SQL JOINS

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 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 would hold a ton of information and it'd be hard to determine a row column structure for so many different types of data**. Databases give us the flexibility to keep things organized neatly in their own tables, so they are easy to find and work with while allowing us to combine tables as needed to solve problems that require several types of data.**

We will try to understand joins using the parch and posey database.

Why would we want to split data into separate tables?

If we look at the orders data table of parch and posey, we 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 need to join another table to connect this data to names.

	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

But first why isn't the customer's name in this table in the first place?

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

This multi-table structure allows queries to execute more quickly.

The accounts and orders tables store fundamentally different types of objects.

Parch and Posey probably only wants one account per company, and they want it 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 are filled. A given customer might have multiple orders, and rather than change a past order, parch and posey might just add a new one. Because these objects operate differently, it makes sense for them to live in different tables.

Another reason to store this information separately has to do with the speed at which database can modify data. When we write a query its execution speed depends on the amount of data we're asking the database to read and the number and type of calculations you're asking it to make.

Database Normalization

Normalization is a technique that allows us to think about the way data will be stored when creating a database.

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

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

The primary key (pkey) is an attribute (column) is a table that acts as a unique identifier for each row in a table. A foreign key (fkey) is an attribute that forms an implied link between two tables that are in a 1:M relationship. The fkey, which is a column in the table of the many, is usually a pkey in the table of the one.

<u>Introduction to joins</u>

1. The whole purpose of Join is to allow us to pull data from more than one table at a time.

```
SELECT orders.*

FROM orders

JOIN accounts

ON orders.account_id = accounts.id;
```

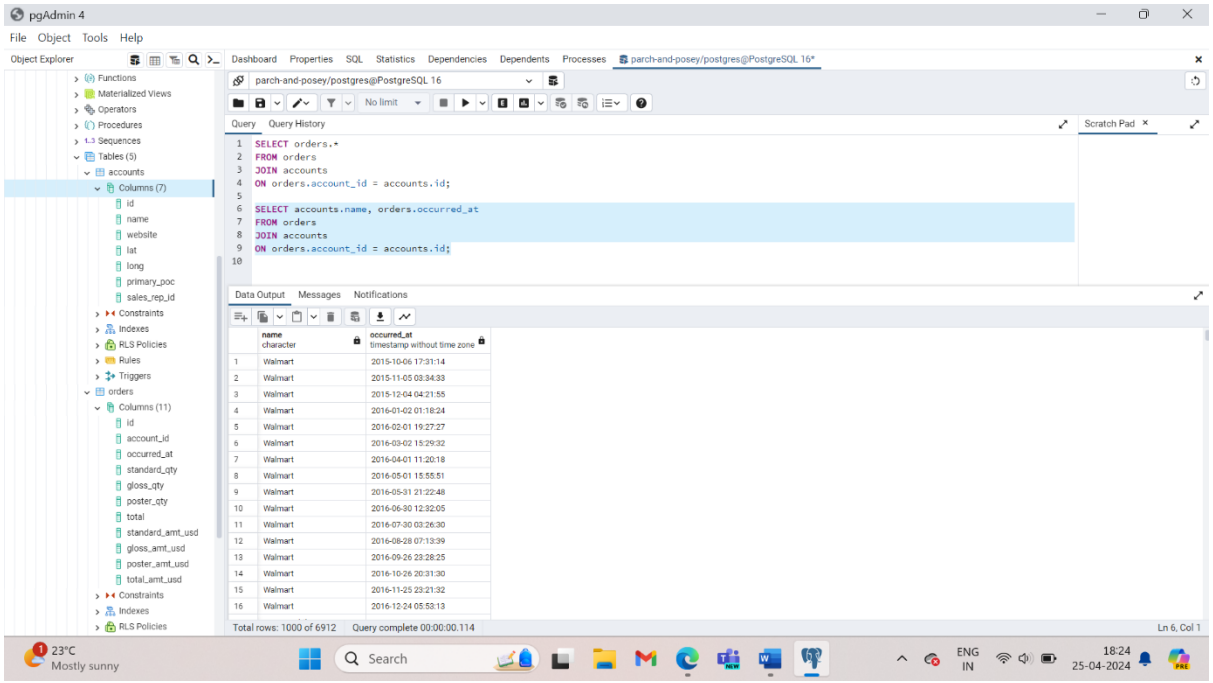
2. If we want to pull only the **account name** and the dates in which that account placed an order, but none of the other columns, we can do this with the following query:

```
SELECT accounts.name, orders.occurred_at

FROM orders

JOIN accounts

ON orders.account_id = accounts.id;
```



