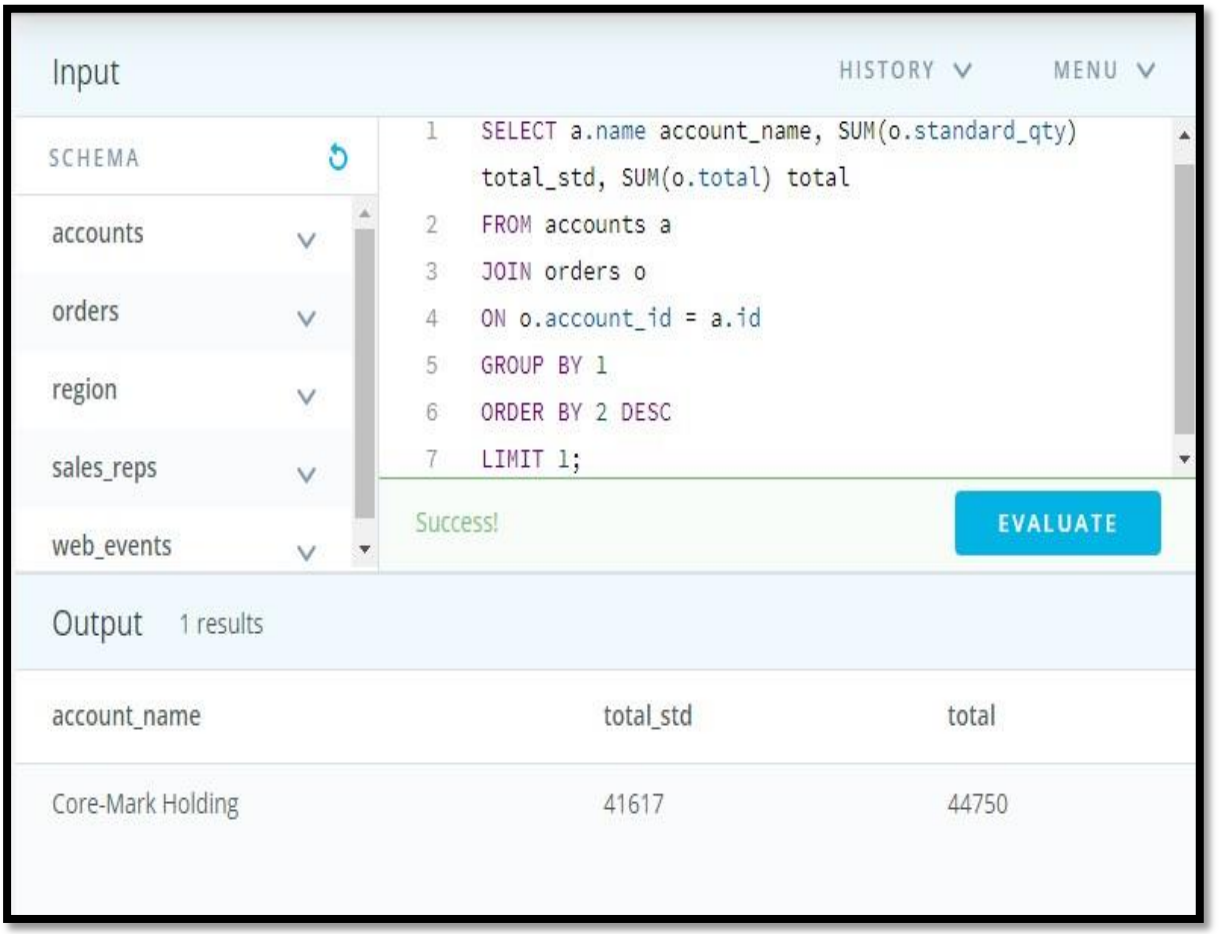
EVALUATE

Output 1 results

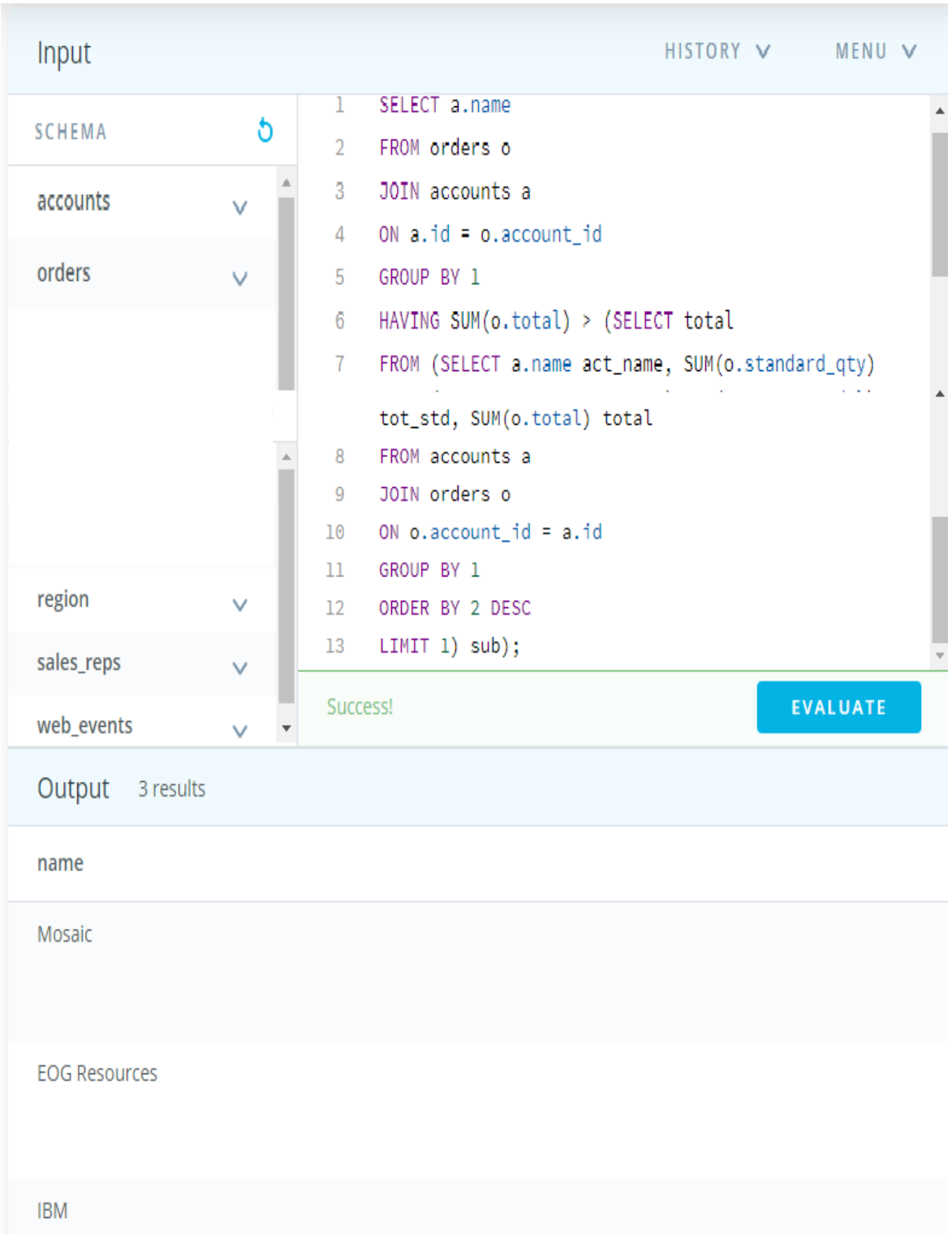
name	total_orders
Northeast	2357

3. For the **name** of the account that purchased the most (in total over their lifetime as a customer) **standard_qty**paper, **how many accounts** still had more in **total** purchases?

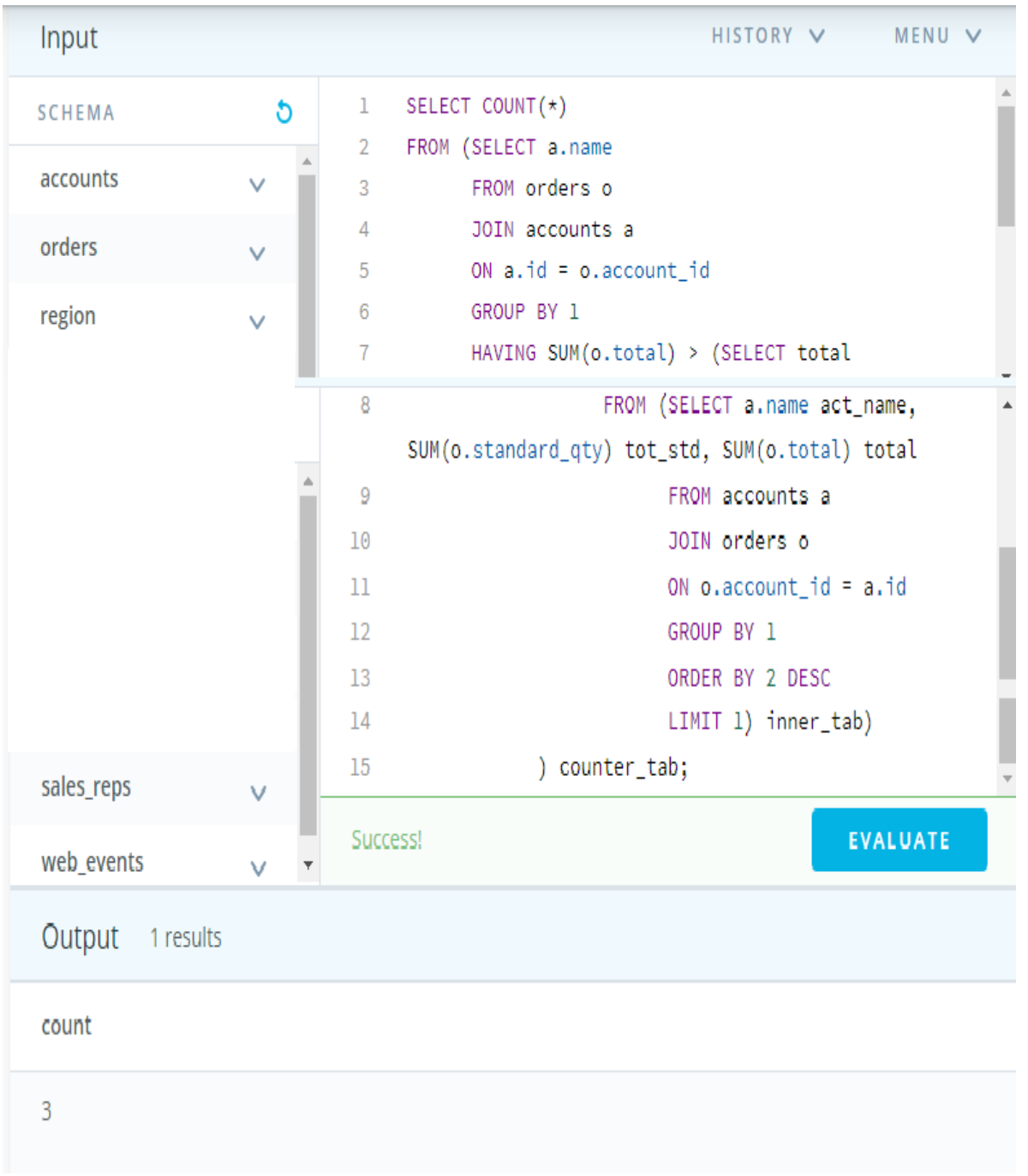
First, we want to find the account that had the most **standard_qty** paper. **The query here pulls that account, as well as the total amount:**



Now, I want to use this to pull all the accounts with more total sales:

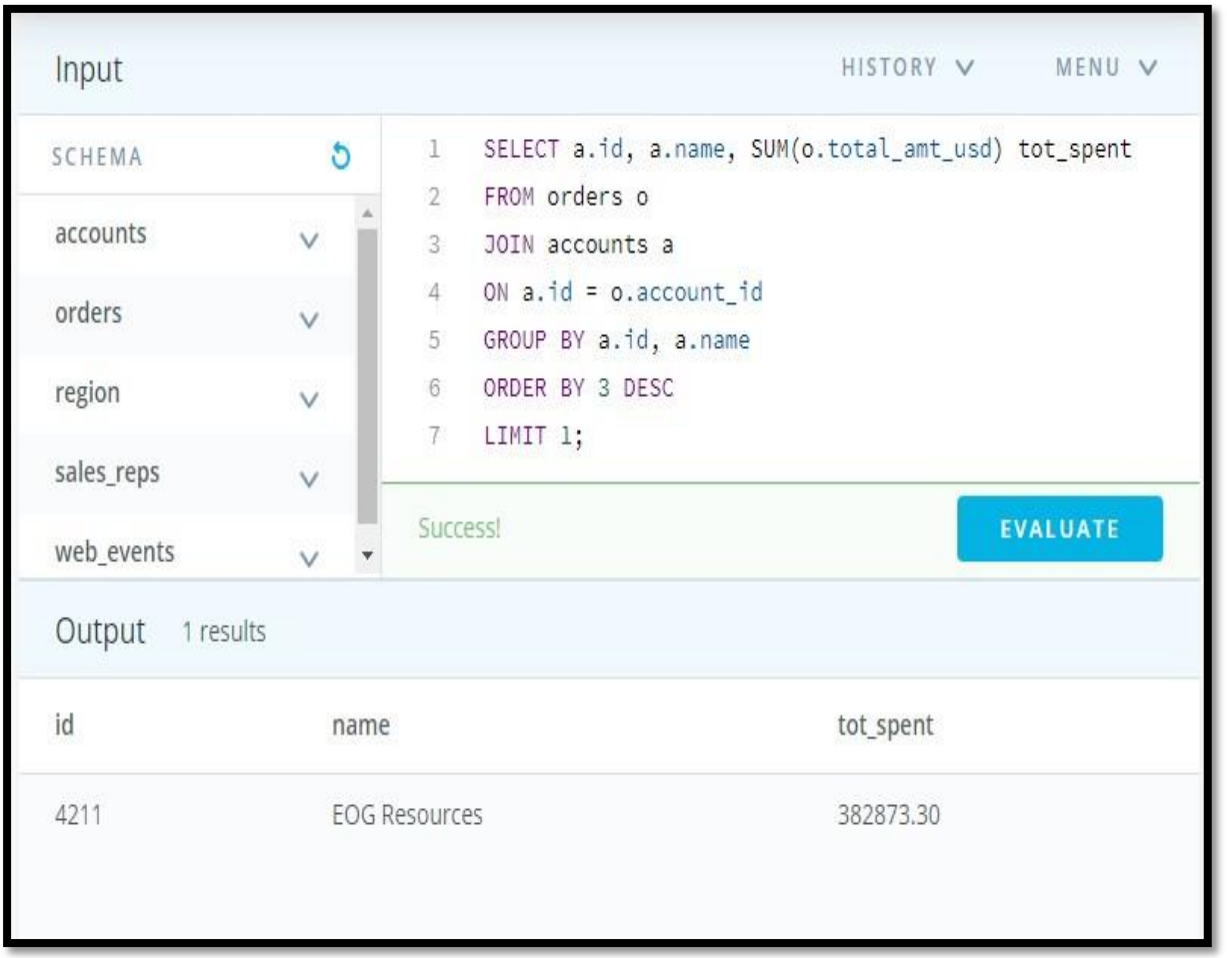


This is now a list of all the accounts with more total orders. We can get the count with just another simple subquery.

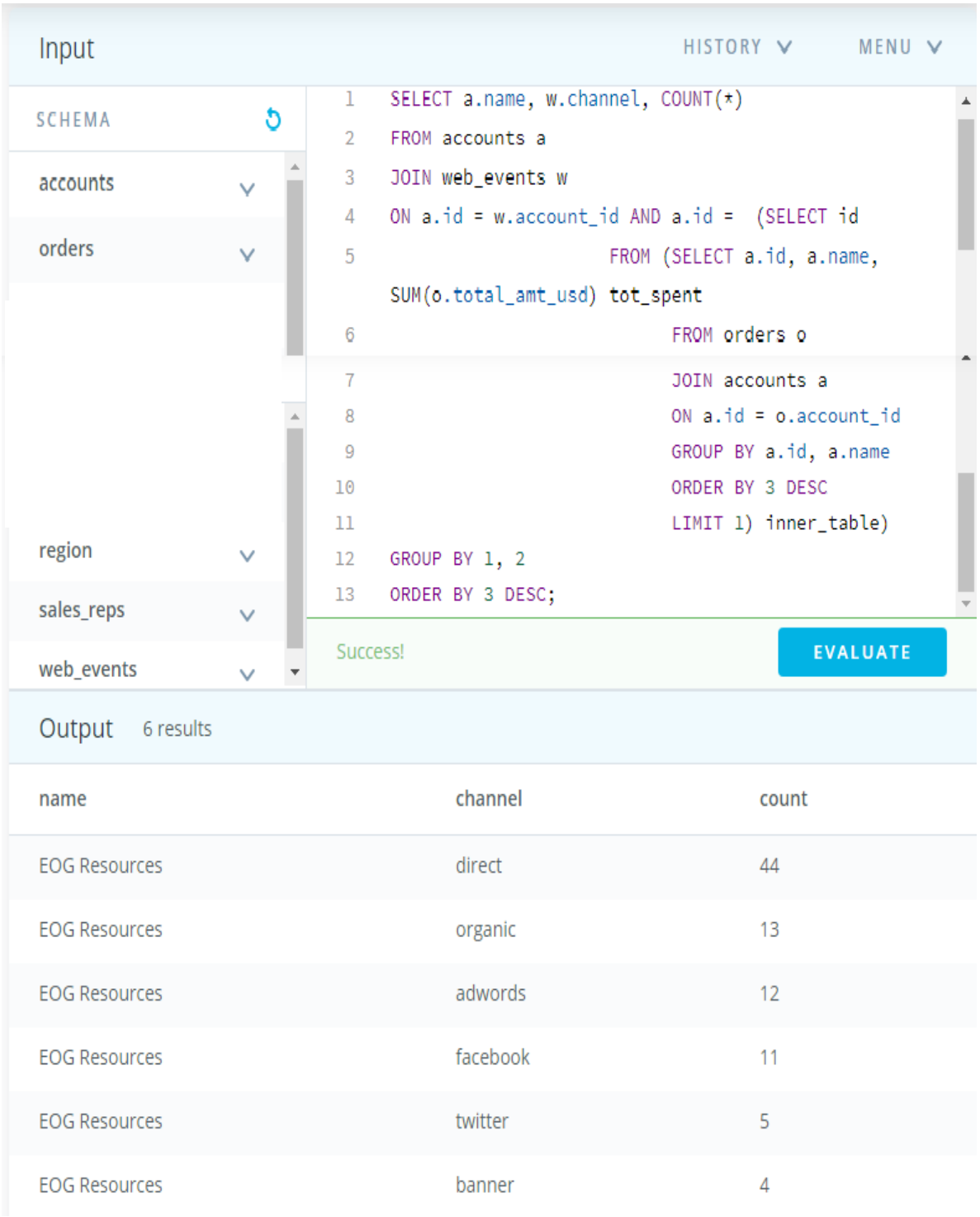

Input
HISTORY
MENU
SCHEMA
accounts
orders
region
sales_reps
web_events
1 SELECT COUNT(*)
2 FROM (SELECT a.name
3 FROM orders o
4 JOIN accounts a
5 ON a.id = o.account_id
6 GROUP BY 1
7 HAVING SUM(o.total) > (SELECT total
8 FROM (SELECT a.name act_name,
SUM(o.standard_qty) tot_std, SUM(o.total) total
9 FROM accounts a
10 JOIN orders o
11 ON o.account_id = a.id
12 GROUP BY 1
13 ORDER BY 2 DESC
14 LIMIT 1) inner_tab)
15) counter_tab;
Success!
EVALUATE
Output 1 results
count
3

4. For the customer that spent the most (in total over their lifetime as a customer) **total_amt_usd**, how many **web_events** did they have for each channel?

Here, we first want to pull the customer with the most spent in lifetime value.



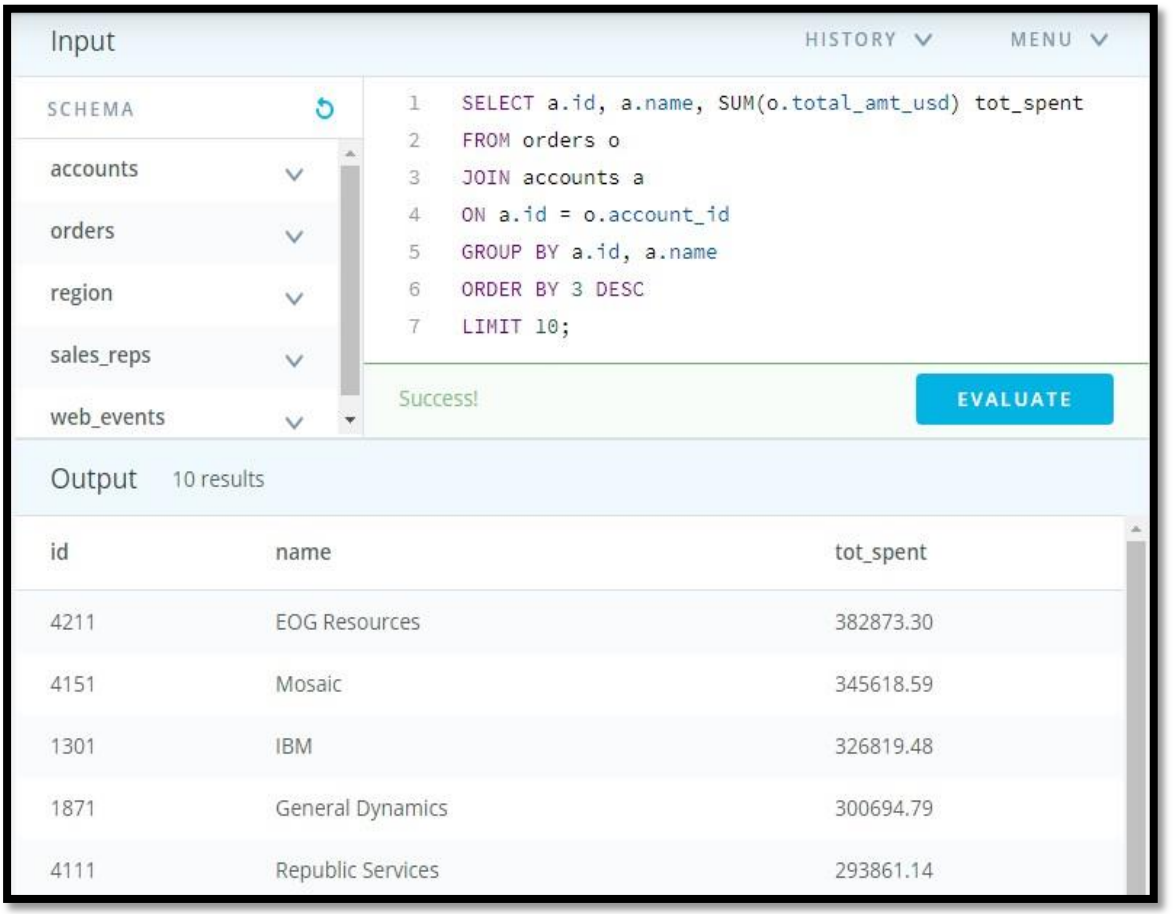
Now, we want to look at the number of events on each channel this company had, which we can match with just the **id**.