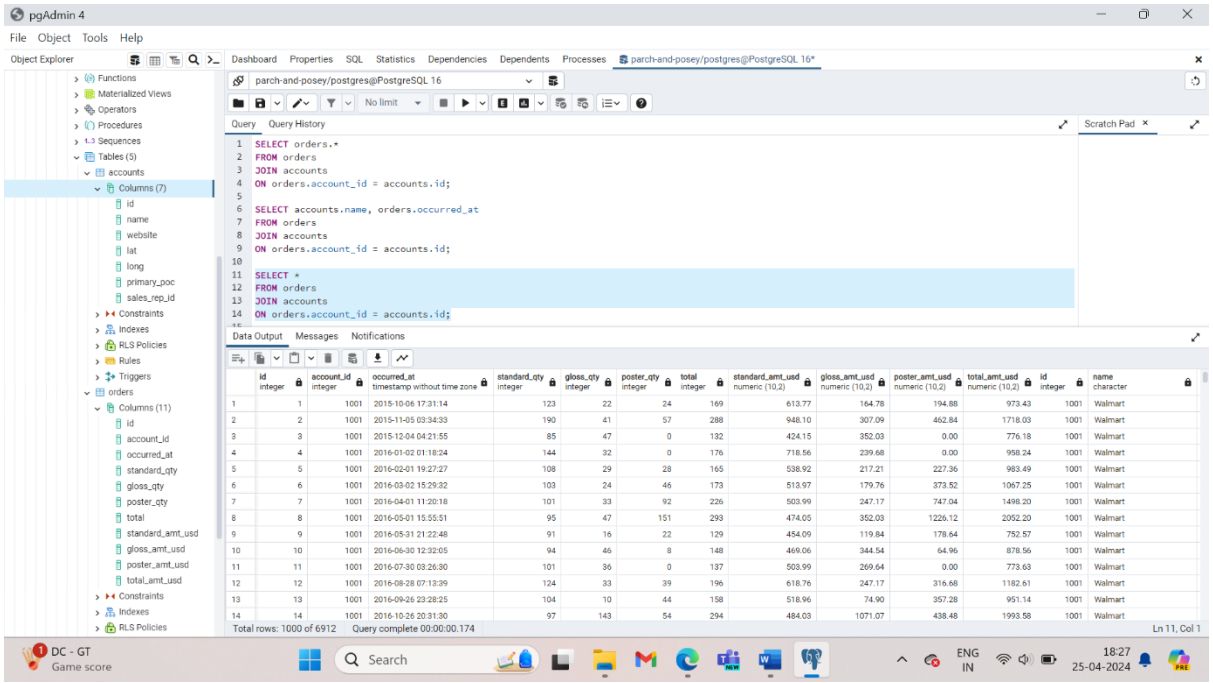
3. This query pulls all the columns from *both* the **accounts** and **orders** table.

SELECT *

FROM orders

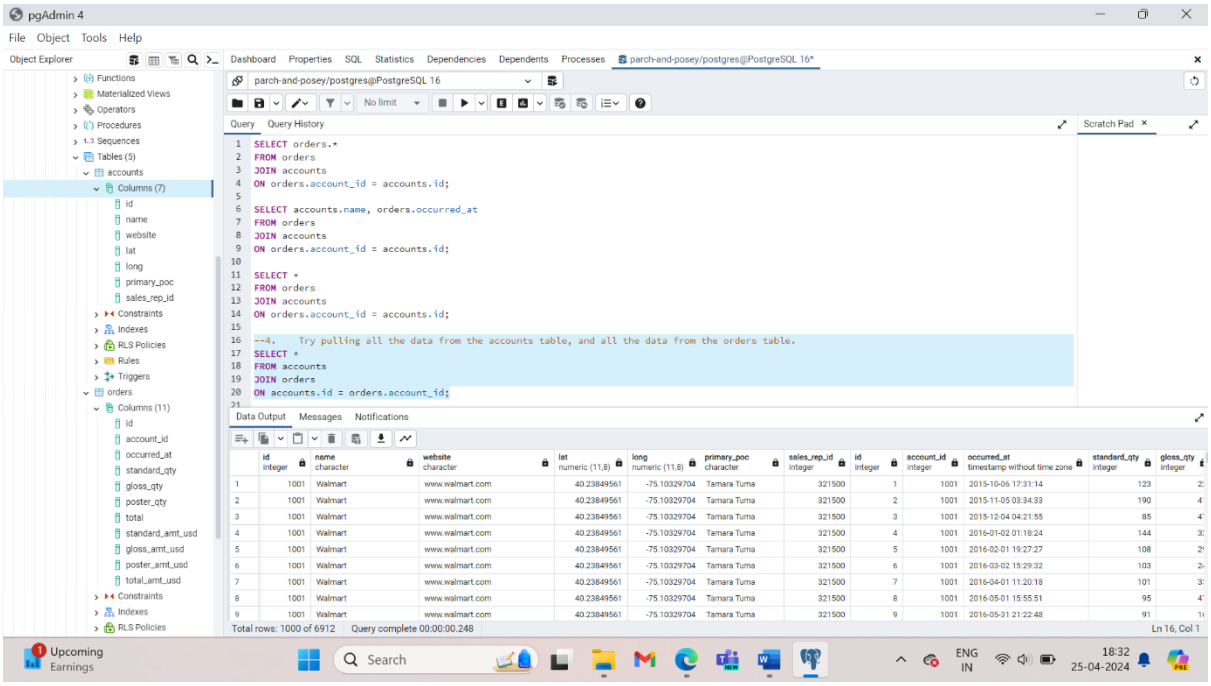
JOIN accounts

ON orders.account_id = accounts.id;

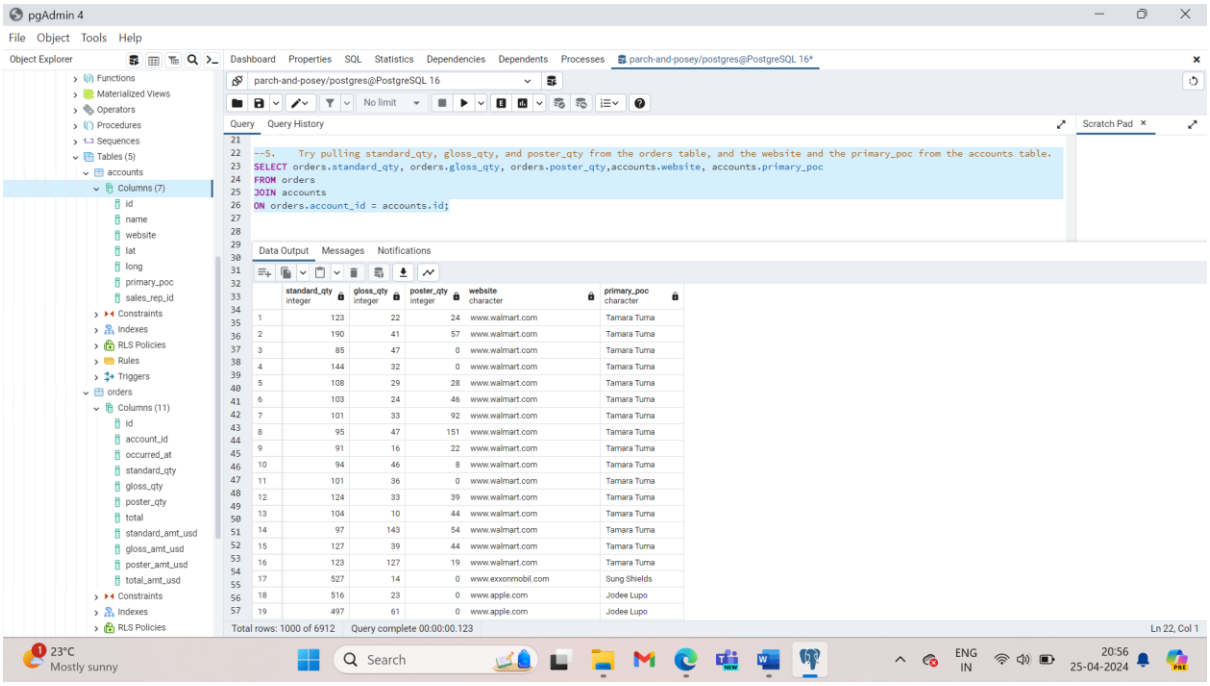


4. Try pulling all the data from the **accounts** table, and all the data from the **orders** table.
   SELECT *
   FROM accounts
   JOIN orders