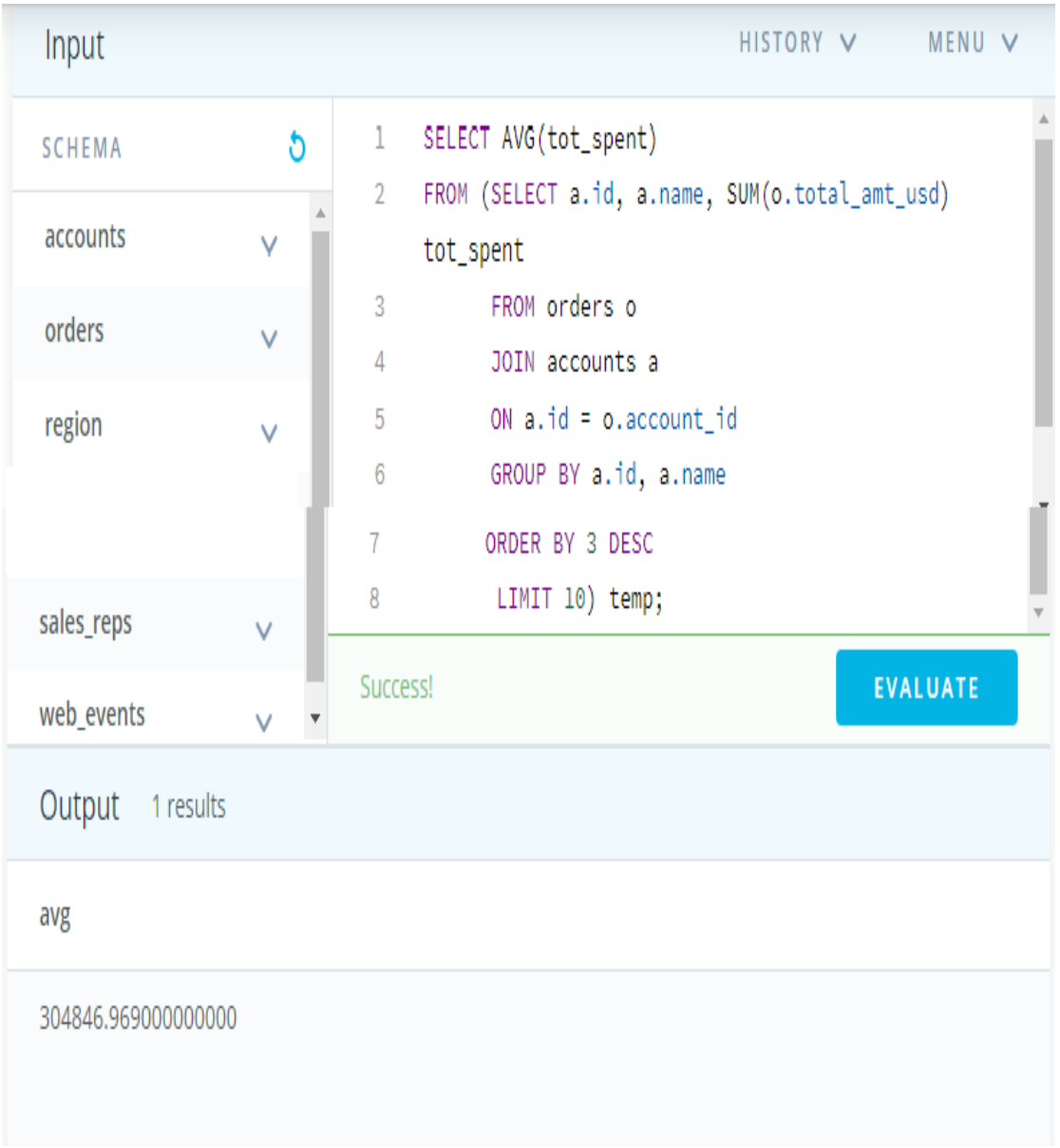
I added an **ORDER BY** for no real reason, and the account name to assure I was only pulling from one account.

5. What is the lifetime average amount spent in terms of **total_amt_usd** for the top 10 total spending **accounts**?

First, we just want to find the top 10 accounts in terms of highest **total_amt_usd**.

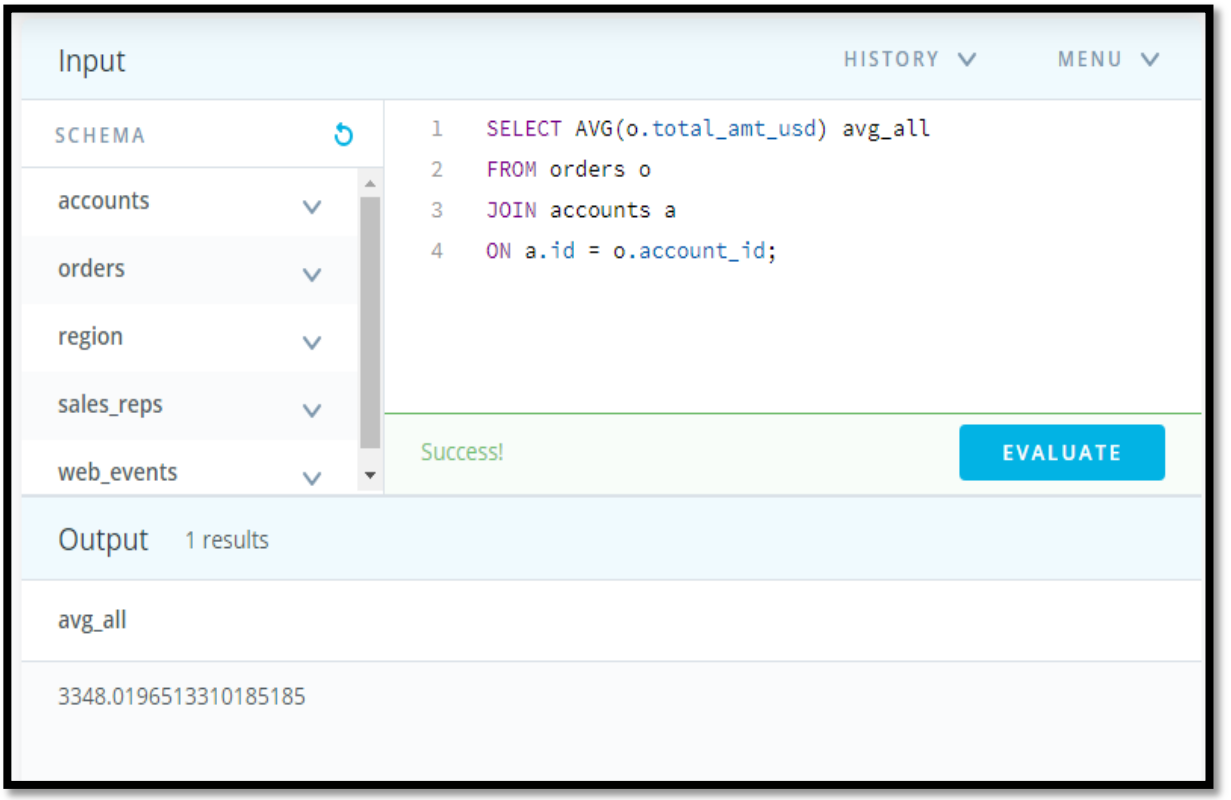


Now, we just want the average of these 10 amounts.

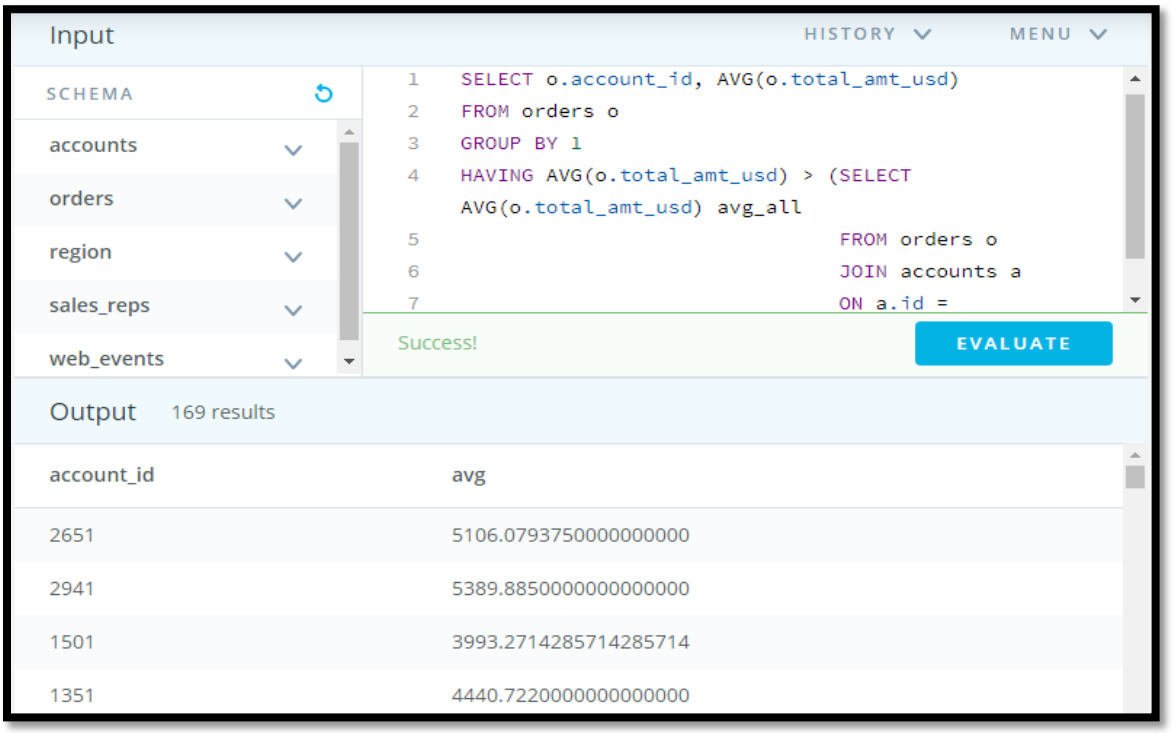

Input
HISTORY
MENU
SCHEMA
accounts
orders
region
sales_reps
web_events
1 SELECT AVG(tot_spent)
2 FROM (SELECT a.id, a.name, SUM(o.total_amt_usd)
tot_spent
3 FROM orders o
4 JOIN accounts a
5 ON a.id = o.account_id
6 GROUP BY a.id, a.name
7 ORDER BY 3 DESC
8 LIMIT 10) temp;
Success!
EVALUATE
Output 1 results
avg
304846.969000000000

6. What is the lifetime average amount spent in terms of **total_amt_usd** for only the companies that spent more than the average of all orders.

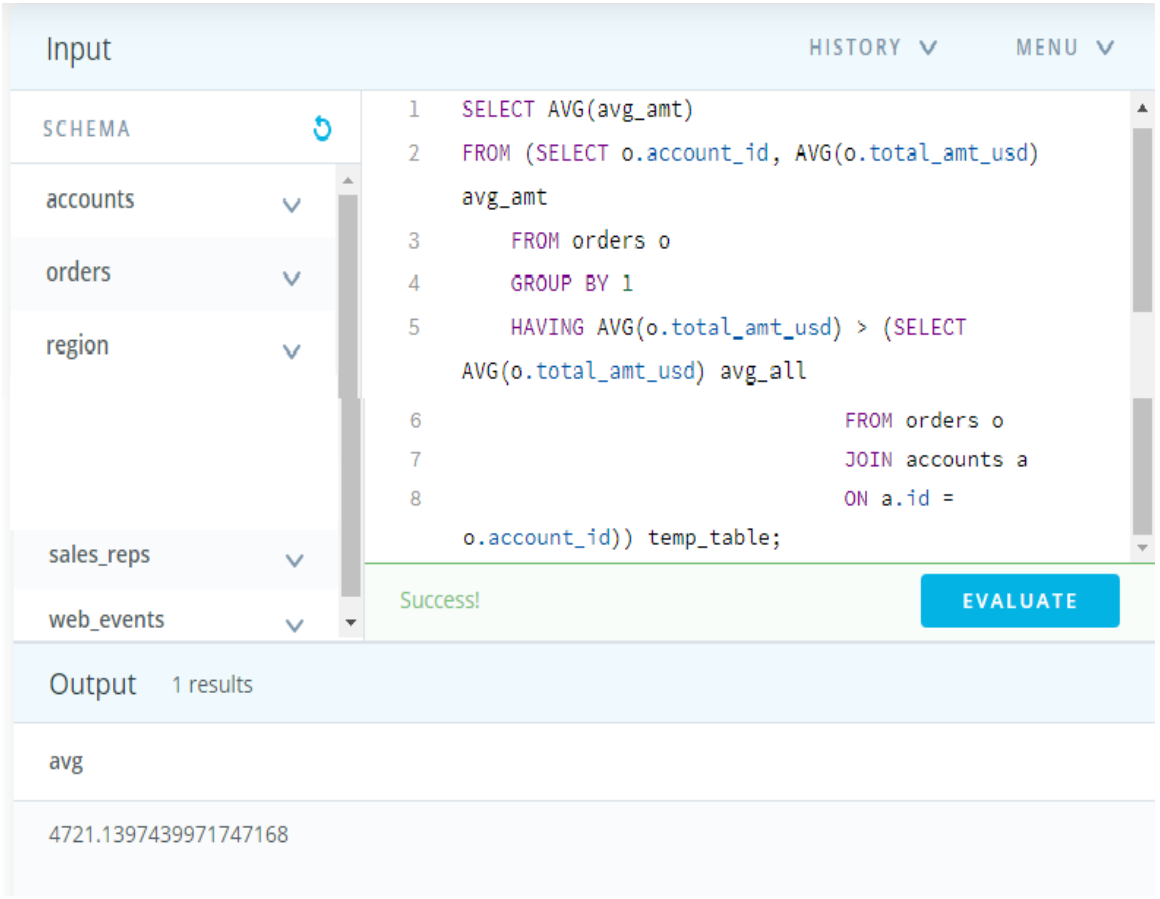
First, we want to pull the average of all accounts in terms of **total_amt_usd**:



Then, we want to only pull the accounts with more than this average amount.



Finally, we just want the average of these values.



WITH

The **WITH** statement is often called a **Common Table Expression** or **CTE**. Though these expressions serve the exact same purpose as subqueries, they are more common in practice, as they tend to be cleaner for a future reader to follow the logic.

In the next concept, we will walk through this example a bit more slowly to make sure you have all the similarities between subqueries and these expressions down for you to use in practice!

If you have a common table expression or subquery that takes a really long time to run, you might want to run it as completely separate query and that write it back into the database at it's own table. Then, you can simply query the new table as you would any other, to finish the thing you're trying to calculate.

The big benefit to this approach is that you can improve the speed at which you explore.

Let's say you've got a subquery that takes an hour to write, and then you've got an outer query that's pretty fast. If you run both of them every time you want to make a small tweak to the outer query, you're going to get really frustrated.

Instead, you can run that inner query once and write it to a table, then iterating on the outer query will be quick and easy. We'll dig into performance in more depth in later lesson.

For now, we'll focus on writing your subqueries and common table expressions into the database as new tables.

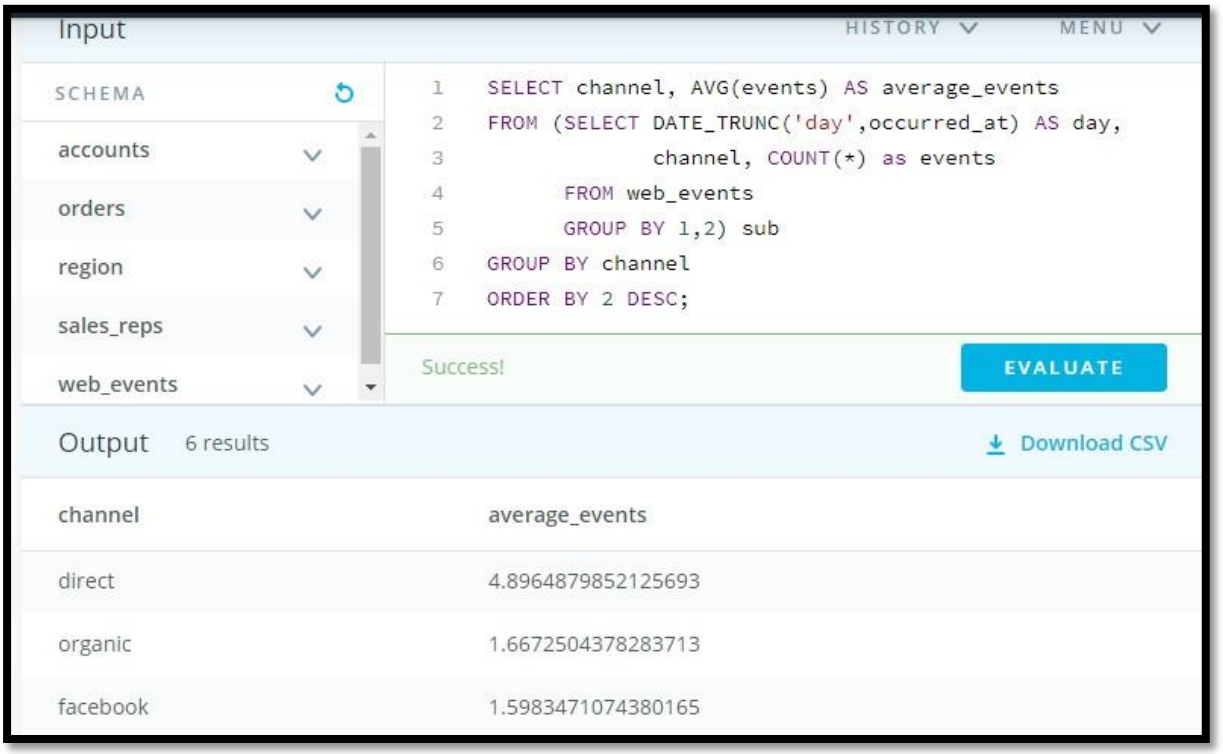
WITH vs. Subquery

Your First WITH (CTE)

The same question as you saw in your first subquery is provided here along with the solution.

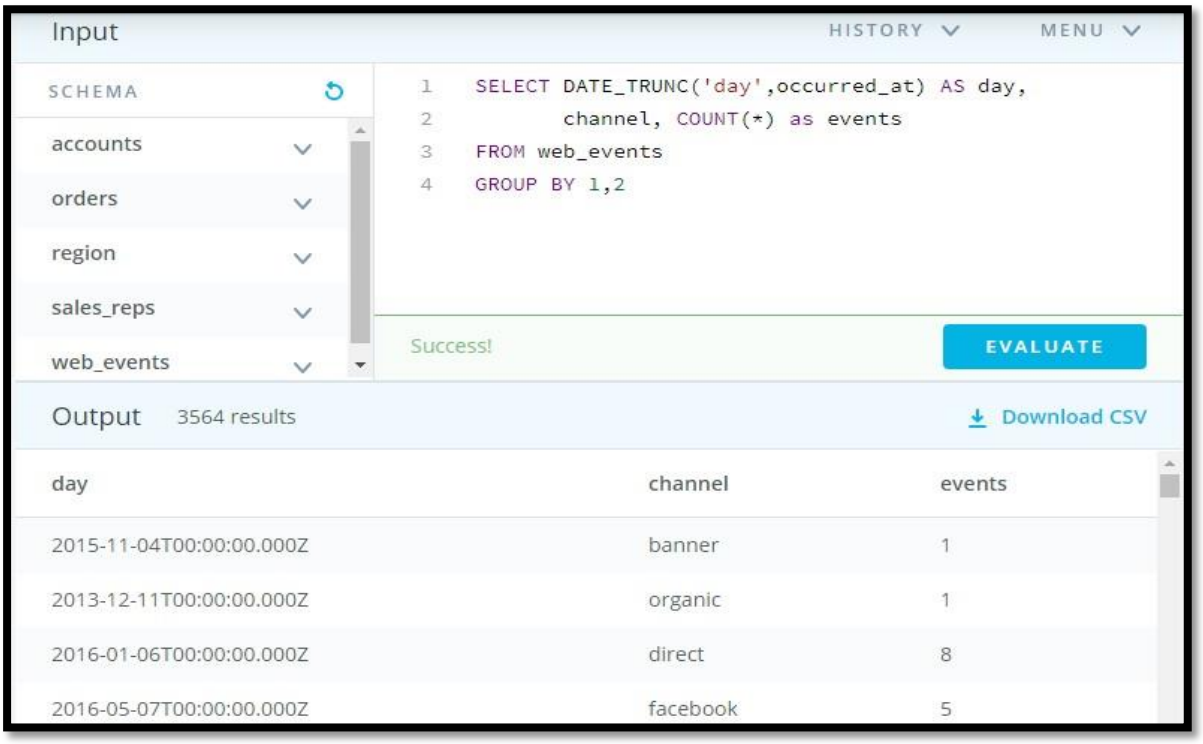
QUESTION: You need to find the average number of events for each channel per day.

SOLUTION:



Let's try this again using a **WITH** statement.