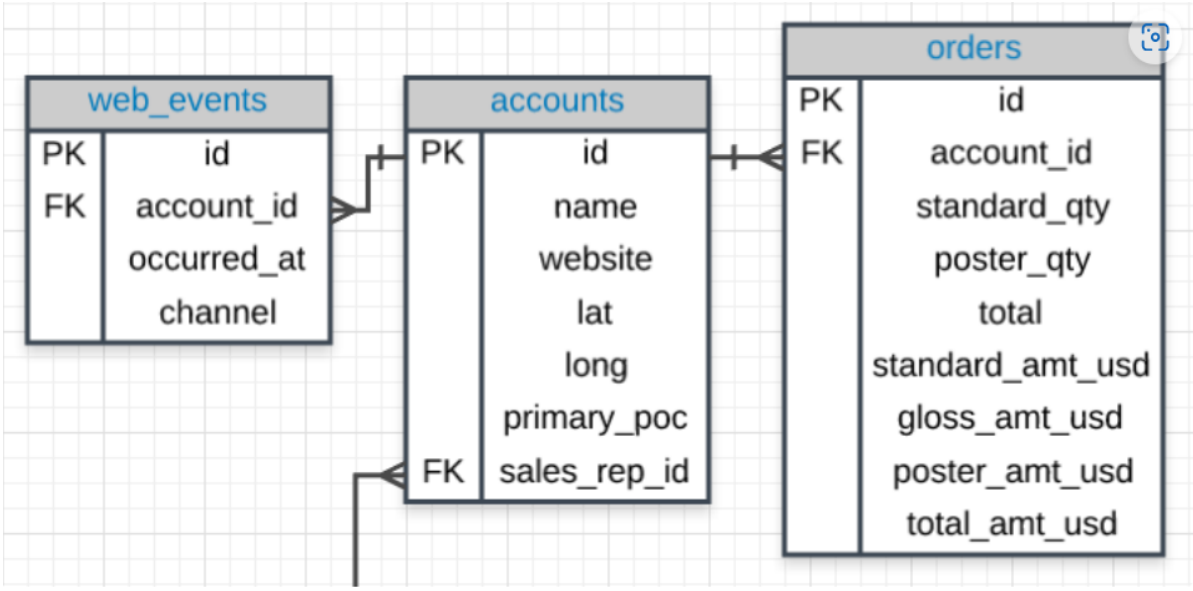
ON accounts.id = orders.accounts_id;