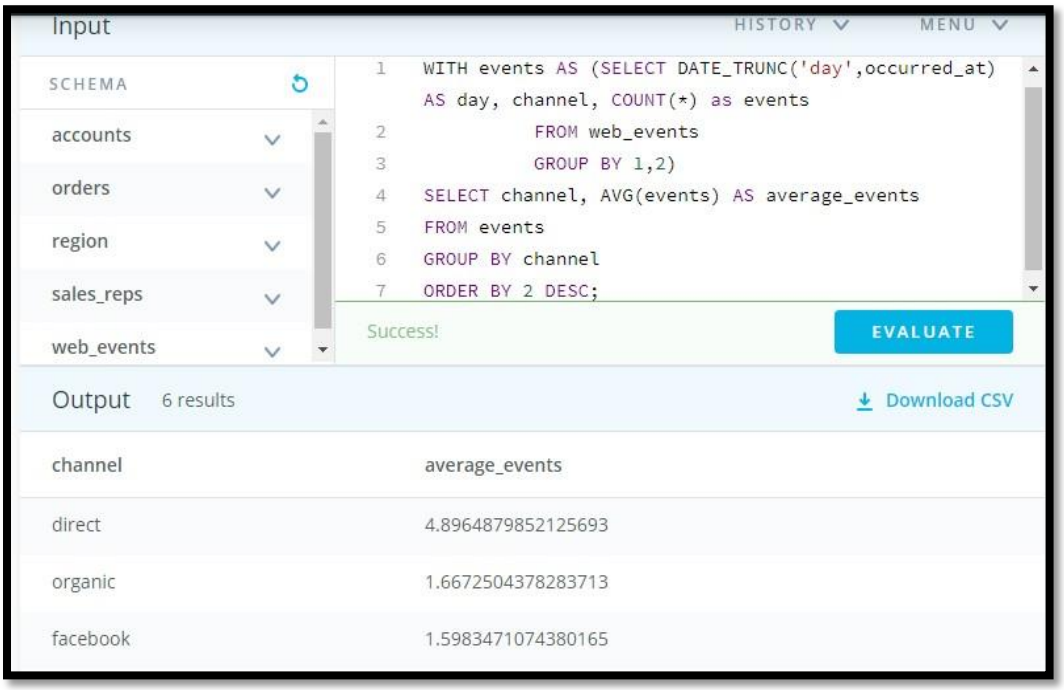
Notice, you can pull the inner query:



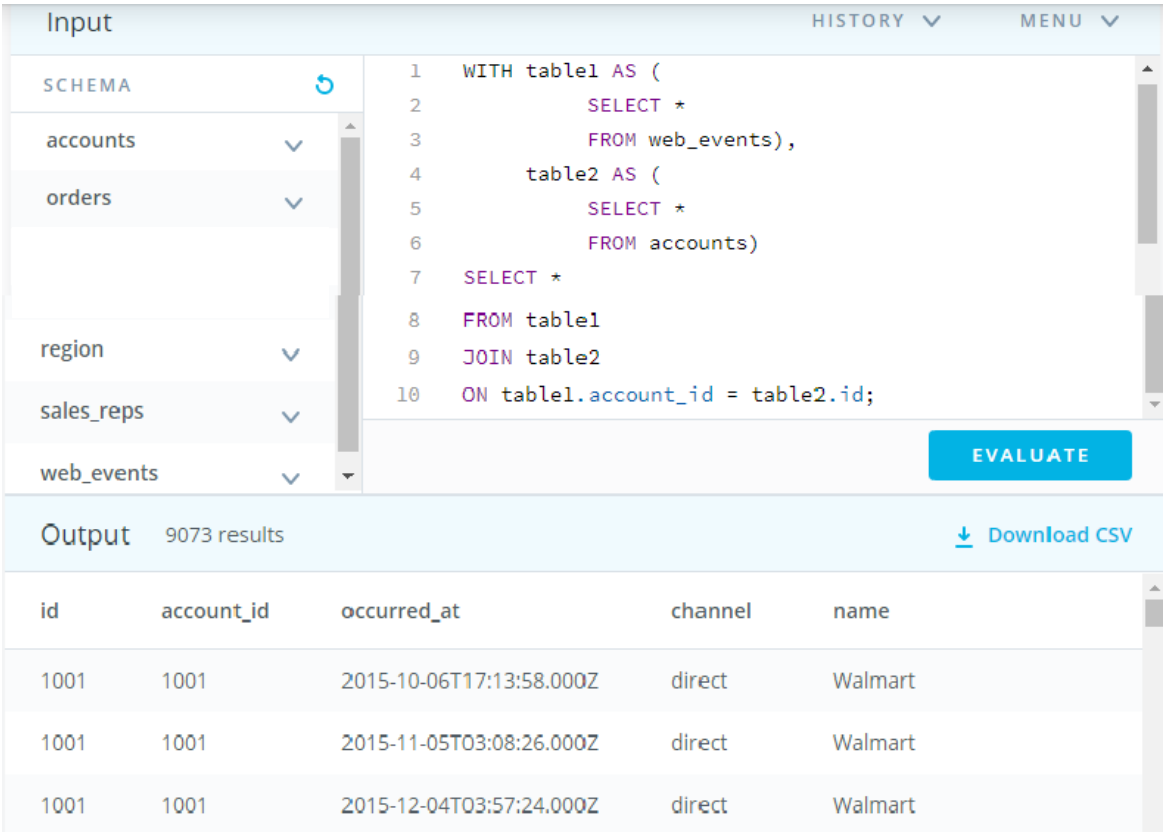
This is the part we put in the **WITH** statement. Notice, we are aliasing the table as events below:

```
WITH events AS (
        SELECT DATE_TRUNC('day',occurred_at) AS day,
                channel, COUNT(*) as events
        FROM web_events
        GROUP BY 1,2)
```

Now, we can use this newly created events table as if it is any other table in our database:



For the above example, we don't need anymore than the one additional table, but imagine we needed to create a second table to pull from. **We can create an additional table to pull from in the following way:**



You can add more and more tables using the **WITH** statement in the same way. The quiz at the bottom will assure you are catching all of the necessary components of these new queries.

Quiz: WITH vs. Subquery

QUIZ QUESTION

Select all of the below that are true regarding **WITH** statements.

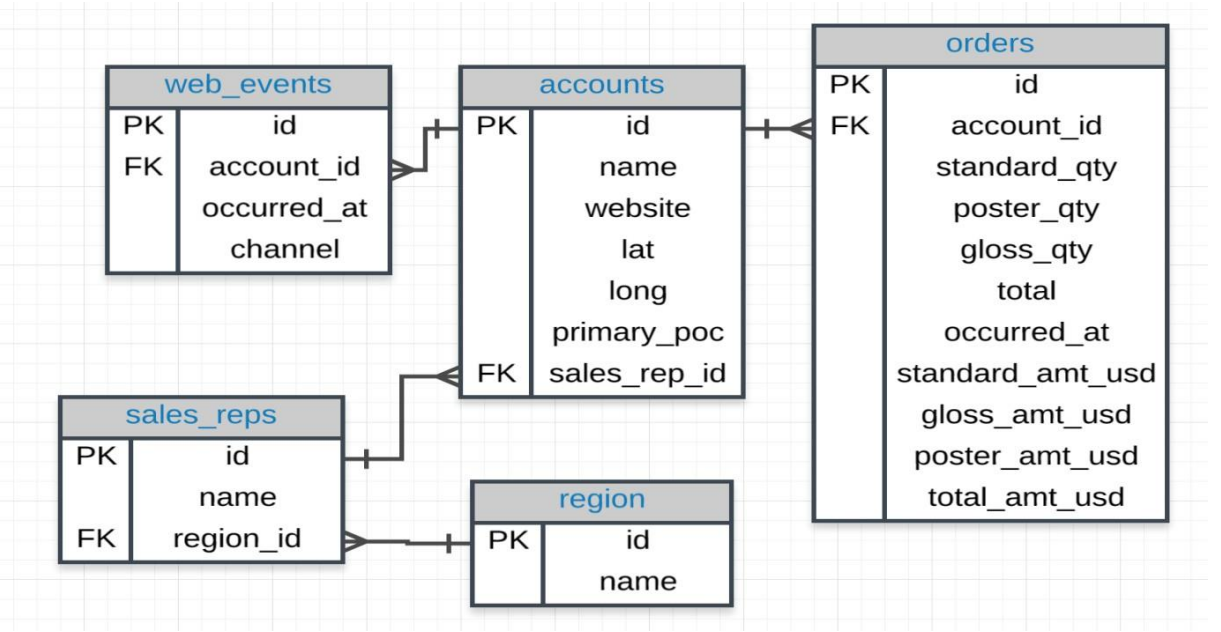
- [ ] When creating multiple tables using **WITH**, you add a comma after every table leading to your final query.

- [x] When creating multiple tables using **WITH**, you add a comma after every table except the last table leading to your final query.

- [x] The new table name is always aliased using table_name AS, which is followed by your query nested between parentheses.

- [ ] You begin each new table using a **WITH** statement.

Quiz: WITH



Essentially a **WITH** statement performs the same task as a **Subquery**. Therefore, you can write any of the queries we worked with in the "Subquery Mania" using a **WITH**. That's what you'll do here. Try to perform each of the earlier queries again, but using a **WITH** instead of a subquery.

Above is the ERD for the database again - it might come in handy as you tackle the quizzes below. You should write your solution as using a **WITH** statement, not by finding one solution and copying the output. The importance of this is that it allows your query to be dynamic in answering the question - even if the data changes, you still arrive at the right answer.

1. Provide the **name** of the **sales_rep** in each **region** with the largest amount of **total_amt_usd** sales.

Input HISTORY MENU

SCHEMA

accounts

orders

region

sales_reps

web_events

```
WITH t1 AS (
  SELECT s.name rep_name, r.name region_name,
SUM(o.total_amt_usd) total_amt
   FROM sales_reps s
   JOIN accounts a
   ON a.sales_rep_id = s.id
   JOIN orders o
   ON o.account_id = a.id
   JOIN region r
   ON r.id = s.region_id
   GROUP BY 1,2
   ORDER BY 3 DESC),
t2 AS (
   SELECT region_name, MAX(total_amt) total_amt
   FROM t1
   GROUP BY 1)
SELECT t1.rep_name, t1.region_name, t1.total_amt
FROM t1
JOIN t2
ON t1.region_name = t2.region_name AND t1.total_amt =
t2.total_amt;
```

Success! EVALUATE

Output 4 results Download CSV

rep_name	region_name	total_amt
Earlie Schleusner	Southeast	1098137.72
Tia Amato	Northeast	1010690.60
Georgianna Chisholm	West	886244.12
Charles Bidwell	Midwest	675637.19

2. For the region with the largest sales **total_amt_usd**, how many **total** orders were placed?

Input HISTORY MENU

SCHEMA

accounts

orders

region

sales_reps

web_events

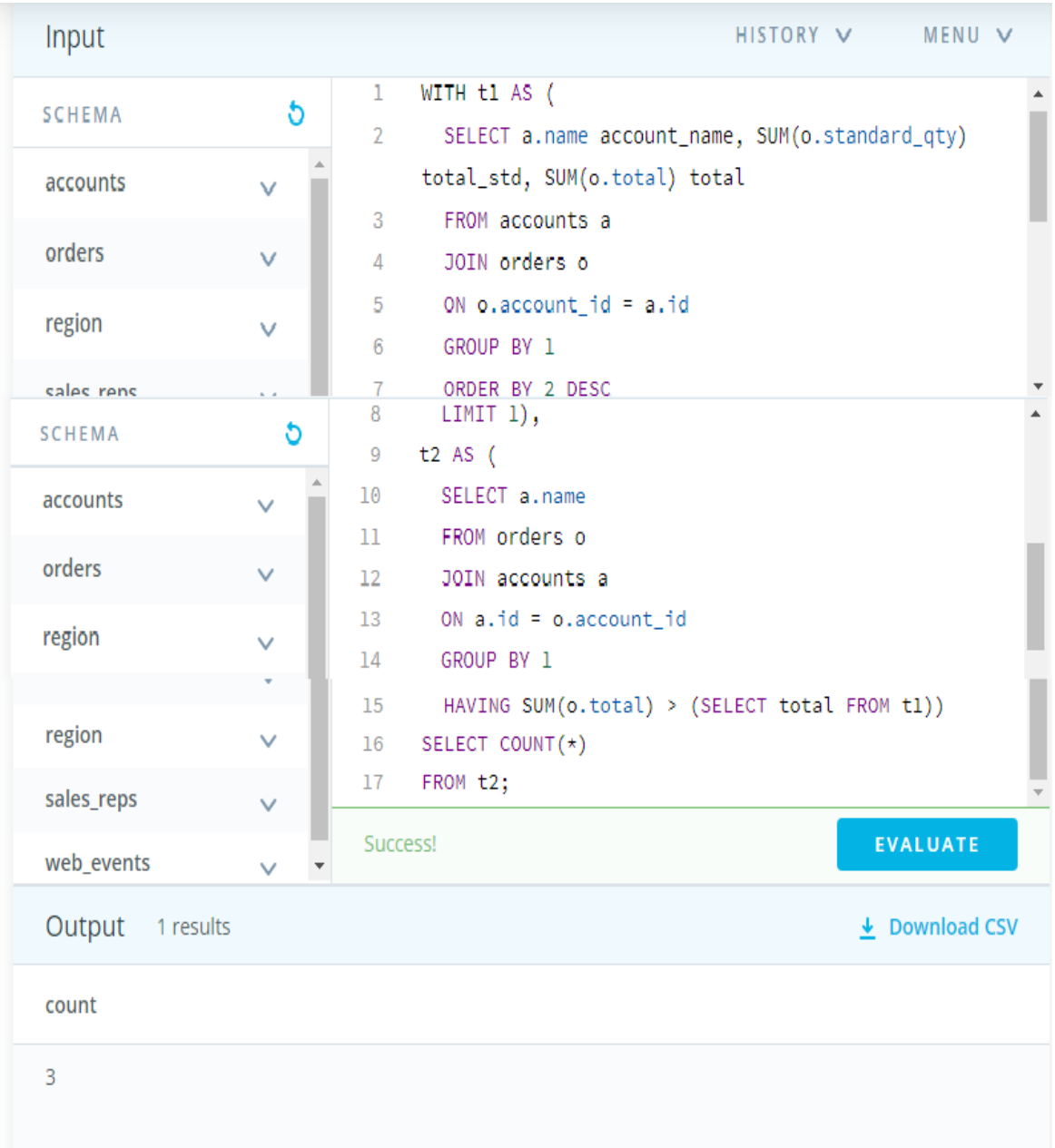
```
WITH t1 AS (
   SELECT r.name region_name, SUM(o.total_amt_usd)
total_amt
   FROM sales_reps s
   JOIN accounts a
   ON a.sales_rep_id = s.id
   JOIN orders o
   ON o.account_id = a.id
   JOIN region r
   ON r.id = s.region_id
   GROUP BY r.name),
t2 AS (
   SELECT MAX(total_amt)
   FROM t1)
SELECT r.name, SUM(o.total) total_orders
FROM sales_reps s
JOIN accounts a
ON a.sales_rep_id = s.id
JOIN orders o
ON o.account_id = a.id
JOIN region r
ON r.id = s.region_id
GROUP BY r.name
HAVING SUM(o.total_amt_usd) = (SELECT * FROM t2);
```

Success! EVALUATE

Output 1 results Download CSV

name	total_orders
Northeast	1230378

3. For the **name** of the account that purchased the most (in total over their lifetime as a customer) **standard_qty**paper, **how many accounts** still had more in **total** purchases?



4. For the customer that spent the most (in total over their lifetime as a customer) **total_amt_usd**, how many **web_events** did they have for each channel?

Input HISTORY MENU

SCHEMA

accounts
orders
region
sales_reps

```
1  WITH t1 AS (
2     SELECT a.id, a.name, SUM(o.total_amt_usd)
   tot_spent
3     FROM orders o
4     JOIN accounts a
5     ON a.id = o.account_id
6     GROUP BY a.id, a.name
7     ORDER BY 3 DESC
8     LIMIT 1)
9  SELECT a.name, w.channel, COUNT(*)
10 FROM accounts a
11 JOIN web_events w
12 ON a.id = w.account_id AND a.id =  (SELECT id FROM
   t1)
13 GROUP BY 1, 2
14 ORDER BY 3 DESC;
```

SCHEMA

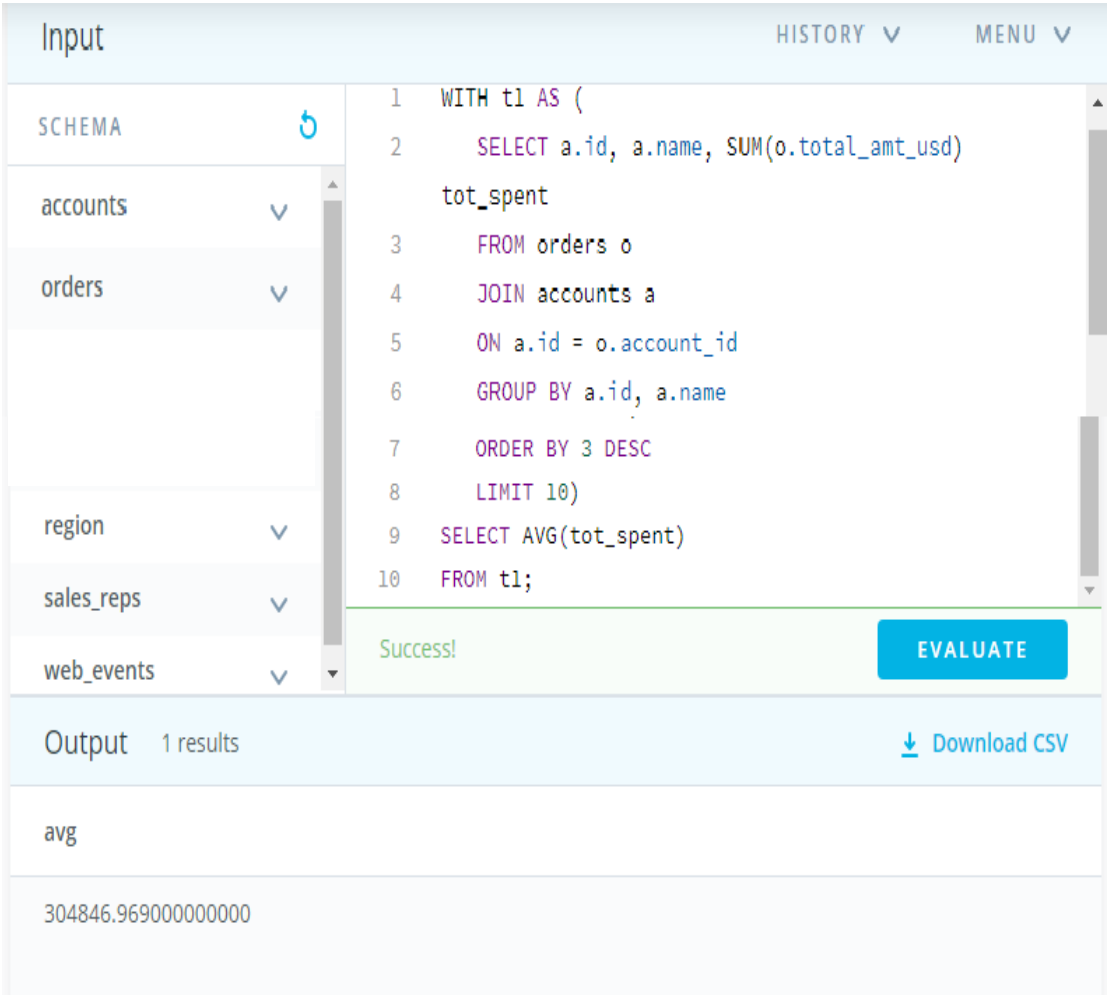
accounts
orders
region
sales_reps
web_events

Success! EVALUATE

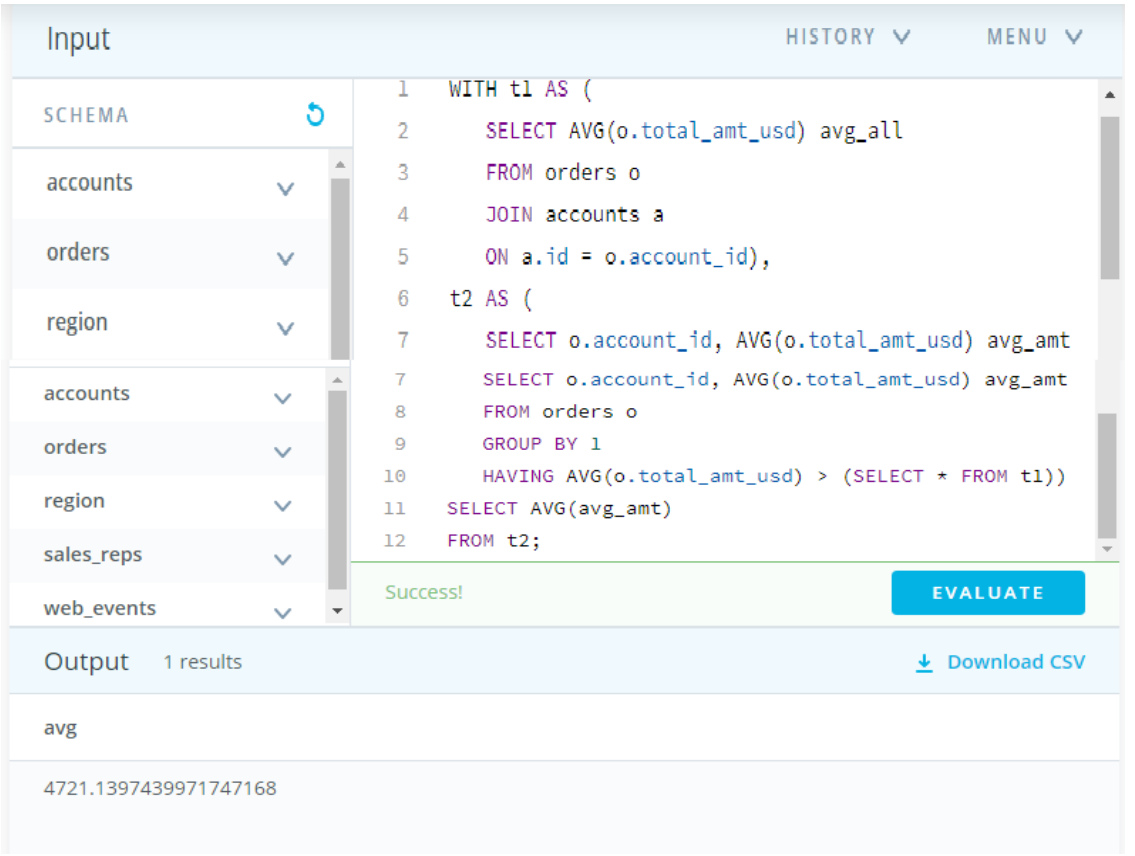
Output 6 results Download CSV

name	channel	count
EOG Resources	direct	44
EOG Resources	organic	13
EOG Resources	adwords	12
EOG Resources	facebook	11
EOG Resources	twitter	5
EOG Resources	banner	4

5. What is the lifetime average amount spent in terms of **total_amt_usd** for the top 10 total spending **accounts**?



6. What is the lifetime average amount spent in terms of **total_amt_usd** for only the companies that spent more than the average of all **accounts**.