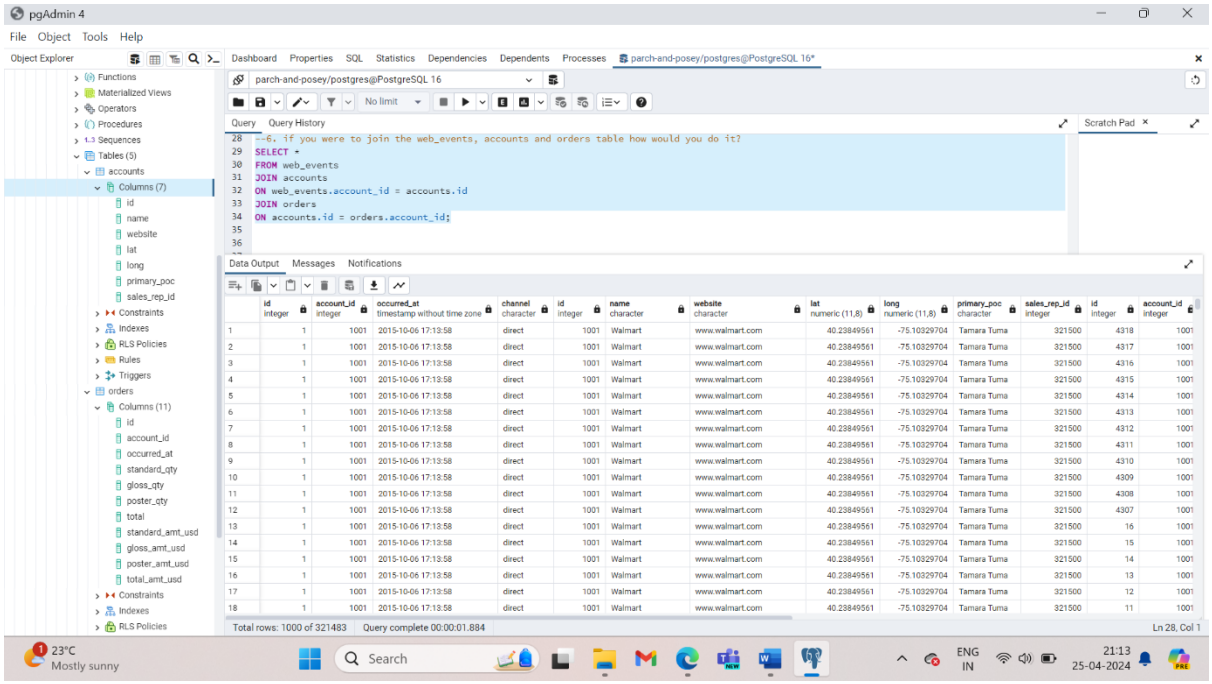
5. Try pulling **standard_qty**, **gloss_qty**, and **poster_qty** from the **orders** table, and the **website** and the **primary_poc** from the **accounts** table.
SELECT orders.standard_qty, orders.gloss_qty, orders.poster_qty,accounts.website, accounts.primary_poc
FROM orders
JOIN accounts
ON orders.account_id = accounts.id;



6. If you were to join the web_events, accounts and orders table, how would you do it?

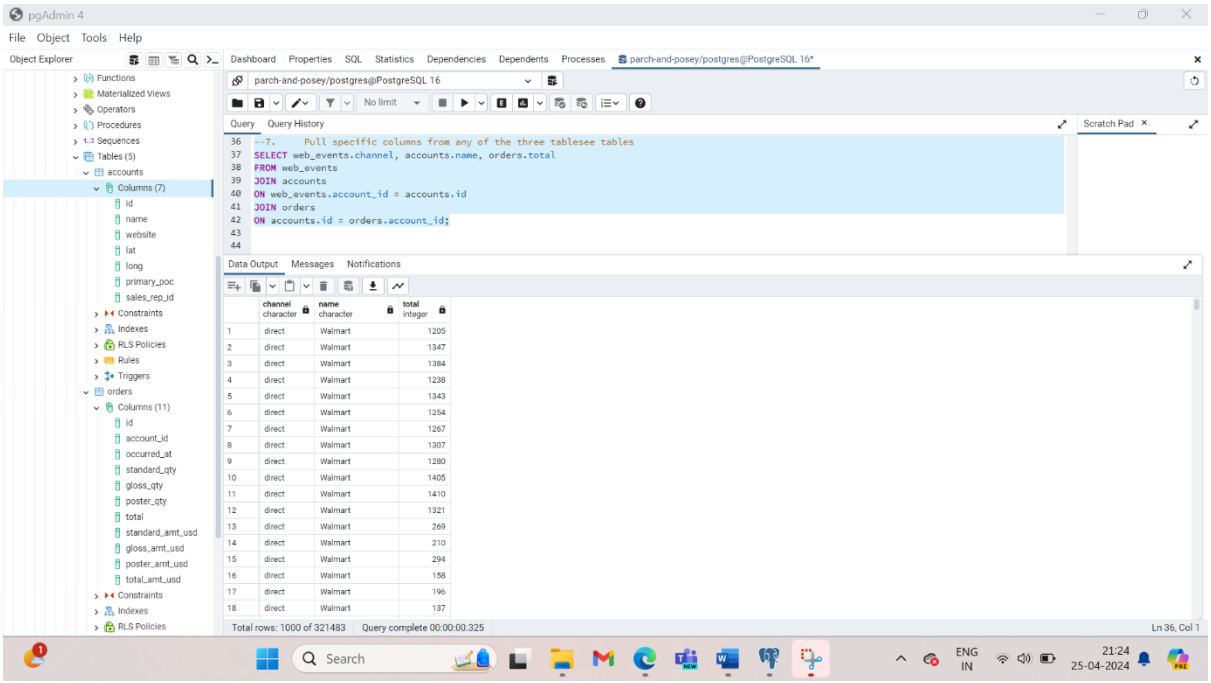



```
SELECT *
FROM web_events
JOIN accounts
ON web_events.account_id = accounts.id
JOIN orders
ON accounts.id = orders.account_id;
```