QUIZ QUESTION

Look back through your solutions to subqueries and CTEs (WITH statements), which do you find more readable? Which is better for performance?

- [ ] CTEs are more readable.
- [x] CTEs are more efficient, as the tables aren't recreated with each subquery portion.

Note: Perhaps you find Subqueries more readable, but most find CTEs to be better for both of these options. They are certainly more efficient from a computing perspective.

Subquery Conclusion

In this lesson, you learned how to write subquueries and use temp tables. Writing queries like the ones you learned here, allows you to substantially increase the types of problems you can solve and the volume of data you can comfortably work with in SQL.

In the next lesson, you'll learn a few tools to handle more directed use cases.

Recap

This lesson was the first of the more advanced sequence in writing SQL. Arguably, the advanced features of **Subqueries** and **CTEs** are the most widely used in an analytics role within a company. Being able to break a problem down into the necessary tables and finding a solution using the resulting table is very useful in practice.

If you didn't get the solutions to these queries on the first pass, don't be afraid to come back another time and give them another try. Additionally, you might try coming up with some questions of your own to see if you can find the solution.

The remaining portions of this course may be key to certain analytics roles, but you have now covered all of the main SQL topics you are likely to use on a day to day basis.

LESSON 30

Introduction to SQL Data Cleaning

In this lesson, you will be learning a number of techniques to

1. Clean and re-structure messy data.
2. Convert columns to different data types.
3. Tricks for manipulating **NULL**s.

This will give you a robust toolkit to get from raw data to clean data that's useful for analysis.

It's often the case, The data isn't structured exactly as you'd like in a couple of lessons you worked with the web events table, which contains a refer URL that's well pretty messy . In this lesson, you'll learn a number of techniques for cleaning and reorganizing messy data, so that you can group it the way you want . You'll also learn how to change your columns data type and some tricks for how to manipulate nulls. Put all that together, and you'll have a robust toolset for getting from row data to something that's very useful for analysis.

LEFT & RIGHT

Here we looked at three new functions:

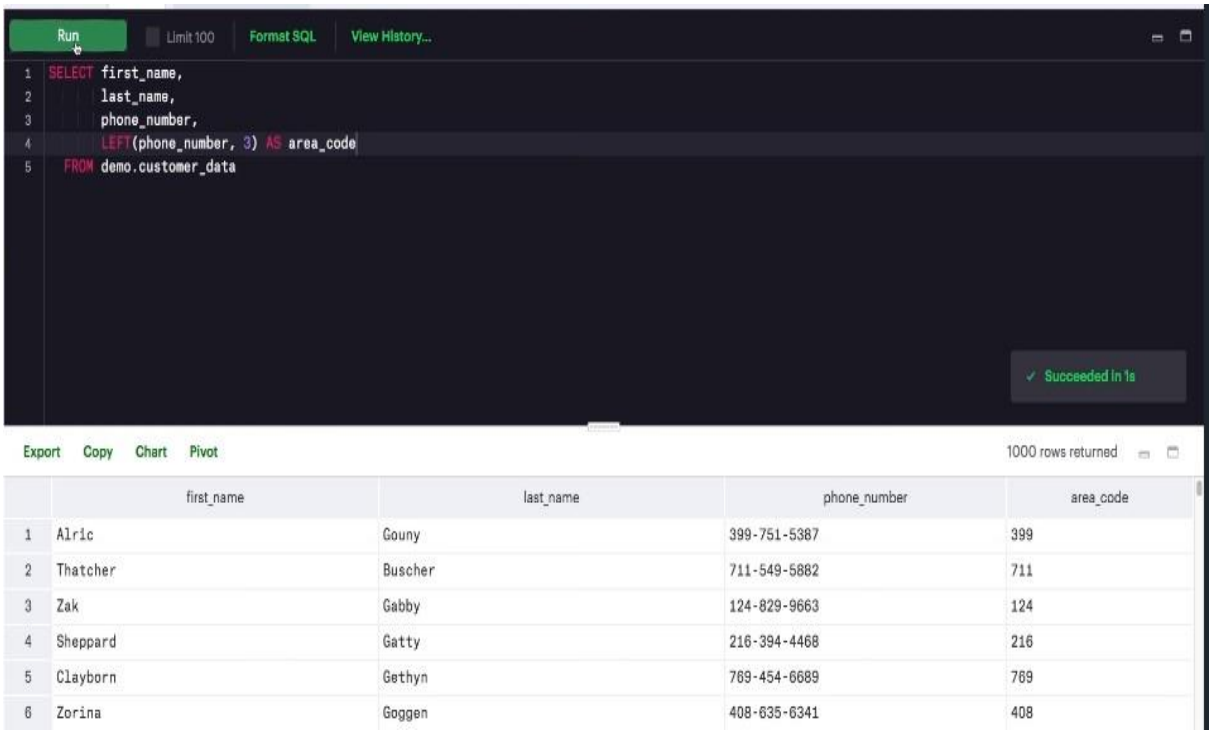
1. **LEFT**
2. **RIGHT**
3. **LENGTH**

LEFT pulls a specified number of characters for each row in a specified column starting at the beginning (or from the left). As you saw here, you can pull the first three digits of a phone number using **LEFT(phone_number, 3)**.

RIGHT pulls a specified number of characters for each row in a specified column starting at the end (or from the right). As you saw here, you can pull the last eight digits of a phone number using **RIGHT(phone_number, 8)**.

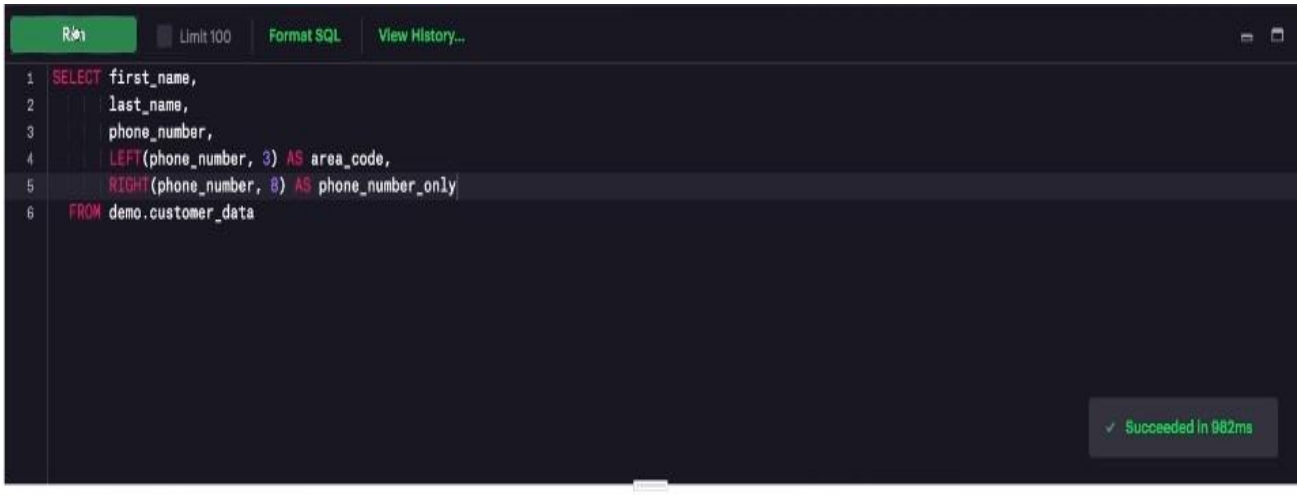
LENGTH provides the number of characters for each row of a specified column. Here, you saw that we could use this to get the length of each phone number as **LENGTH(phone_number)**.

We'll start by looking at this data set. It's a list of leads for Porch and Posy, Prospective customers whose business that they might try to win. Please keep in mind that this isn't anybody's real personal information. This was randomly generated for the sake of this lesson.
Our goal here is to clean up this data set, to make it more useful for analysis. For text fields, that means making clean groups that will be useful to aggregate across.
Let's start by pulling the area code out of the phone number. Since the structure of a phone number in this data set is always the same, we can use that to our advantage. The area code is always the first three characters of a phone number. We can get this using a left function. You can use left to pull certain number of characters from the left side of the string, and present them as a separate string.



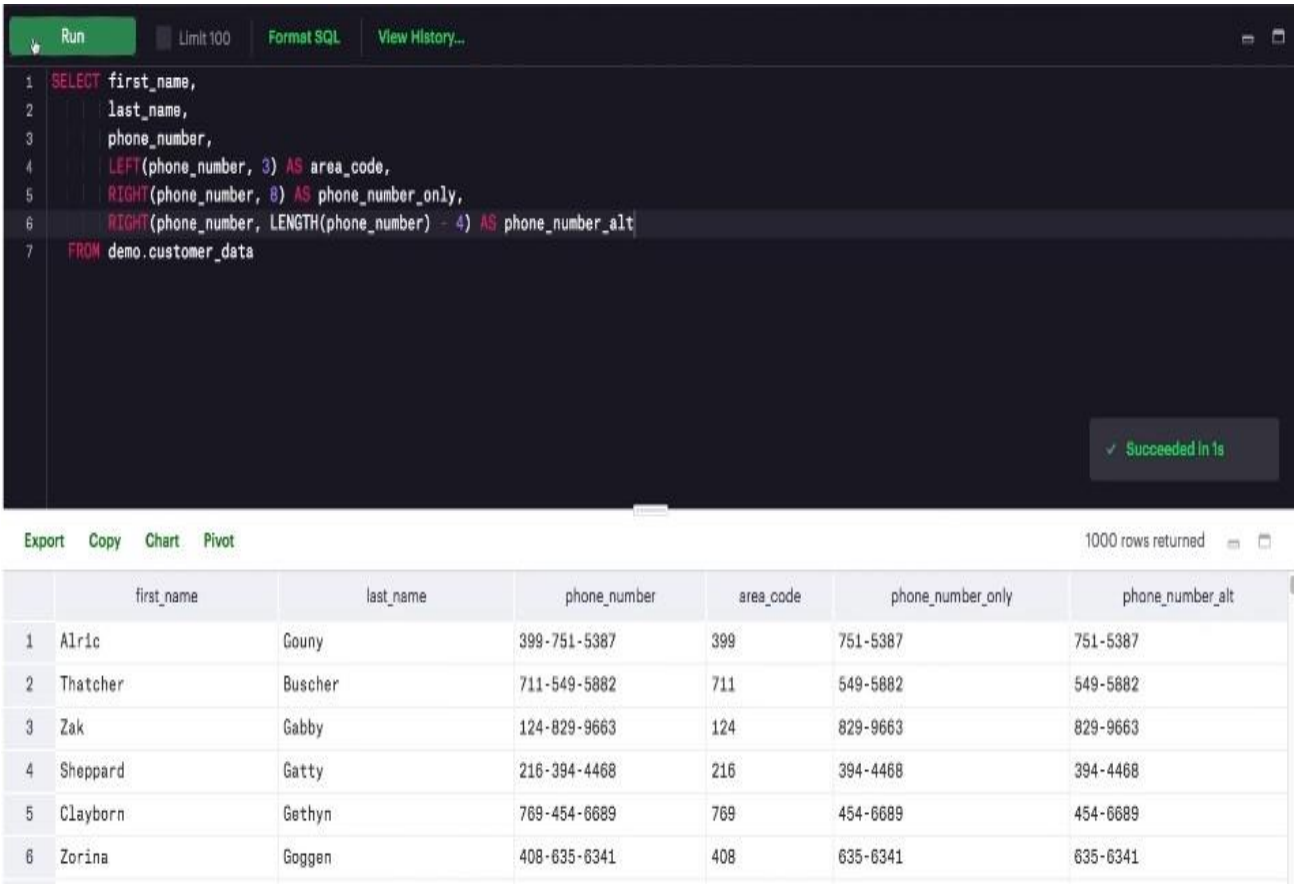
	first_name	last_name	phone_number	area_code
1	Alric	Gouny	399-751-5387	399
2	Thatcher	Buscher	711-549-5882	711
3	Zak	Gabby	124-829-9663	124
4	Sheppard	Gatty	216-394-4468	216
5	Clayborn	Gethyn	769-454-6689	769
6	Zorina	Goggen	408-635-6341	408

If we just want the phone number, we can use right, which dose the same thing but from the right side. Right works well in this case, because we know that the number of characters will be consistent across the entire phone number field.



	first_name	last_name	phone_number	area_code	phone_number_only
1	Alric	Gouny	399-751-5387	399	751-5387
2	Thatcher	Buscher	711-549-5882	711	549-5882
3	Zak	Gabby	124-829-9663	124	829-9663
4	Sheppard	Gatty	216-394-4468	216	394-4468
5	Clayborn	Gethyn	769-454-6689	769	454-6689
6	Zorina	Goggen	408-635-6341	408	635-6341

If it wasn't consistent, it's still possible to pull a string from the right side, in a way that makes sense. The length function returns the length of the string .So, the length of the phone number will always return 12 in this dataset. Since we know that the first three characters will be the area code, and they'll be followed by a dash. So, total of four characters. We could represent the right function as a function of the length.



	first_name	last_name	phone_number	area_code	phone_number_only	phone_number_alt
1	Alric	Gouny	399-751-5387	399	751-5387	751-5387
2	Thatcher	Buscher	711-549-5882	711	549-5882	549-5882
3	Zak	Gabby	124-829-9663	124	829-9663	829-9663
4	Sheppard	Gatty	216-394-4468	216	394-4468	394-4468
5	Clayborn	Gethyn	769-454-6689	769	454-6689	454-6689
6	Zorina	Goggen	408-635-6341	408	635-6341	635-6341

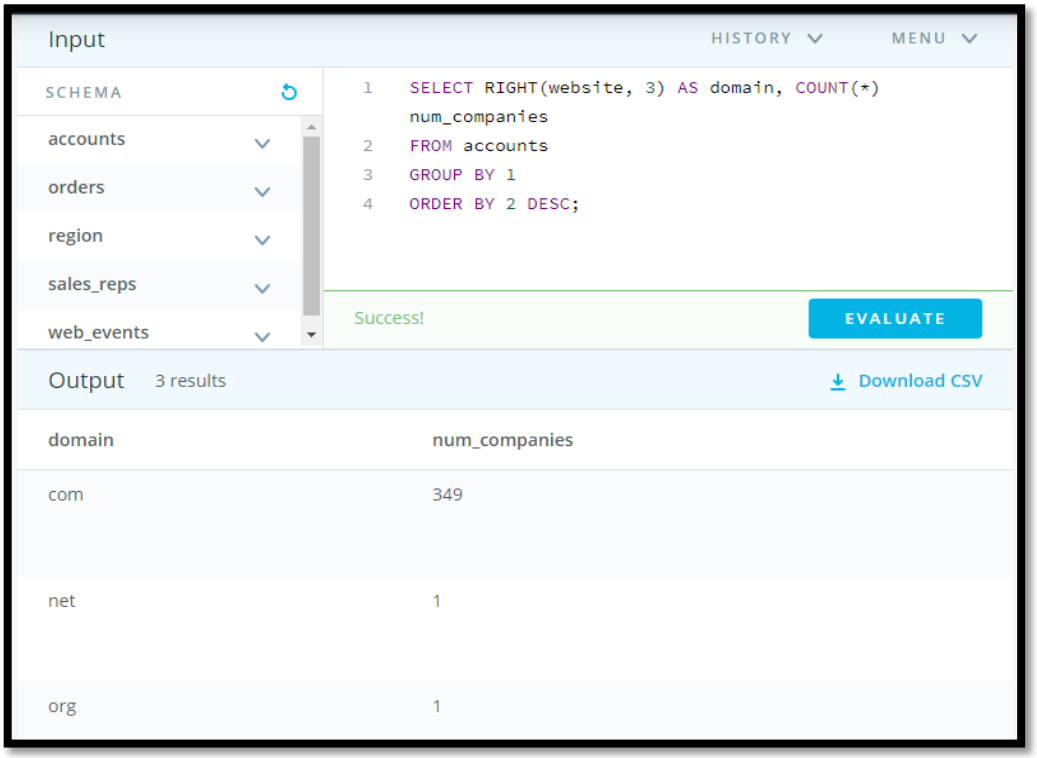
As you can see, the results in the phone number alt field are the same as the result in the phone number only field. This is because , length of the phone number minus four is going to resolve as twelve minus four or eight. Which is the same as the phone number only call.

When using functions within other functions, it's important to remember that the innermost functions will be evaluated first, followed by the functions that encapsulate

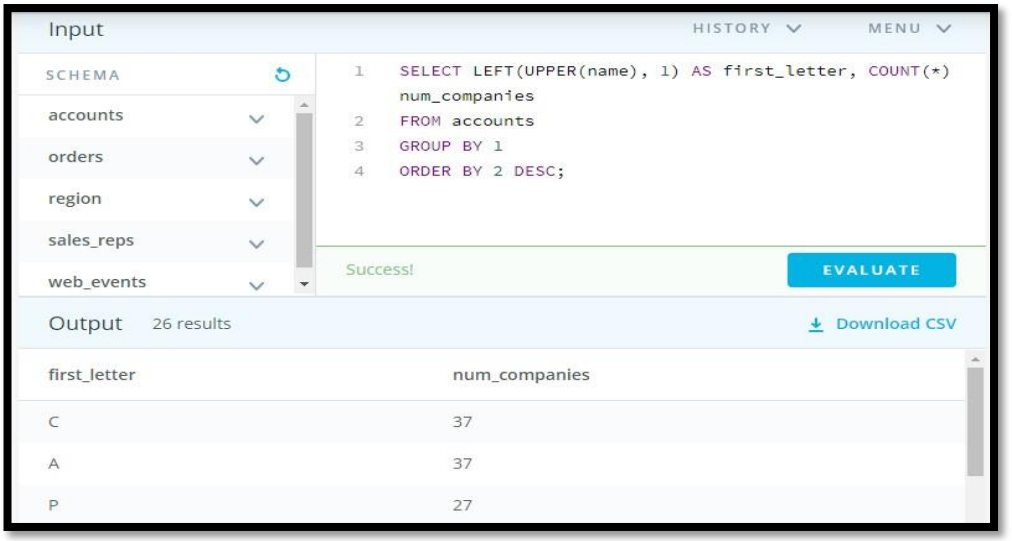
Quiz: LEFT & RIGHT

LEFT & RIGHT Quizzes

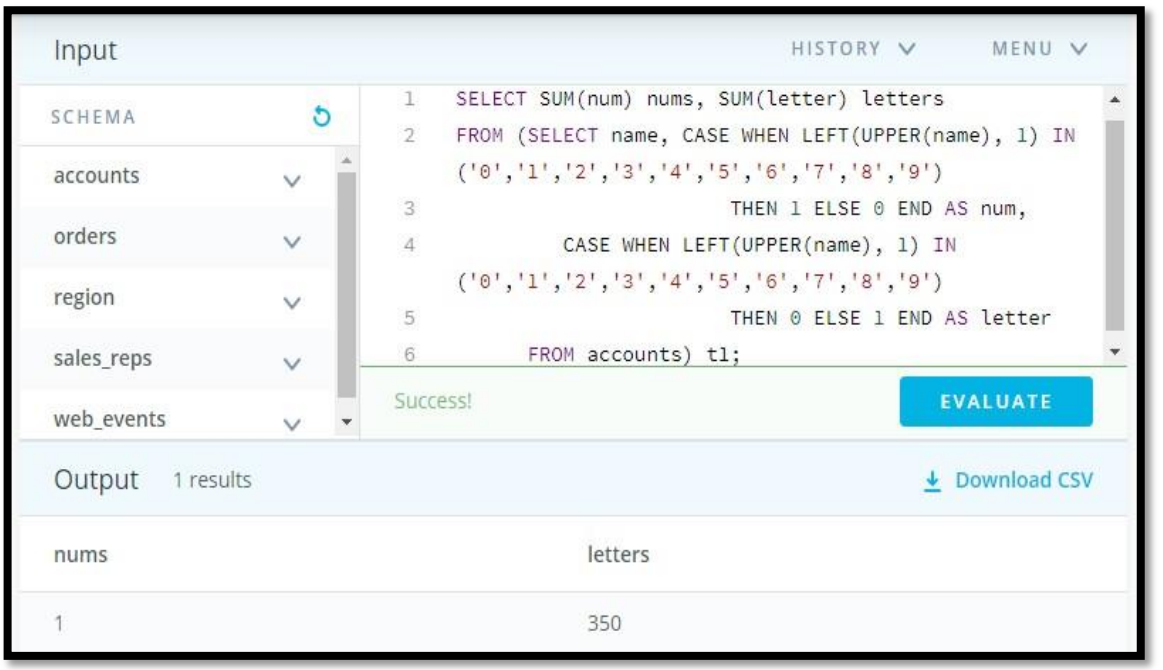
1. In the **accounts** table, there is a column holding the **website** for each company. The last three digits specify what type of web address they are using.



2. There is much debate about how much the name (or even the first letter of a company name) matters. Use the **accounts** table to pull the first letter of each company name to see the distribution of company names that begin with each letter (or number).

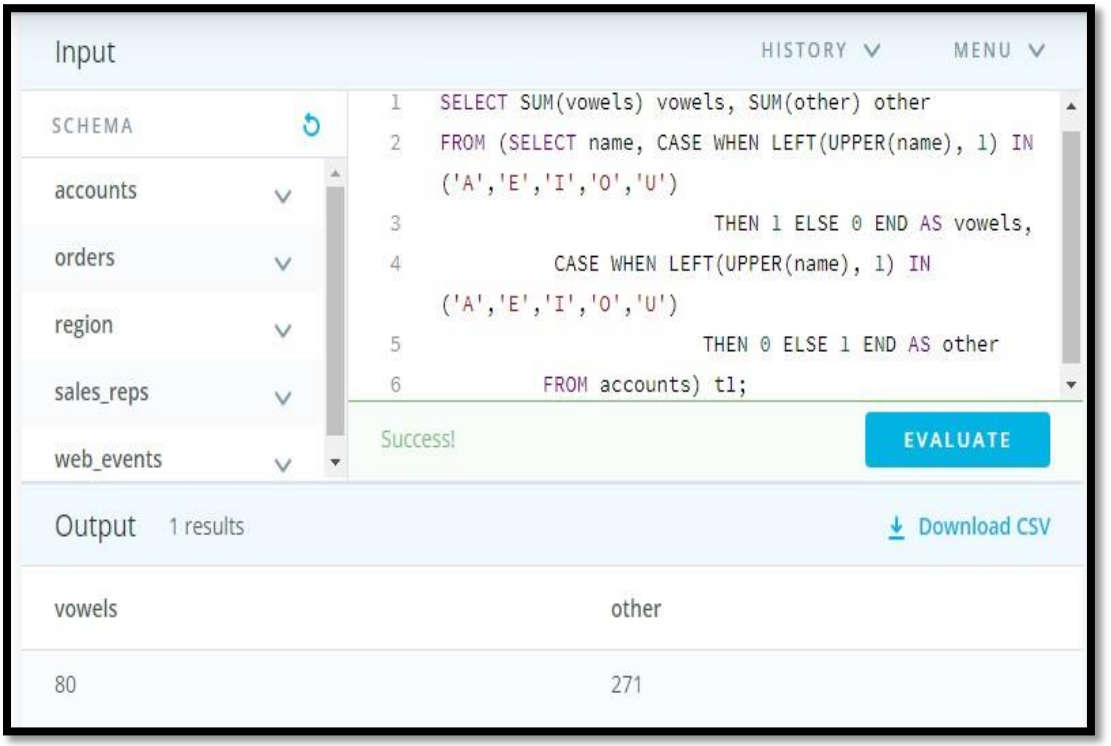


3. Use the **accounts** table and a **CASE** statement to create two groups: one group of company names that start with a number and a second group of those company names that start with a letter. What proportion of company names start with a letter?



There are 350 company names that start with a letter and 1 that starts with a number. This gives a ratio of 350/351 that are company names that start with a letter or 99.7%.

4. Consider vowels as a, e, i, o, and u. What proportion of company names start with a vowel, and what percent start with anything else?



There are 80 company names that start with a vowel and 271 that start with other characters. Therefore 80/351 are vowels or 22.8%. Therefore, 77.2% of company names do not start with vowels.

POSITION, STRPOS, & SUBSTR

In this lesson, you learned about:

1. **POSITION**
2. **STRPOS**
3. **LOWER**
4. **UPPER**

POSITION takes a character and a column, and provides the index where that character is for each row. The index of the first position is 1 in SQL. If you come from another programming language, many begin indexing at 0. Here, you saw that you can pull the index of a comma as **POSITION(',' IN city_state)**.

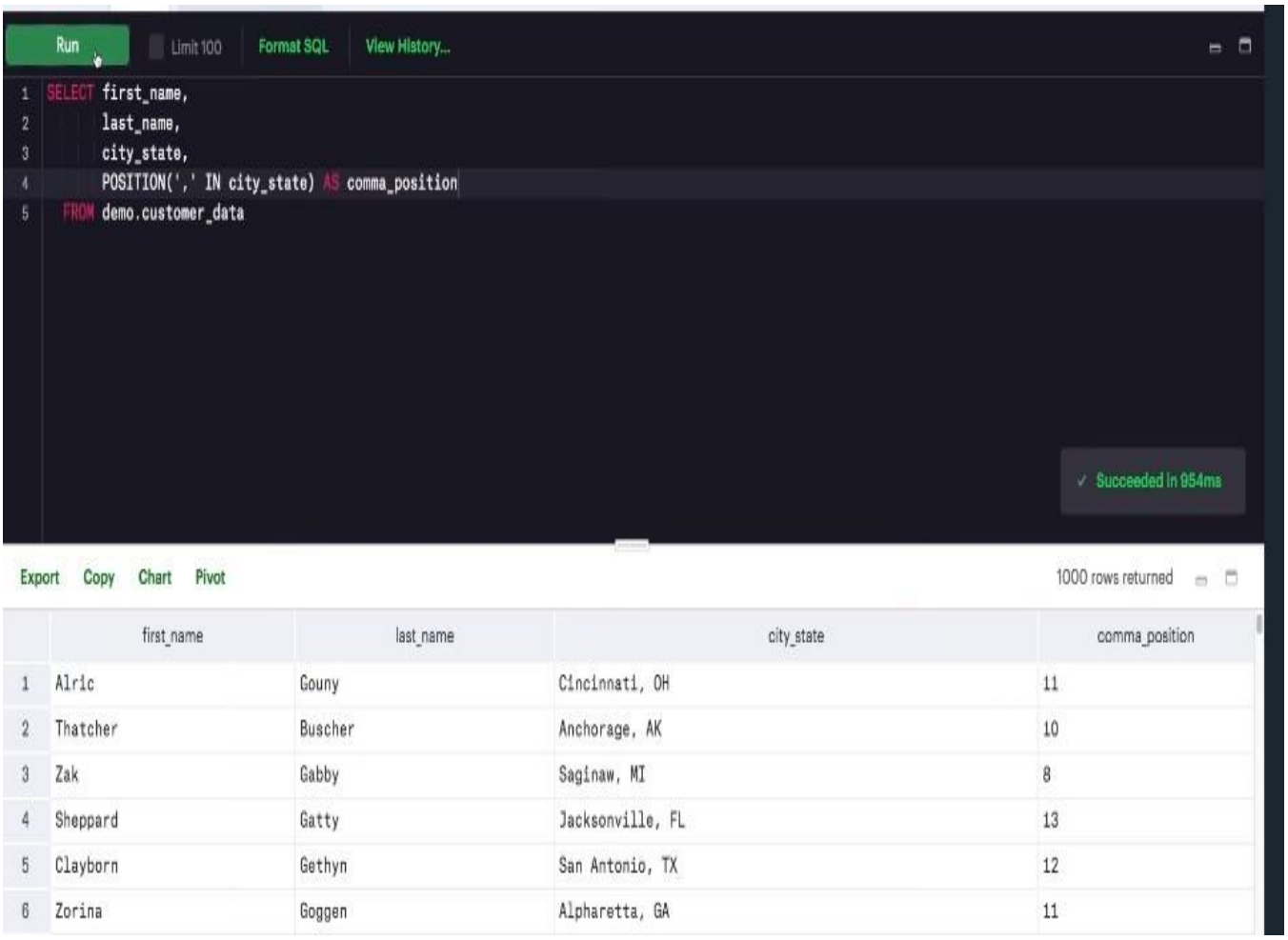
STRPOS provides the same result as **POSITION**, but the syntax for achieving those results is a bit different as shown here: **STRPOS(city_state, ',')**.

Note, both **POSITION** and **STRPOS** are case sensitive, so looking for **A** is different than looking for **a**.

Therefore, if you want to pull an index regardless of the case of a letter, you might want to use **LOWER** or **UPPER** to make all of the characters lower or uppercase.

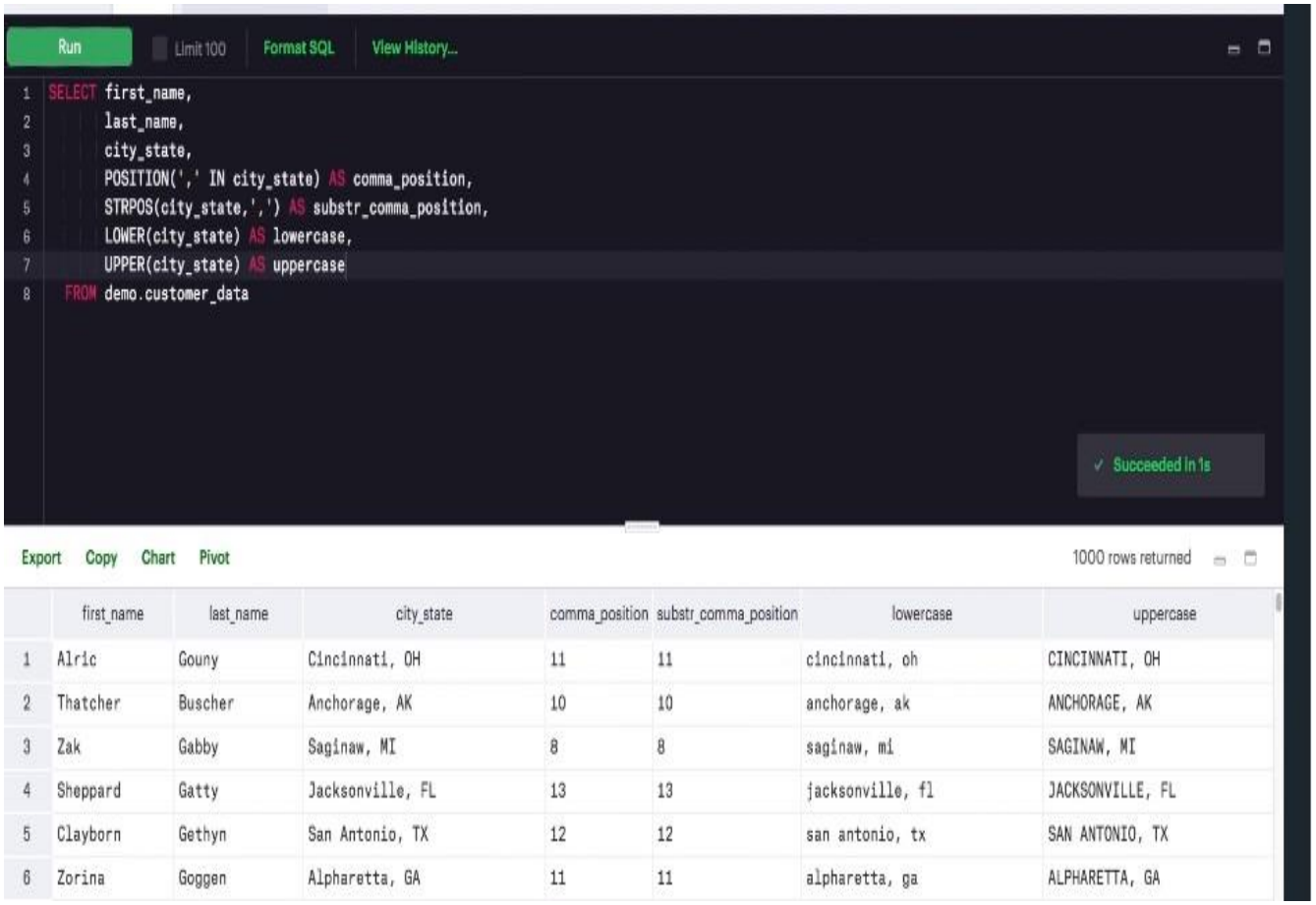
Left and right work pretty well in specific circumstances. When the date is structured very cleanly with a certain number of characters. If you want a separate city and state, you've got to do a little more work. The first thing that will be helpful is figuring out exactly where the city and state split. Since it will be different for each row, we have to use a function that will find the comma and identify how far into the record it is.
Position allows you to specify a sub-string, then it returns a numerical value equal to how far away from the left that particular character appears.

You can also use the string position function which is annotated as STRPOS to achieve the same results. just replace in with a comma and switch the order of the string and the sub-string.



	first_name	last_name	city_state	comma_position
1	Alric	Gouny	Cincinnati, OH	11
2	Thatcher	Buscher	Anchorage, AK	10
3	Zak	Gabby	Saginaw, MI	8
4	Sheppard	Gatty	Jacksonville, FL	13
5	Clayborn	Gethyn	San Antonio, TX	12
6	Zorina	Goggen	Alpharetta, GA	11

Importantly, both the position and string position functions are case sensitive. If you want to look for a character regardless of its case, you can make the entire string upper or lower case.
You can use lower to force every character in a string to become lowercase. similarly, you can use UPPER to make all the letters appear as uppercase. Let's complete the loop and pull just the city out into its own field. We can do this by nesting the position inside a Left function.



```
Run | Limit 100 | Format SQL | View History...

SELECT first_name,
       last_name,
       city_state,
       POSITION(',' IN city_state) AS comma_position,
       STRPOS(city_state,',') AS substr_comma_position,
       LOWER(city_state) AS lowercase,
       UPPER(city_state) AS uppercase
  FROM demo.customer_data
```

✓ Succeeded in 1s

Export Copy Chart Pivot — 1000 rows returned

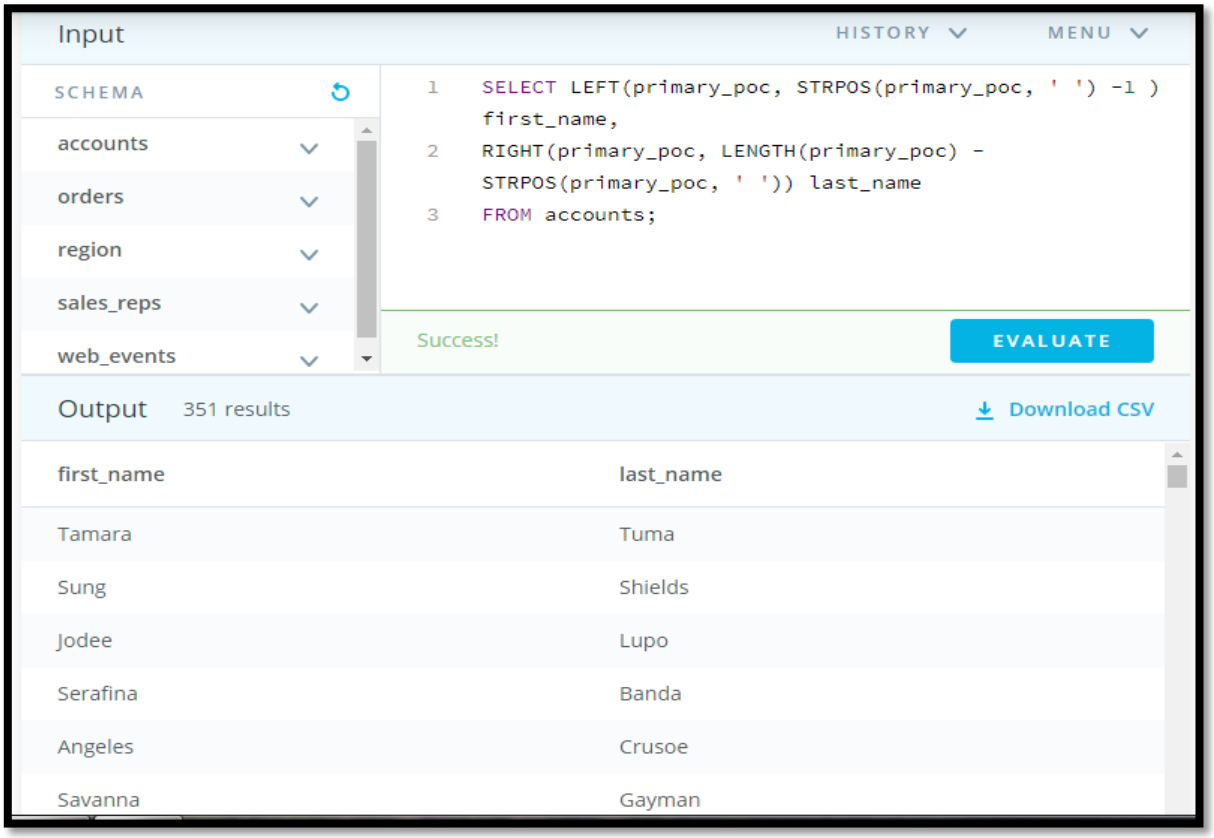
	first_name	last_name	city_state	comma_position	substr_comma_position	lowercase	uppercase
1	Alric	Gouny	Cincinnati, OH	11	11	cincinnati, oh	CINCINNATI, OH
2	Thatcher	Buscher	Anchorage, AK	10	10	anchorage, ak	ANCHORAGE, AK
3	Zak	Gabby	Saginaw, MI	8	8	saginaw, mi	SAGINAW, MI
4	Sheppard	Gatty	Jacksonville, FL	13	13	jacksonville, fl	JACKSONVILLE, FL
5	Clayborn	Gethyn	San Antonio, TX	12	12	san antonio, tx	SAN ANTONIO, TX
6	Zorina	Goggen	Alpharetta, GA	11	11	alpharetta, ga	ALPHARETTA, GA

As you can see, what we really want isn't the full text up to the position of the comma, we want to end one position before the comma so that it's not included in our city column. We can do this by subtracting one within the left function.

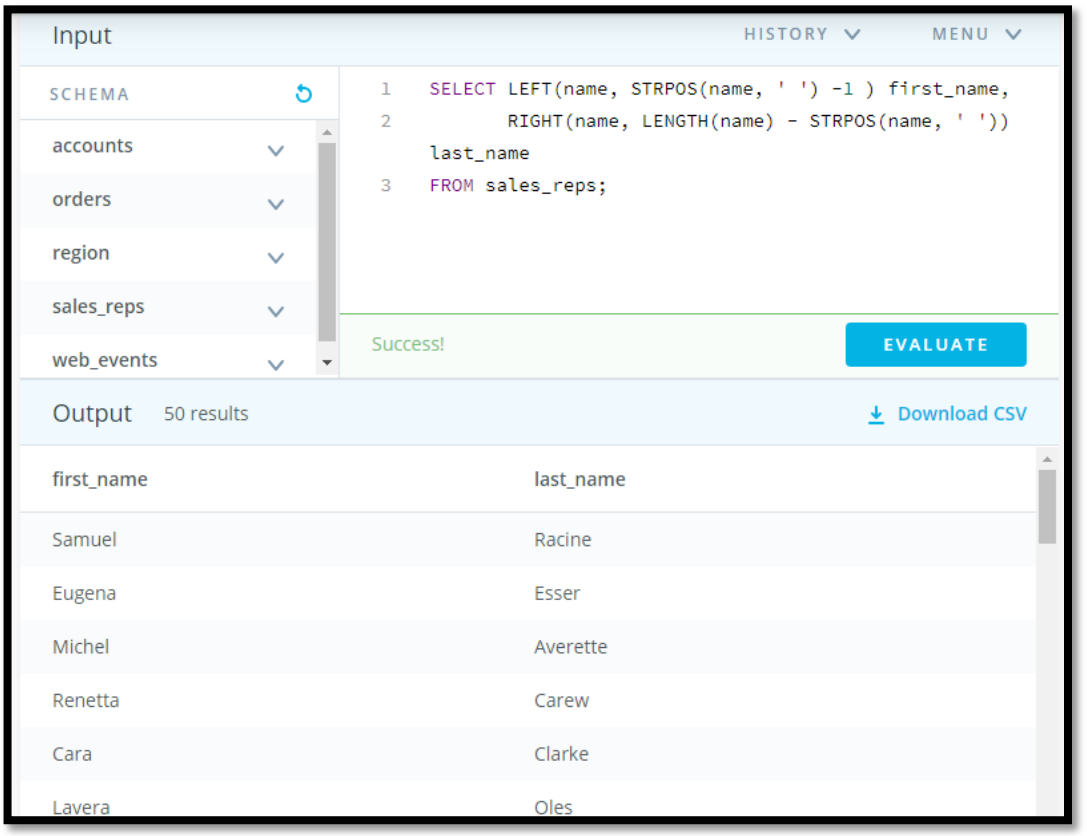
Quiz: POSITION, STRPOS, & SUBSTR – AME DATA AS QUIZ 1

You will need to use what you have learned about **LEFT** & **RIGHT**, as well as what you know about **POSITION** or **STRPOS** to do the following quizzes.

1. Use the accounts table to create **first** and **last** name columns that hold the first and last names for the primary_poc.



2. Now see if you can do the same thing for every rep name in the sales_rep table. Again provide **first** and **last** name columns.



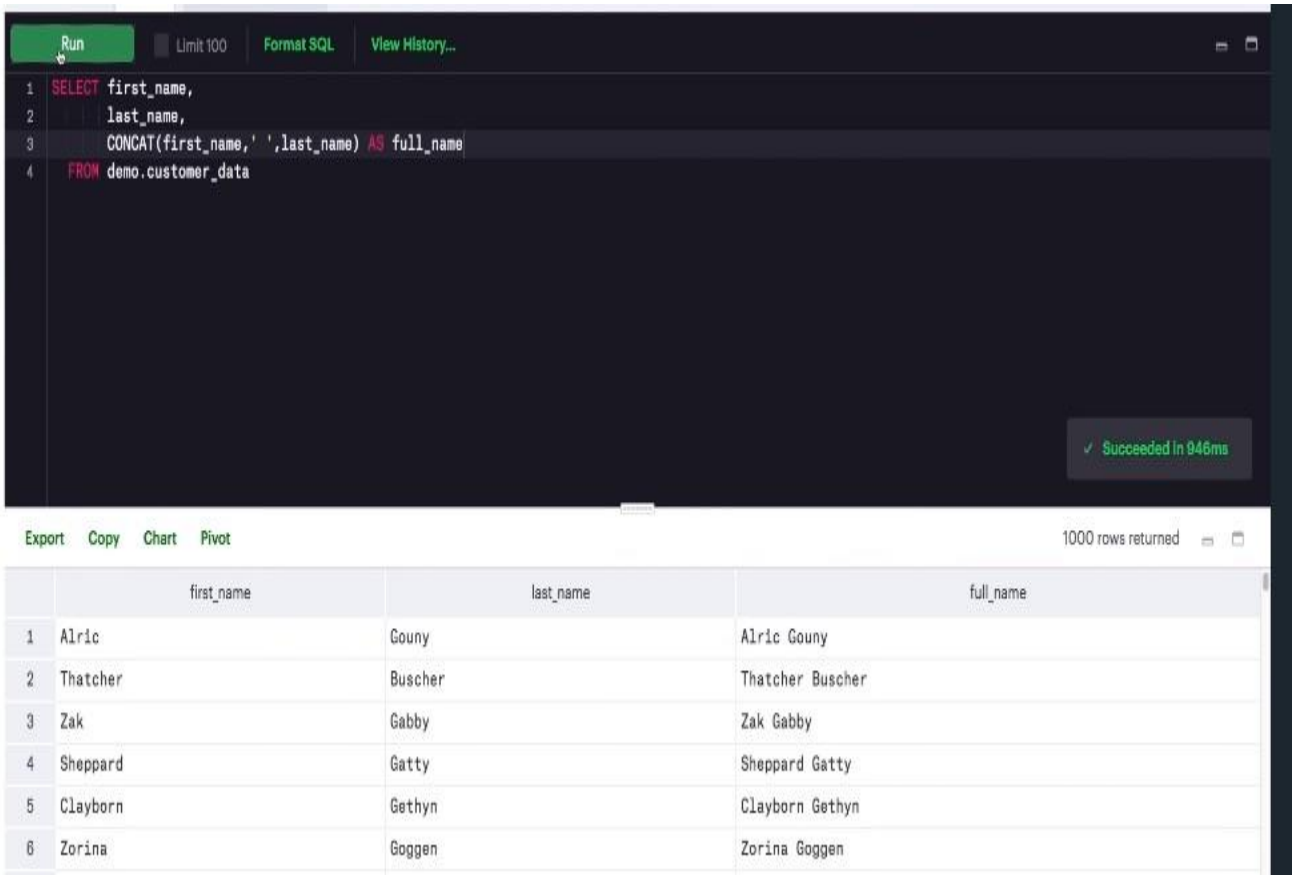
CONCAT

In this lesson you learned about:

1. **CONCAT**
2. Piping ||

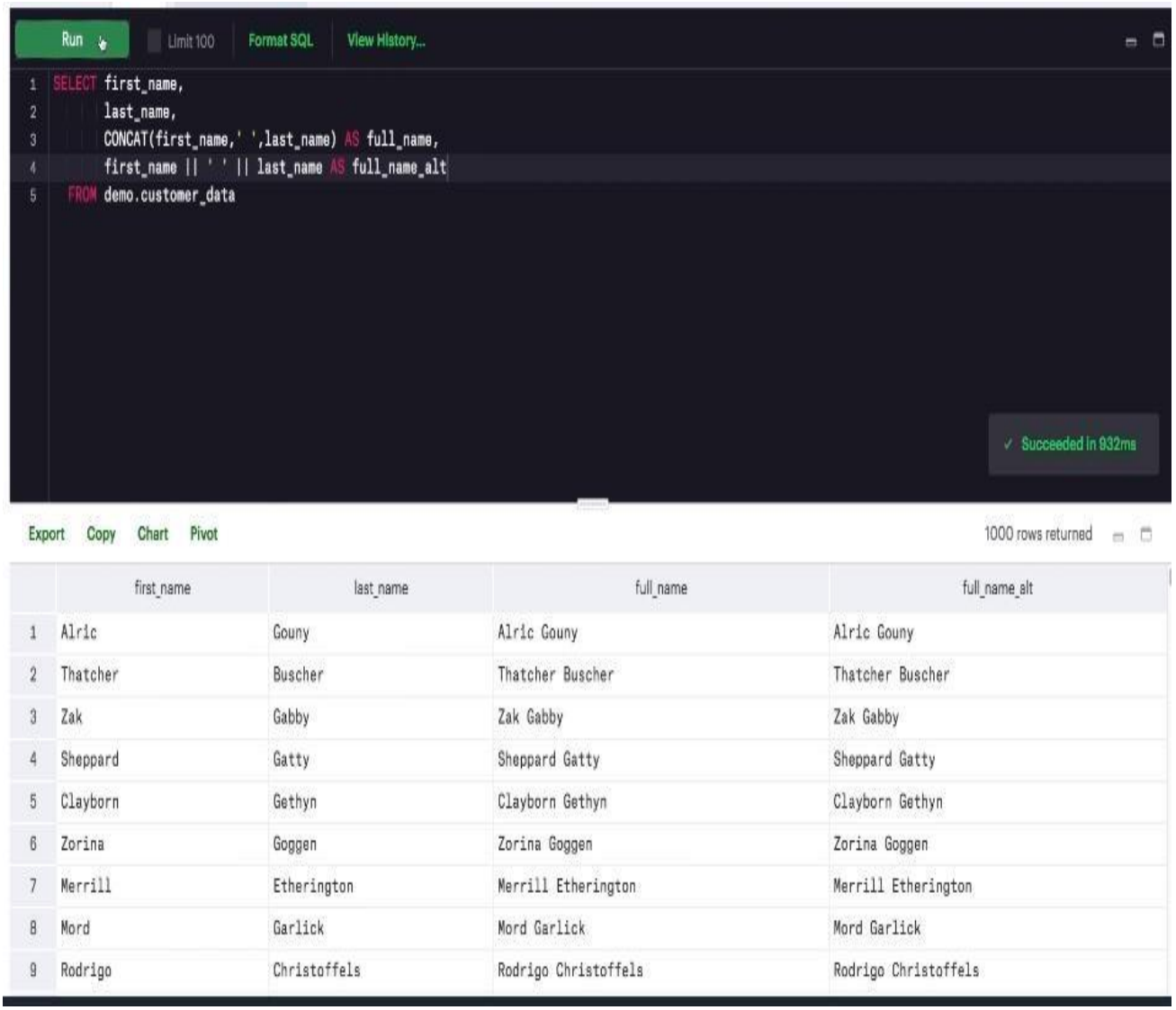
Each of these will allow you to combine columns together across rows. In this video, you saw how first and last names stored in separate columns could be combined together to create a full name: **CONCAT(first_name, ' ', last_name)** or with piping as **first_name || ' ' || last_name**.

Now let's say we want to combine first and last names into a single full name column. You can combine strings from several columns using CONCAT. Simply order the values you want to concatenate and separate them with commas. If you want a hard code values, enclose them in single quotes the way that I'm doing with this space.



	first_name	last_name	full_name
1	Alric	Gouny	Alric Gouny
2	Thatcher	Buscher	Thatcher Buscher
3	Zak	Gabby	Zak Gabby
4	Sheppard	Gatty	Sheppard Gatty
5	Clayborn	Gethyn	Clayborn Gethyn
6	Zorina	Goggen	Zorina Goggen

Alternatively, you can use two pipe characters to perform the same concatenation.

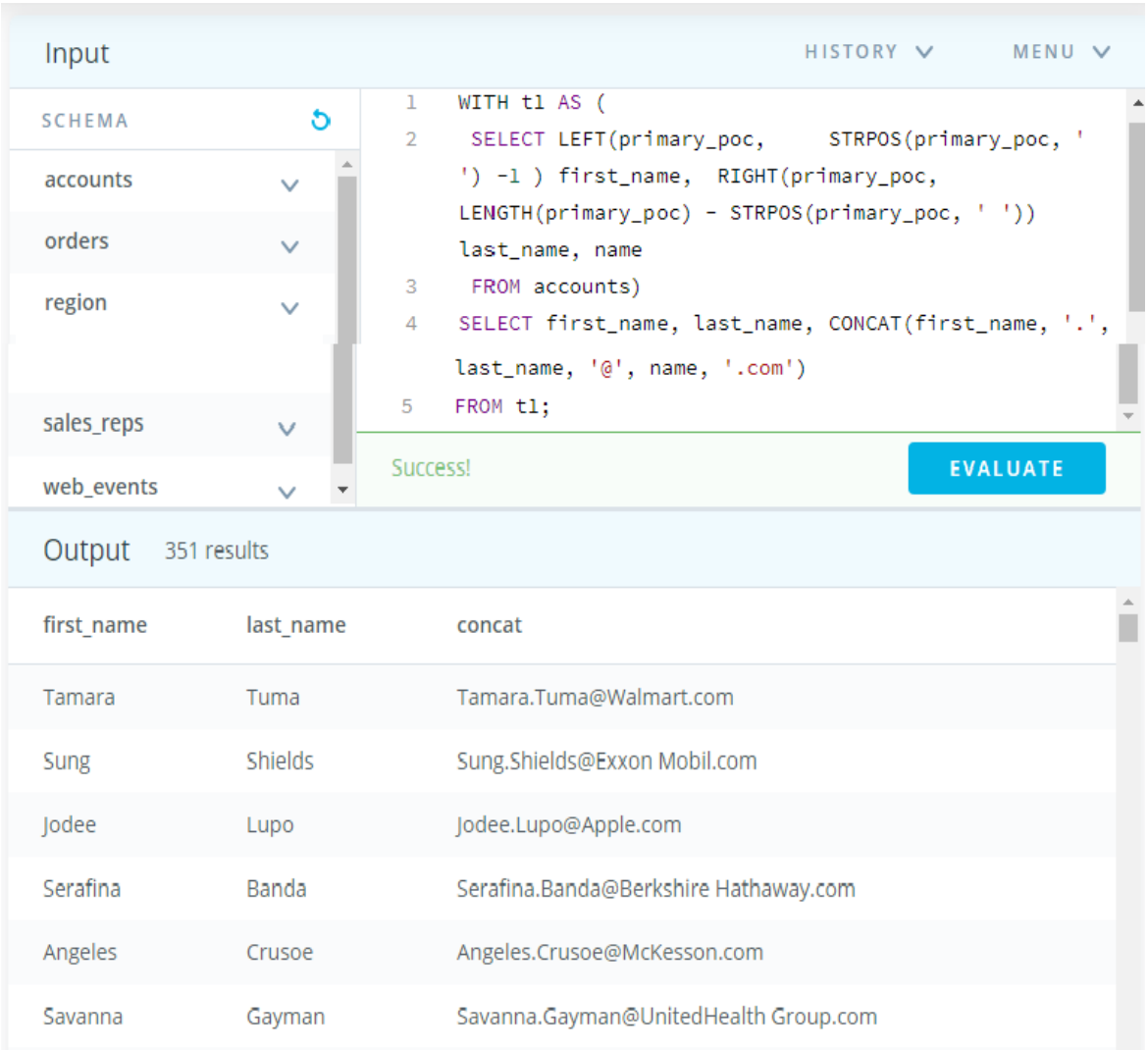


	first_name	last_name	full_name	full_name_alt
1	Alric	Gouny	Alric Gouny	Alric Gouny
2	Thatcher	Buscher	Thatcher Buscher	Thatcher Buscher
3	Zak	Gabby	Zak Gabby	Zak Gabby
4	Sheppard	Gatty	Sheppard Gatty	Sheppard Gatty
5	Clayborn	Gethyn	Clayborn Gethyn	Clayborn Gethyn
6	Zorina	Goggen	Zorina Goggen	Zorina Goggen
7	Merrill	Etherington	Merrill Etherington	Merrill Etherington
8	Mord	Garlick	Mord Garlick	Mord Garlick
9	Rodrigo	Christoffels	Rodrigo Christoffels	Rodrigo Christoffels

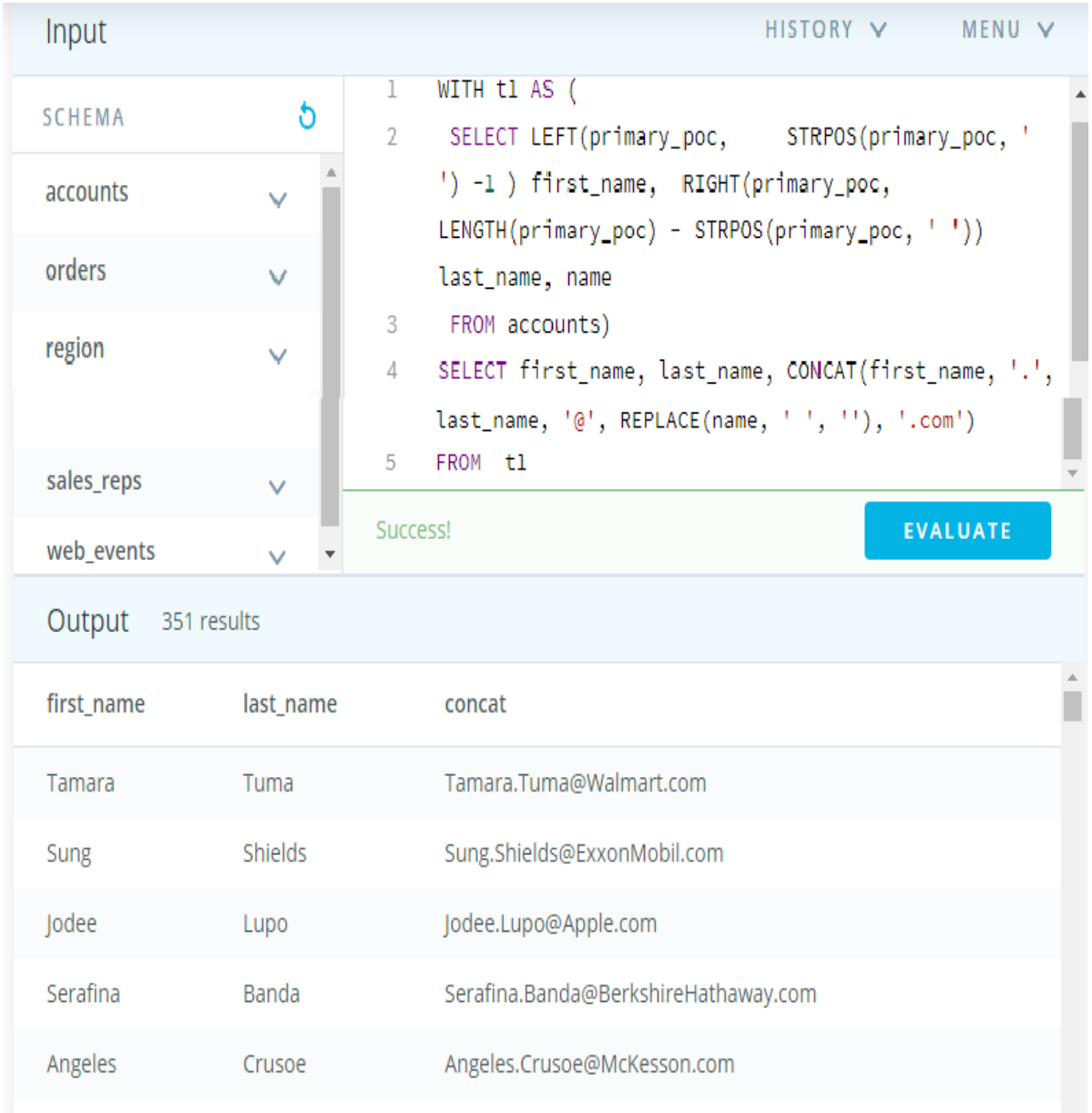
Quiz: CONCAT

Quizzes CONCAT

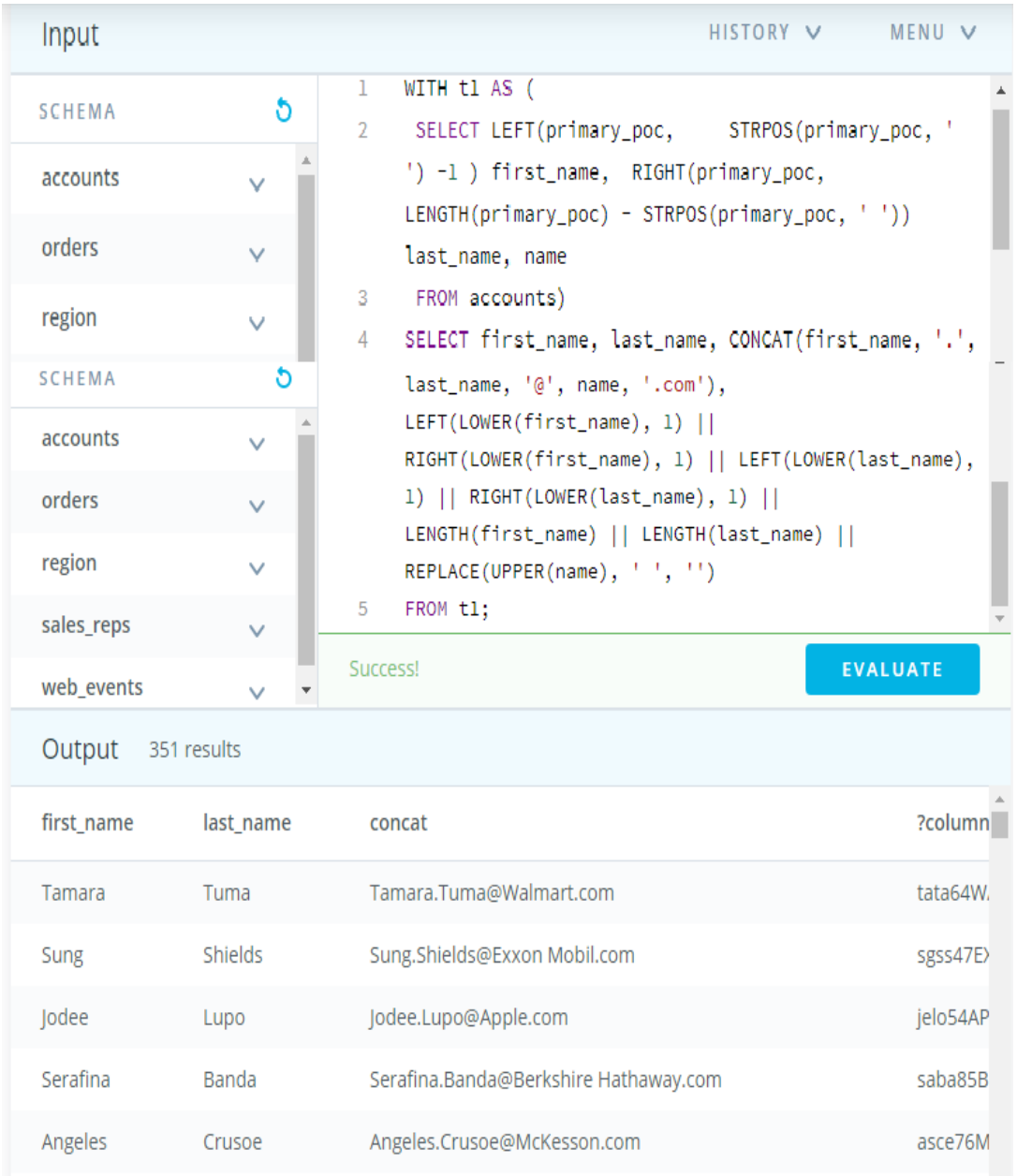
1. Each company in the accounts table wants to create an email address for each primary_poc. The email address should be the first name of the **primary_poc** . last name **primary_poc** @ company name .com.



2. You may have noticed that in the previous solution some of the company names include spaces, which will certainly not work in an email address. See if you can create an email address that will work by removing all of the spaces in the account name, but otherwise your solution should be just as in question 1.



3. We would also like to create an initial password, which they will change after their first log in. The first password will be the first letter of the primary_poc's first name (lowercase), then the last letter of their first name (lowercase), the first letter of their last name (lowercase), the last letter of their last name (lowercase), the number of letters in their first name, the number of letters in their last name, and then the name of the company they are working with, all capitalized with no spaces.



CAST

In this video, you saw additional functionality for working with dates including:

1. **TO_DATE**
2. **CAST**
3. Casting with ::

DATE_PART('month', TO_DATE(month, 'month')) here changed a month name into the number associated with that particular month.

Then you can change a string to a date using **CAST**. **CAST** is actually useful to change lots of column types. Commonly you might be doing as you saw here, where you change a string to a date using **CAST(date_column AS DATE)**. However, you might want to make other changes to your columns in terms of their data types. You can see other examples here.
In this example, you also saw that instead of **CAST(date_column AS DATE)**, you can use **date_column::DATE**.

Expert Tip

Most of the functions presented in this lesson are specific to strings. They won't work with dates, integers or floating-point numbers. However, using any of these functions will automatically change the data to the appropriate type.

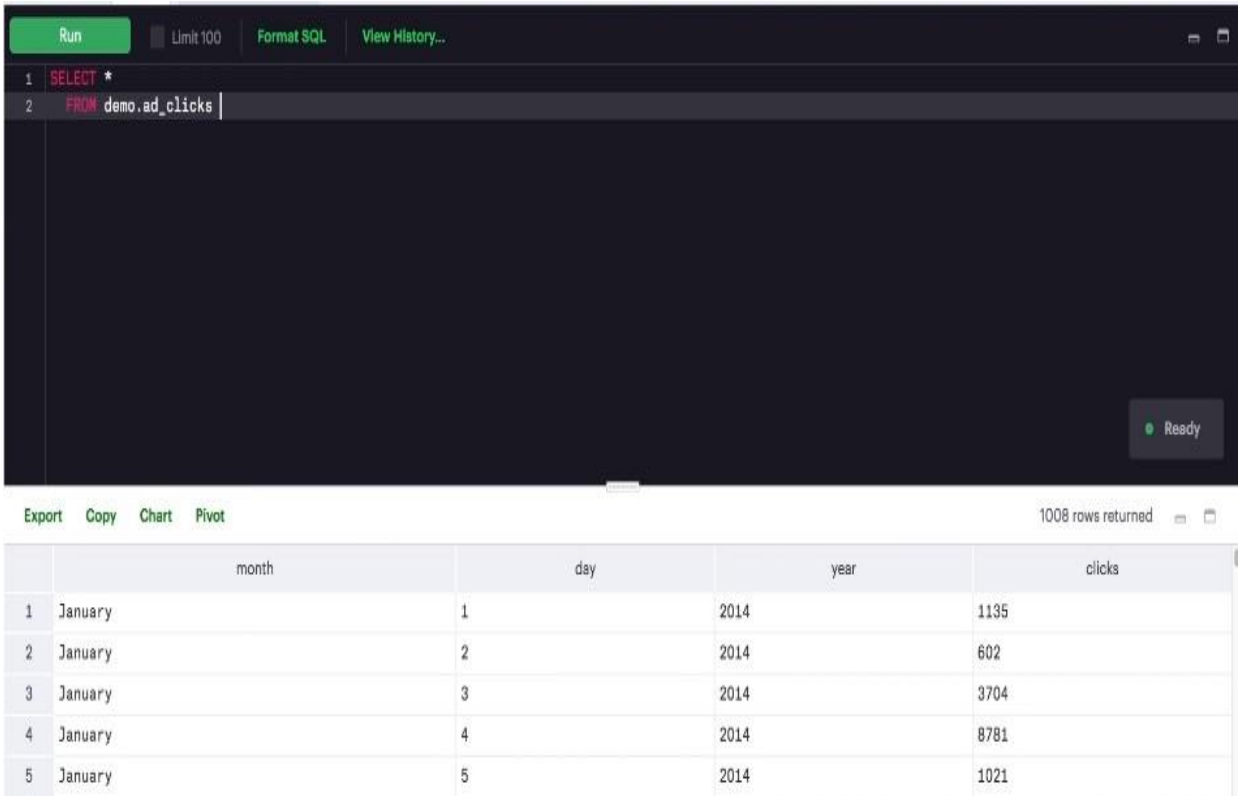
LEFT, **RIGHT**, and **TRIM** are all used to select only certain elements of strings, but using them to select elements of a number or date will treat them as strings for the purpose of the function.

Though we didn't cover **TRIM** in this lesson explicitly, it can be used to remove characters from the beginning and end of a string. This can remove unwanted spaces at the beginning or end of a row that often happen with data being moved from Excel or other storage systems.

There are a number of variations of these functions, as well as several other string functions not covered here. Different databases use subtle variations on these functions, so be sure to look up the appropriate database's syntax if you're connected to a private database.The **Postgres literature** contains a lot of the related functions.

We've already explored date functions like date trunk and date part. In an upcoming lesson, we'll dig into even more functions for getting value from dates. The vast majority of analysis you perform in a professional setting will have dates attached to it, but you might not be able to get value out of them, if the dates are not formatted correctly.