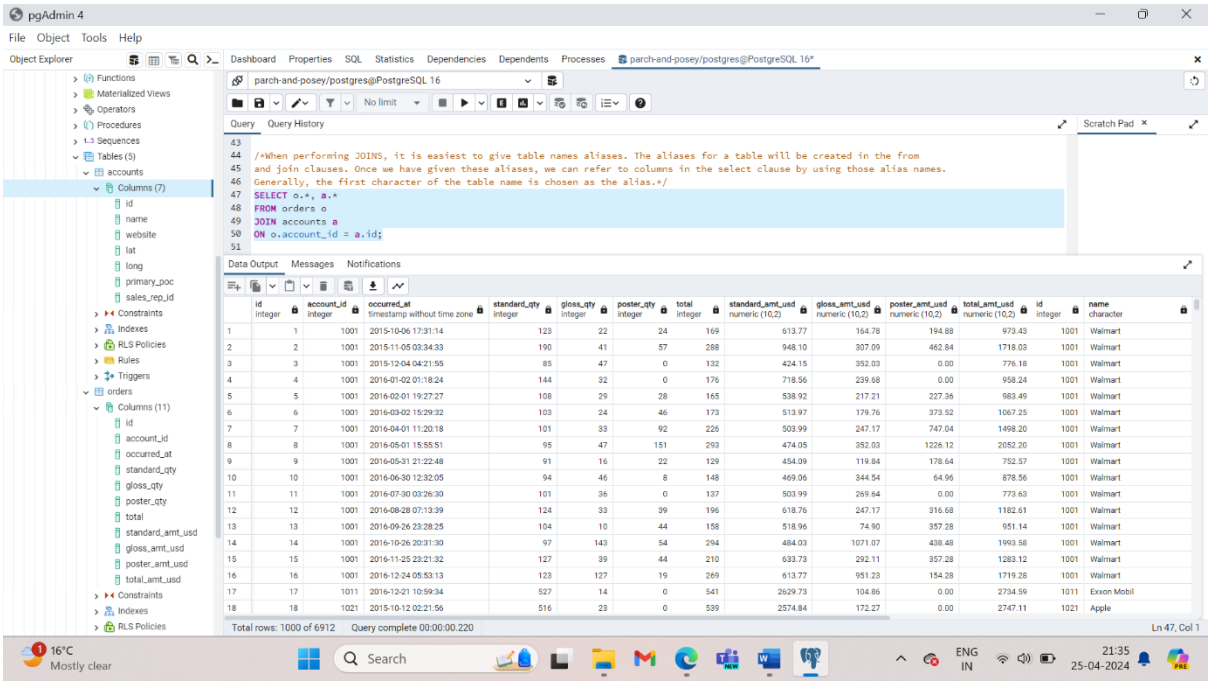
7. Pull specific columns from any of the three tablesee tables

```
SELECT web_events.channel, accounts.name, orders.total
FROM web_events
JOIN accounts
ON web_events.account_id = accounts.id
JOIN orders
ON accounts.id = orders.account_id;
```