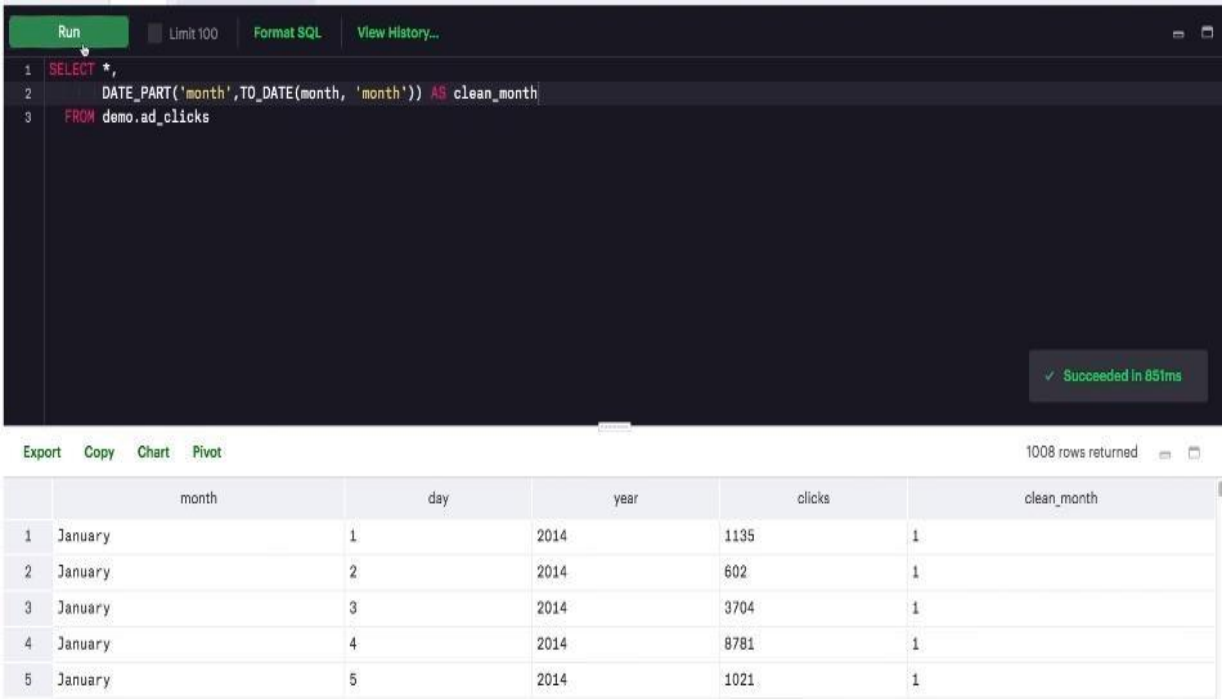
Let's say you've got three separate columns for day, month, and year of a given record.



	month	day	year	clicks
1	January	1	2014	1135
2	January	2	2014	602
3	January	3	2014	3704
4	January	4	2014	8781
5	January	5	2014	1021

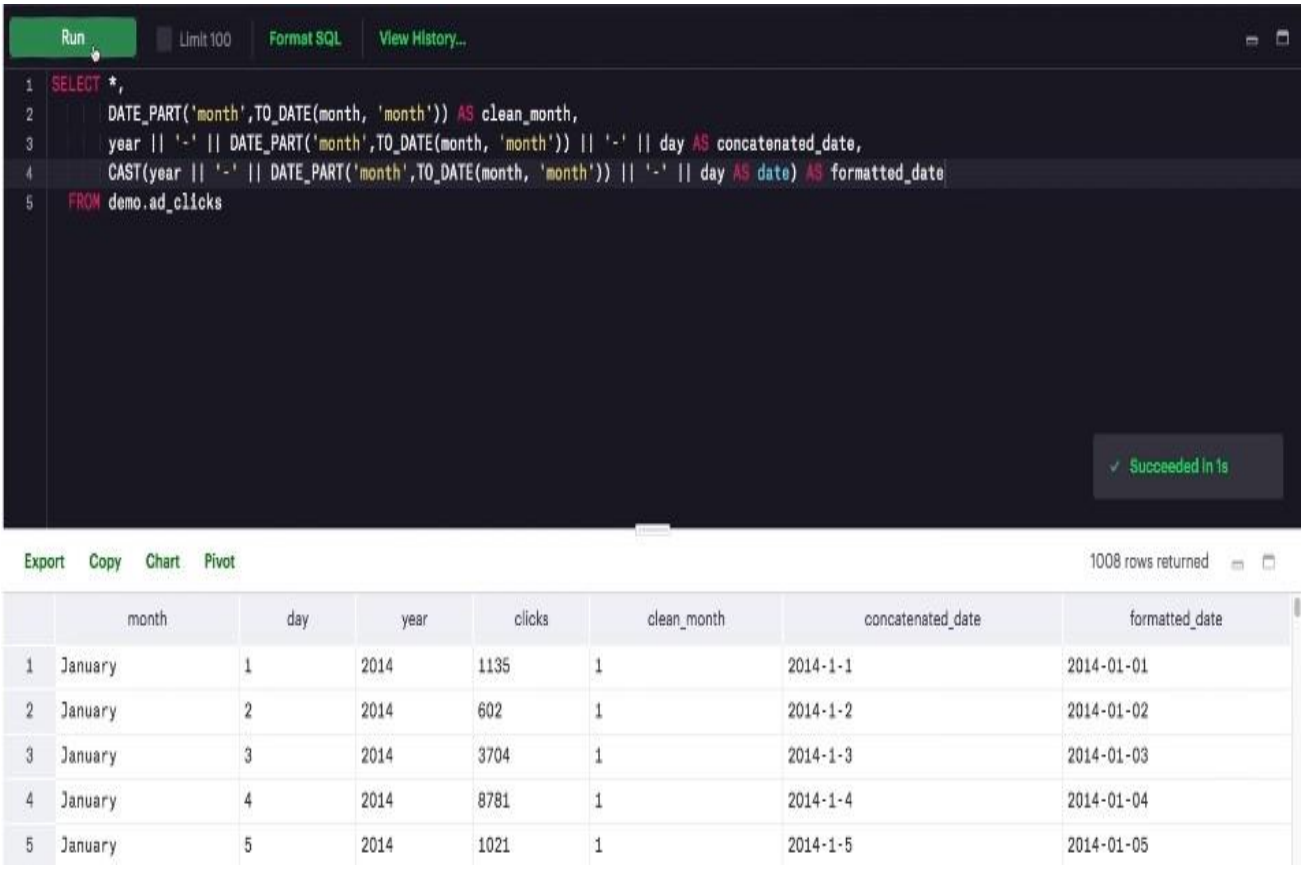
Problems like this are common for data that was manipulated in excel at some point, or exported from a third party system, or even data than was just manually entered.

In order to really maximize value here, you'll need to convert month names into numbers, then concatenate all of these fields together along with hyphens, and then tell the database to understand that the resulting output is a date, which you can do by using the cast function.
Let's tackle the first couple steps of this first.



	month	day	year	clicks	clean_month
1	January	1	2014	1135	1
2	January	2	2014	602	1
3	January	3	2014	3704	1
4	January	4	2014	8781	1
5	January	5	2014	1021	1

This first step converted month names into numbers. As you can see, January comes out as one. Now let's concatenate that value together with the year end the day to create something that looks like a real date. For the last step, we'll cast this new field as a date format.



	month	day	year	clicks	clean_month	concatenated_date	formatted_date
1	January	1	2014	1135	1	2014-1-1	2014-01-01
2	January	2	2014	602	1	2014-1-2	2014-01-02
3	January	3	2014	3704	1	2014-1-3	2014-01-03
4	January	4	2014	8781	1	2014-1-4	2014-01-04
5	January	5	2014	1021	1	2014-1-5	2014-01-05

You can see here that the database now understand this as a date and has attached the appropriate zeros to these records so that it reads properly. The cast function can be a little tricky to read, but luckily there's also a shorthand. Rather than wrapping all of this in a cast function, we can simply add two colons and then the data type we'd like to cast at the end of this section.

The cast function is most useful for turning strings into numbers or dates.

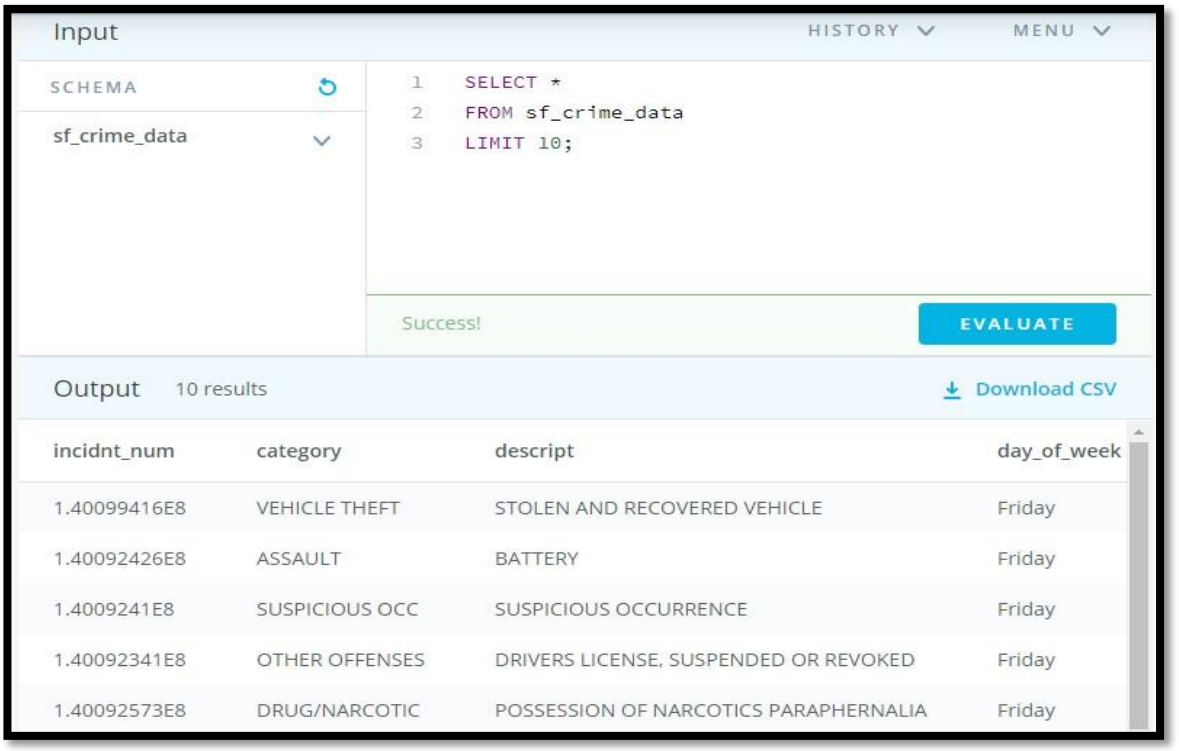
Typically, if you want to turn a number into a string performing any type of string operation on it like left, right, or substring will automatically cast the data while performing the operation.

Quiz: CAST

For this set of quiz questions, you are going to be working with a single table in the environment below. This is a different dataset than Parch & Posey, as all of the data in that particular dataset were already clean.

Tasks to complete:

1. Write a query to look at the top 10 rows to understand the columns and the raw data in the dataset called sf_crime_data.



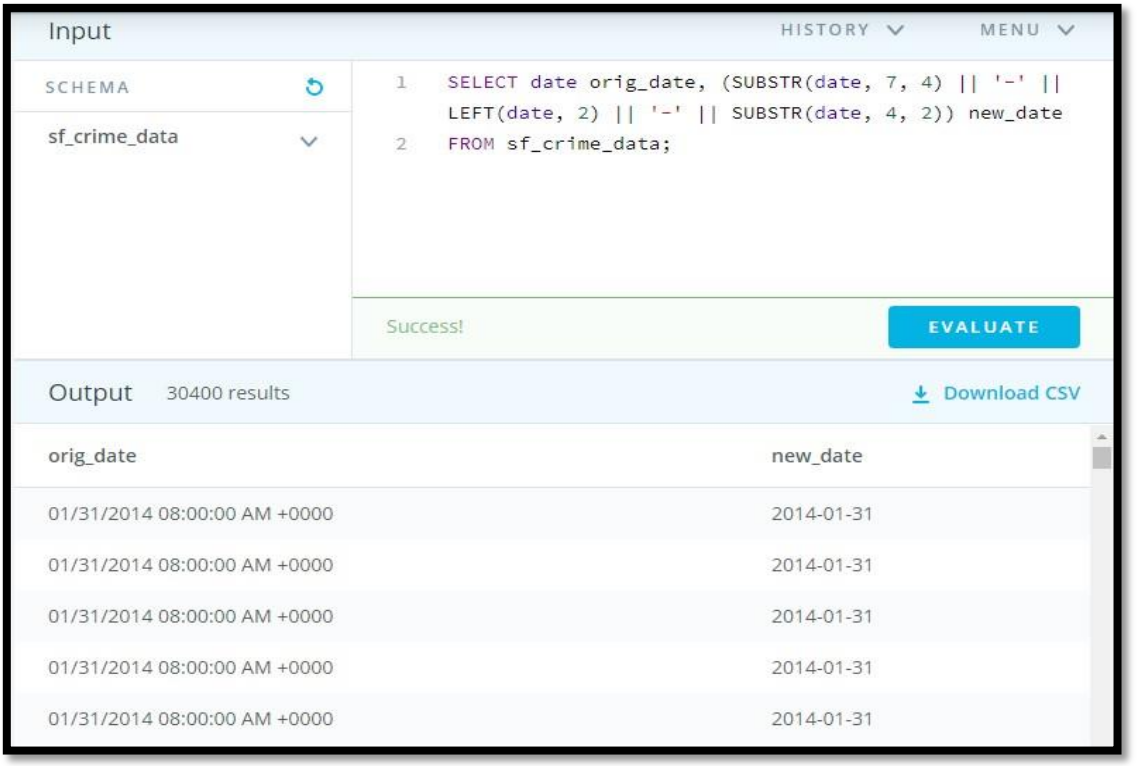
2. Remembering back to the lesson on dates, use the **Quiz Question** at the bottom of this page to make sure you remember the format that dates should use in SQL.

yyyy-mm-dd

3. Look at the date column in the **sf_crime_data** table. Notice the date is not in the correct format.

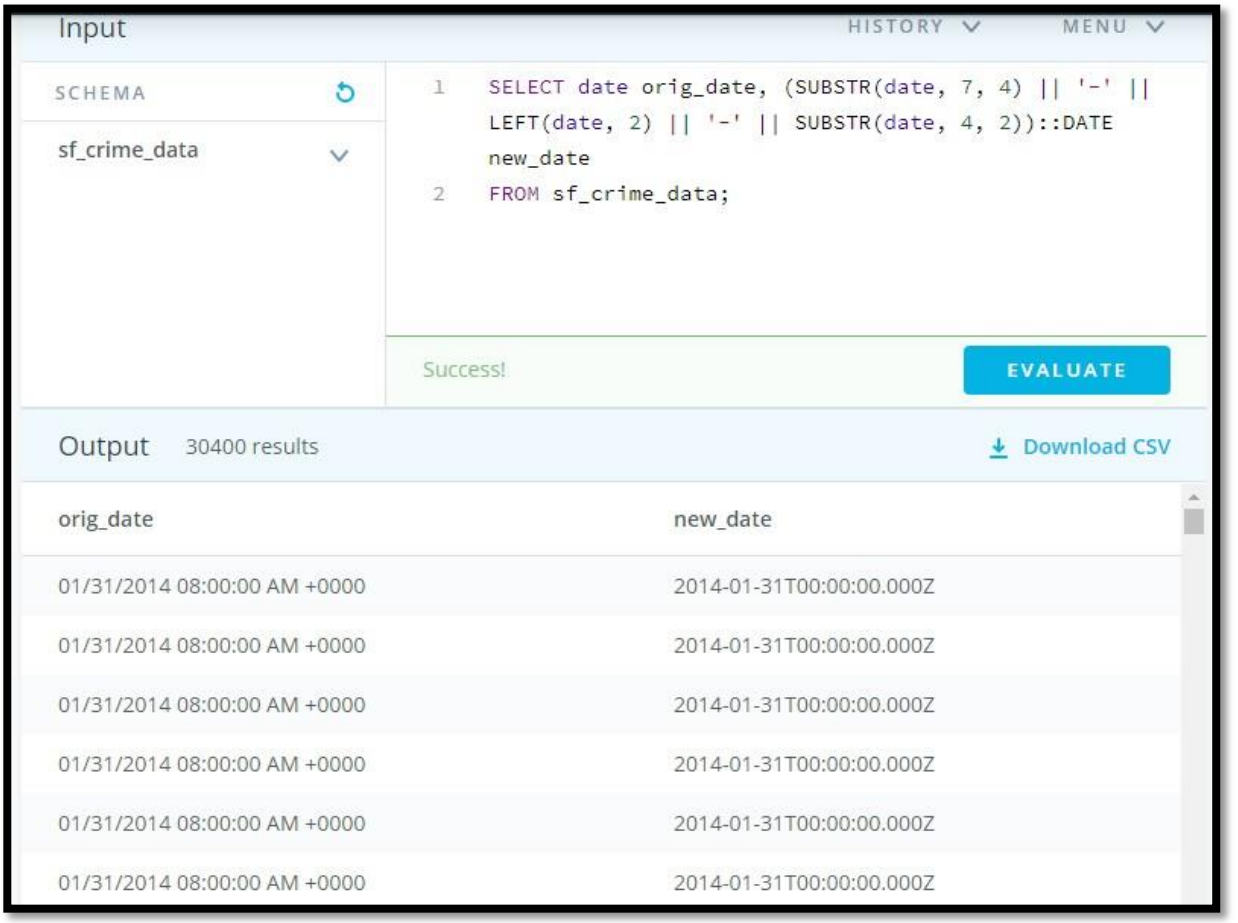
The format of the date column is **mm/dd/yyyy** with times that are not correct also at the end of the date.

4. Write a query to change the date into the correct SQL date format. You will need to use at least **SUBSTR** and **CONCAT** to perform this operation.



5. Once you have created a column in the correct format, use either CAST or :: to convert this to a date.

Notice, this new date can be operated on using **DATE_TRUNC**.



QUIZ QUESTION

What is the correct format of dates in SQL?

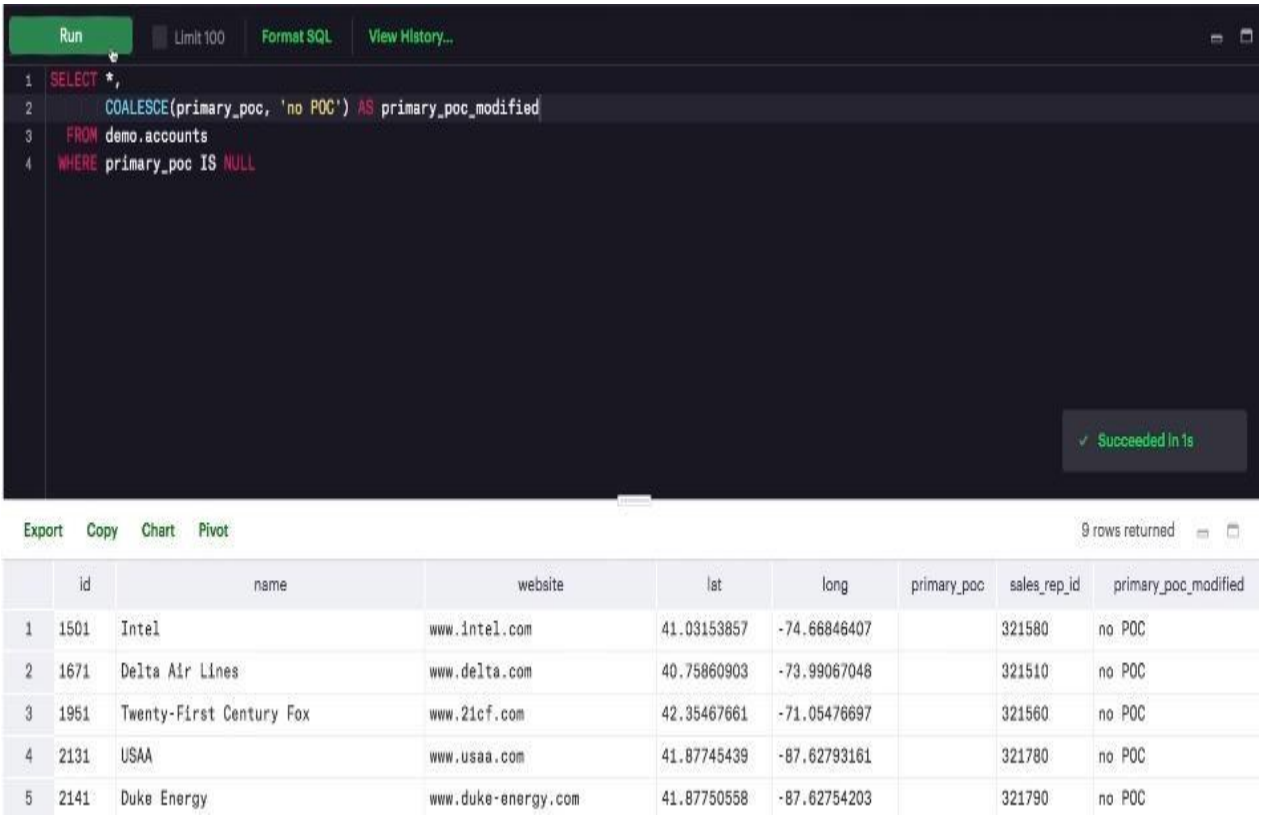
- dd-mm-yyyy
- mm-dd-yyyy
- yyyy-mm-dd
- yyyy-dd-mm

COALESCE

In this lesson, you learned about how to use **COALESCE** to work with NULL values. Unfortunately, our dataset does not have the **NULL** values that were fabricated in this dataset, so you will work through a different example in the next concept to get used to the **COALESCE** function.

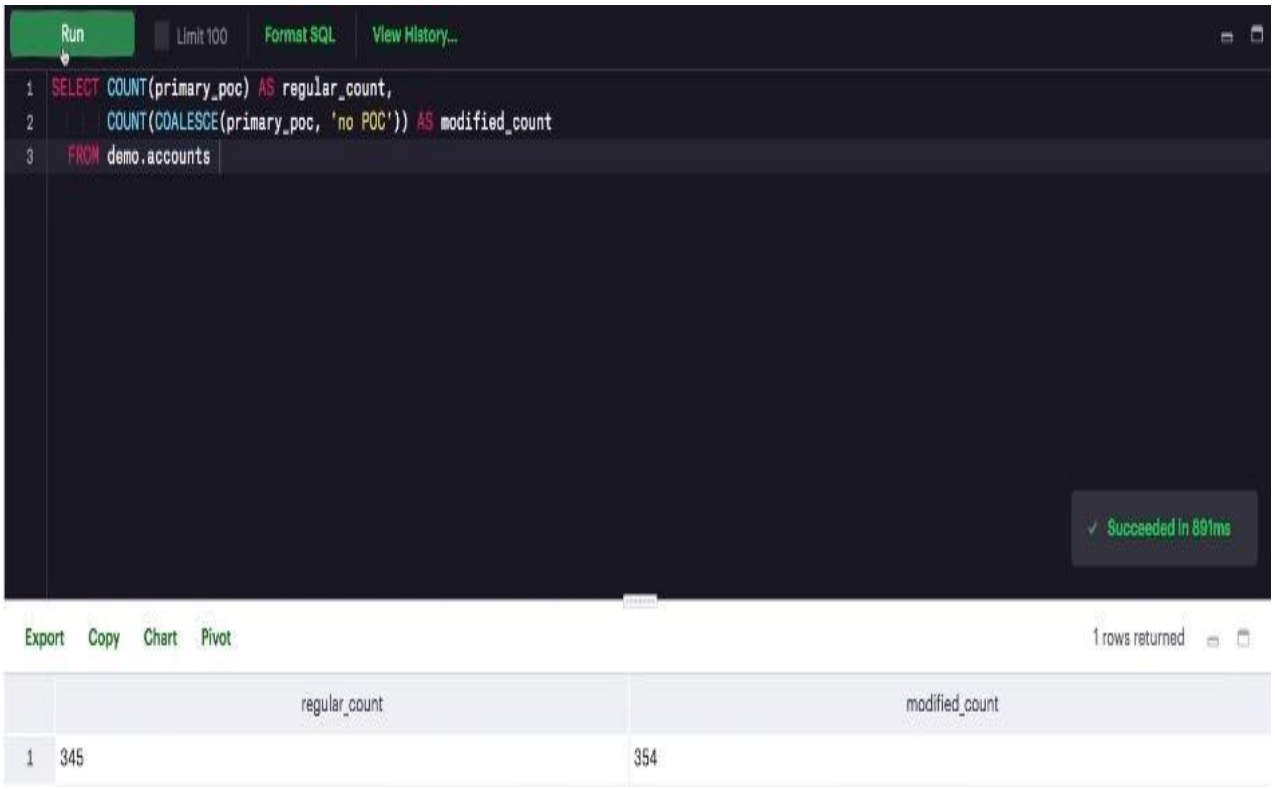
In general, **COALESCE** returns the first non-NULL value passed for each row. Hence why the lesson used **no_poc** if the value in the row was NULL.

Occasionally, you'll end up with a data set that has some nulls that you'd prefer to contain actual values. Looking at the accounts table, you might want to clearly label a no primary point of contact as no POC, so the results will be easily understandable. In cases like this, you can use coalesce to replace the null values.



	id	name	website	lat	long	primary_poc	sales_rep_id	primary_poc_modified
1	1501	Intel	www.intel.com	41.03153857	-74.66846407		321580	no POC
2	1671	Delta Air Lines	www.delta.com	40.75860903	-73.99067048		321510	no POC
3	1951	Twenty-First Century Fox	www.21cf.com	42.35467661	-71.05476697		321560	no POC
4	2131	USAA	www.usaa.com	41.87745439	-87.62793161		321780	no POC
5	2141	Duke Energy	www.duke-energy.com	41.87750558	-87.62754203		321790	no POC

This is something you may want to do frequently when using numerical data where you might want to display nulls as zero. Also, when performing outer joins that result in some unmatched rows, you may want those unmatched rows to display something other than a null value. Of course, this is most valuable when working with a function that treats nulls differently from zero such as a count or an average. We can demonstrate this by wrapping the coalesce in a count function and counting the primary POC column without the coalesce, as well.



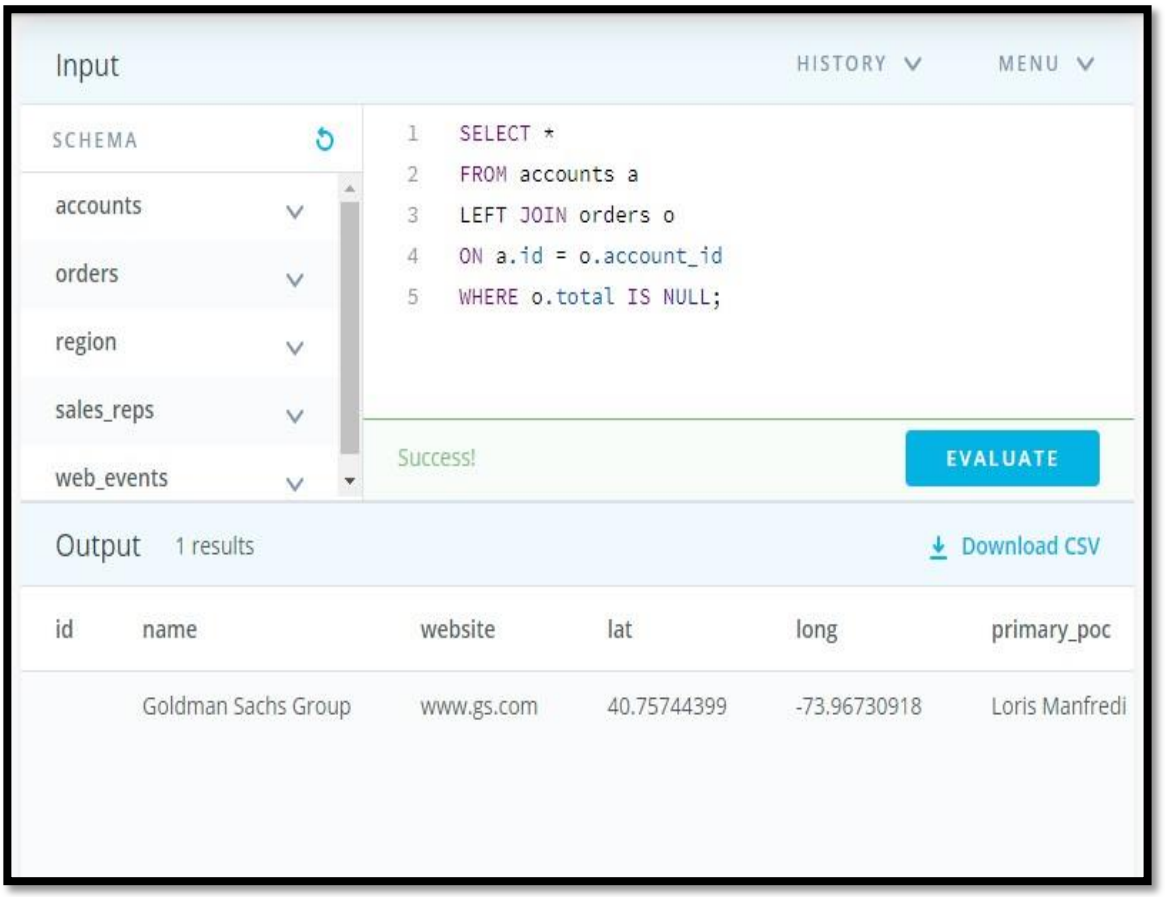
As you can see, the count that includes the coalesce, includes nine more results than the other regular

Quiz: COALESCE

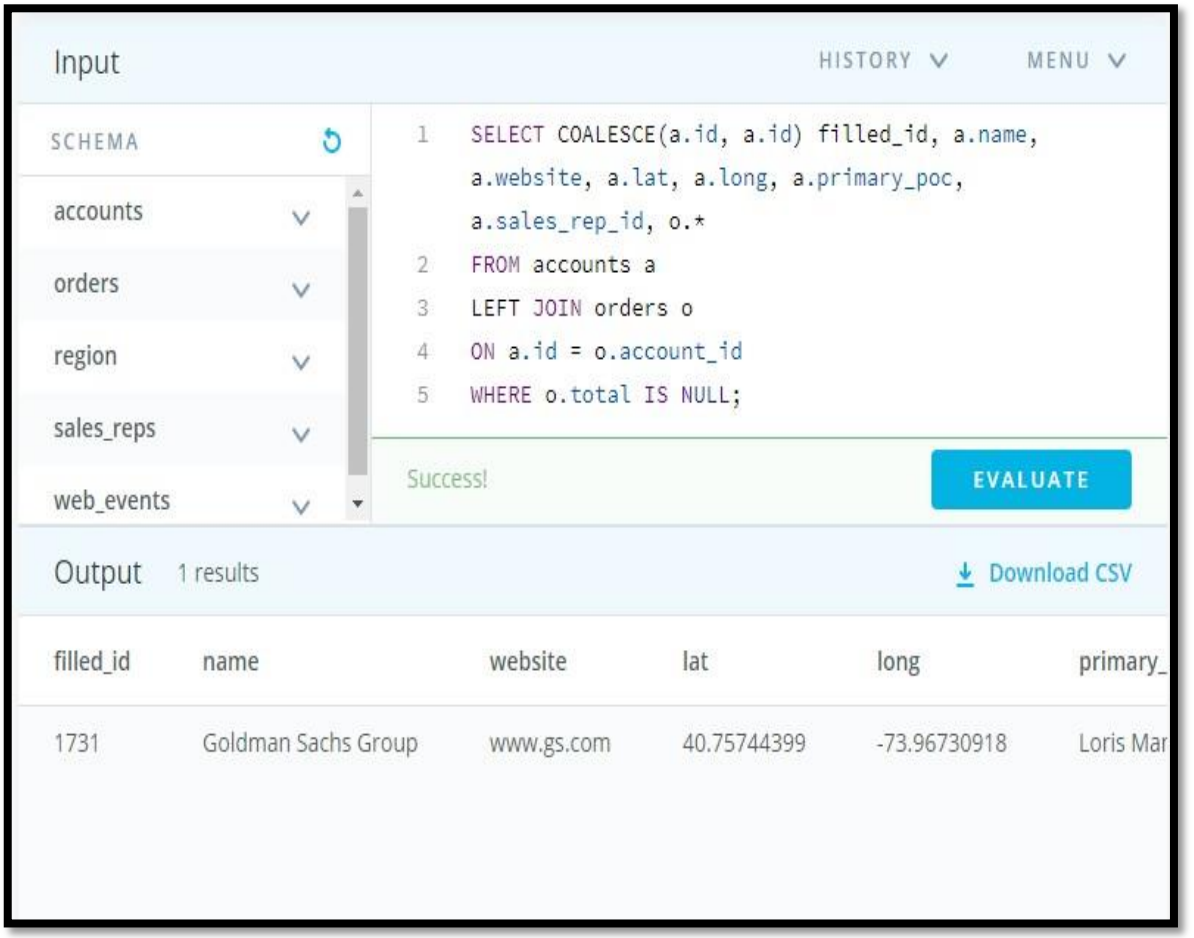
In this quiz, we will walk through the previous example using the following task list.

Tasks to complete:

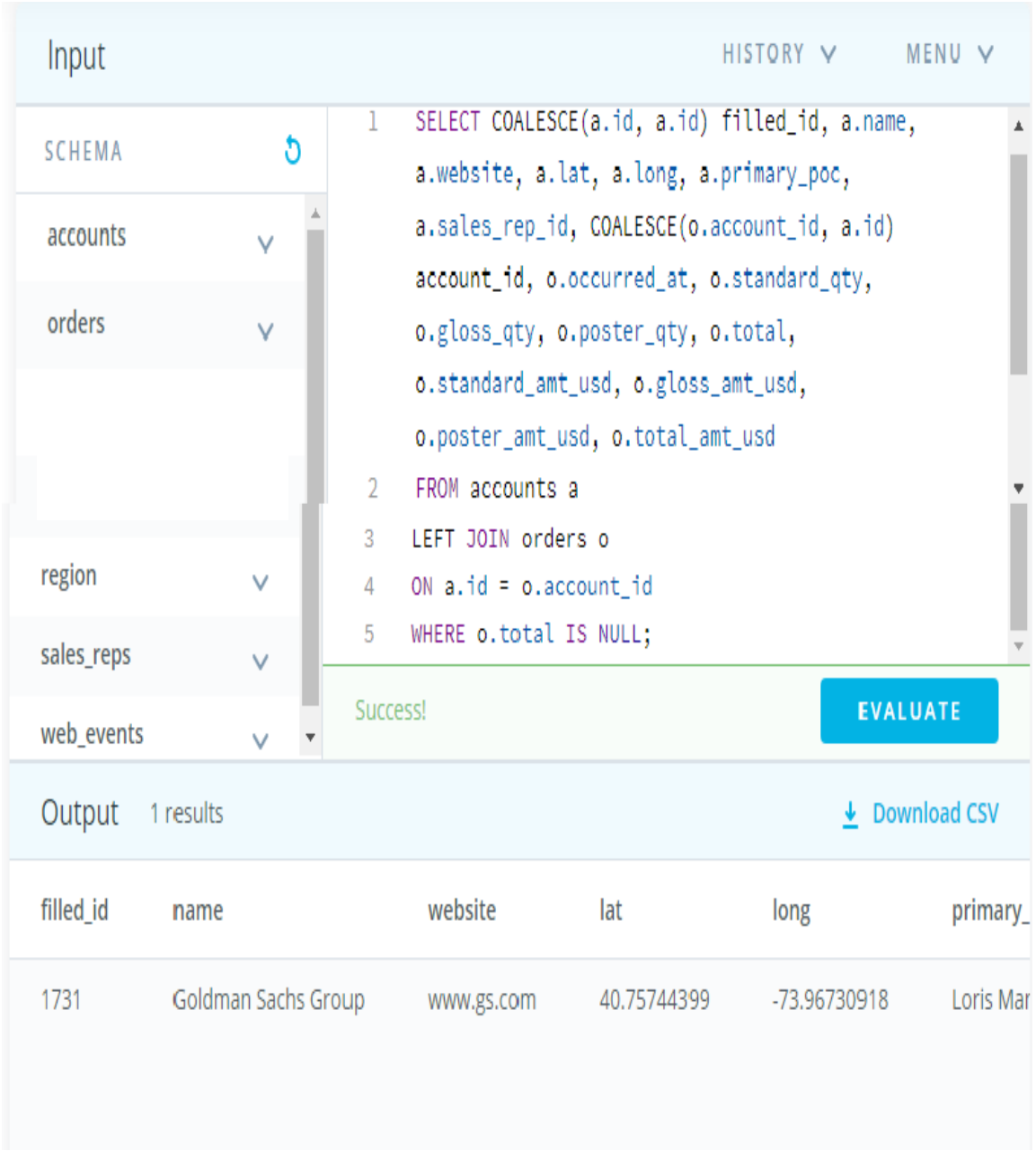
1. Run the query below to notice the row with missing data.



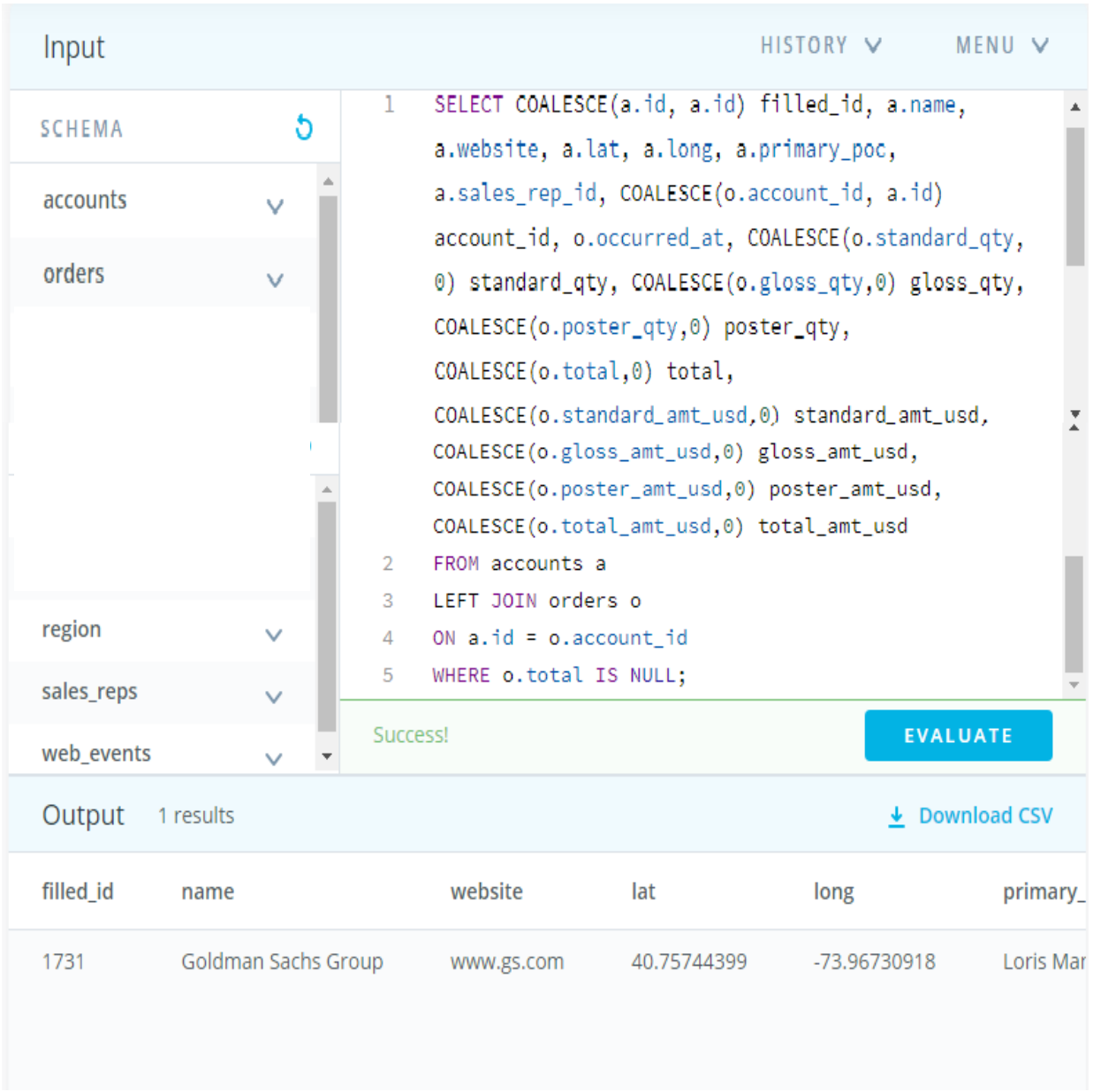
2. Use **COALESCE** to fill in the accounts.id column with the account.id for the NULL value for the table in 1.



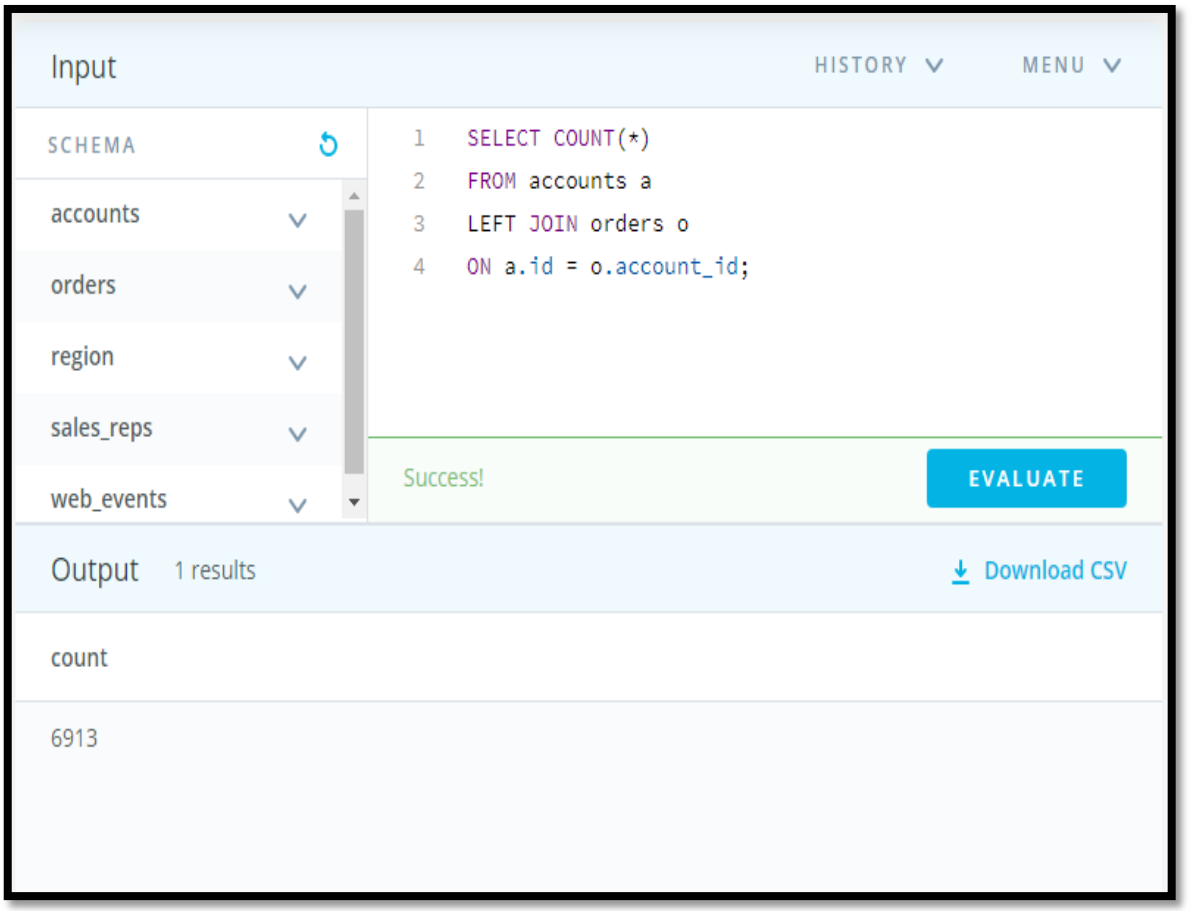
3. Use **COALESCE** to fill in the orders.account_id column with the account.id for the NULL value for the table in 1.



4. Use **COALESCE** to fill in each of the **qty** and **usd** columns with 0 for the table in 1.



5.Run the query in 1 with the **WHERE** removed and **COUNT** the number of ids .



6.Run the query in 5, but with the **COALESCE** function used in questions 2 through 4.