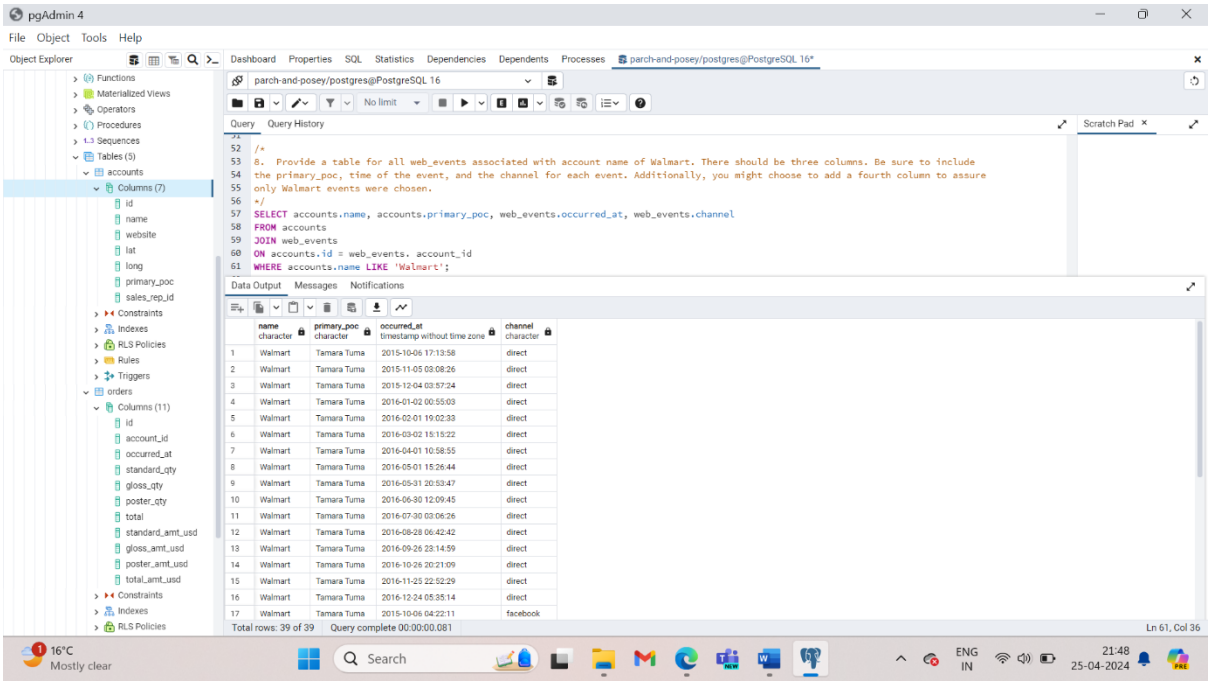
When performing JOINS, it is easiest to give table names aliases. The aliases for a table will be created in the from and join clauses. Once we have given these aliases, we can refer to columns in the select clause by using those alias names. Generally, the first character of the table name is chosen as the alias.

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

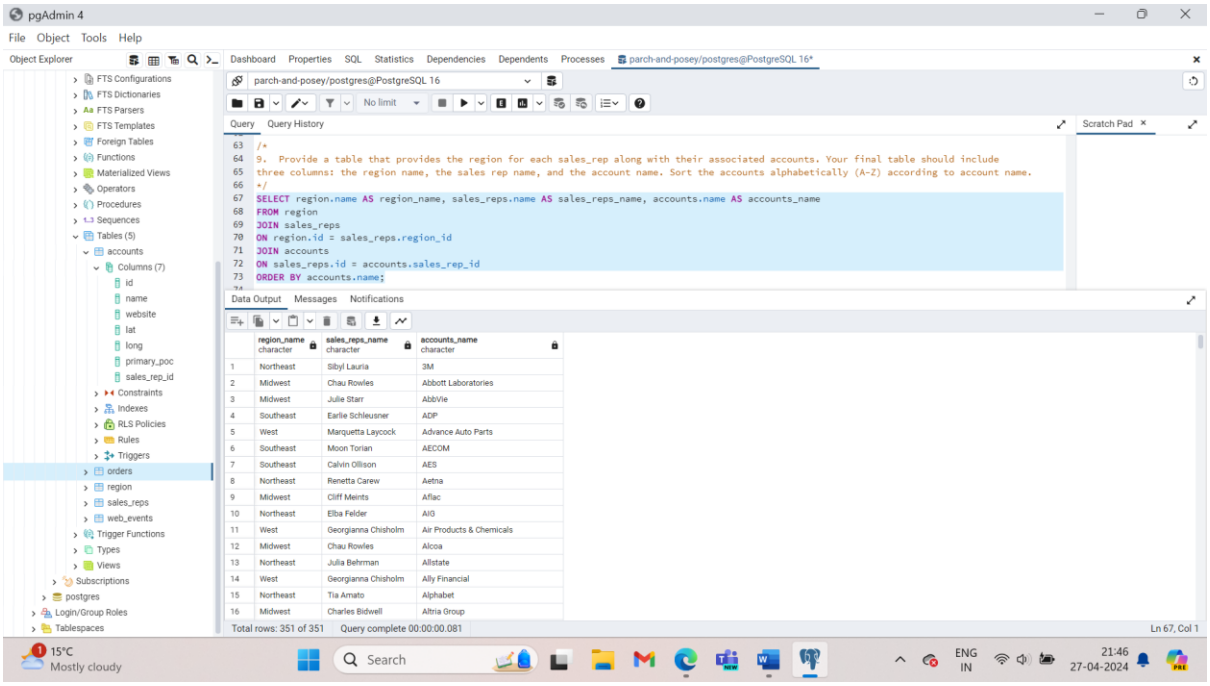


8. Provide a table for all **web_events** associated with account **name** of Walmart. There should be three columns. Be sure to include the primary_poc, time of the event, and the channel for each event. Additionally, you might choose to add a fourth column to assure only Walmart events were chosen.

SELECT accounts.name, accounts.primary_poc, web_events.occurred_at,
web_events.channel
FROM accounts
JOIN web_events
ON accounts.id = web_events. account_id;

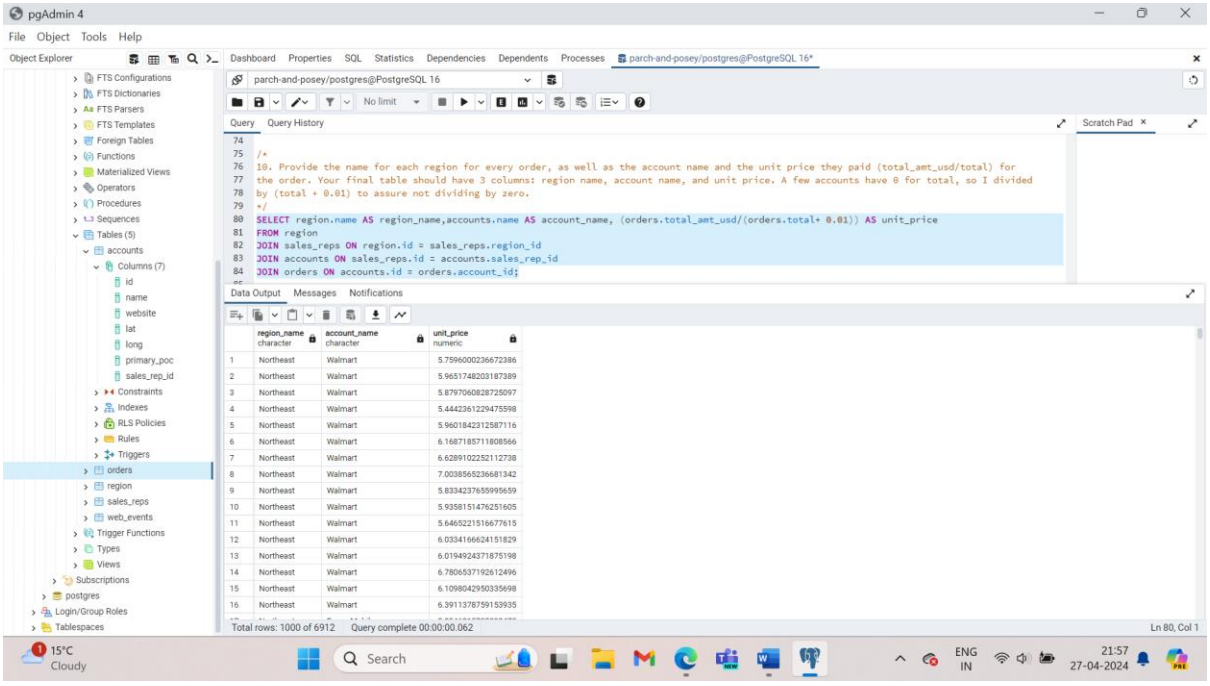


9. Provide a table that provides the **region** for each **sales_rep** along with their associated **accounts**. Your final table should include three columns: the region **name**, the sales rep **name**, and the account **name**. Sort the accounts alphabetically (A-Z) according to account name.
SELECT regions.name, sales_reps.name, accounts.name
FROM regions
JOIN sales_reps
ON regions.id = sales_reps.region_id
JOIN accounts
ON sales_reps.id = accounts.sales_rep_id
ORDER BY accounts.name;



10. 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.

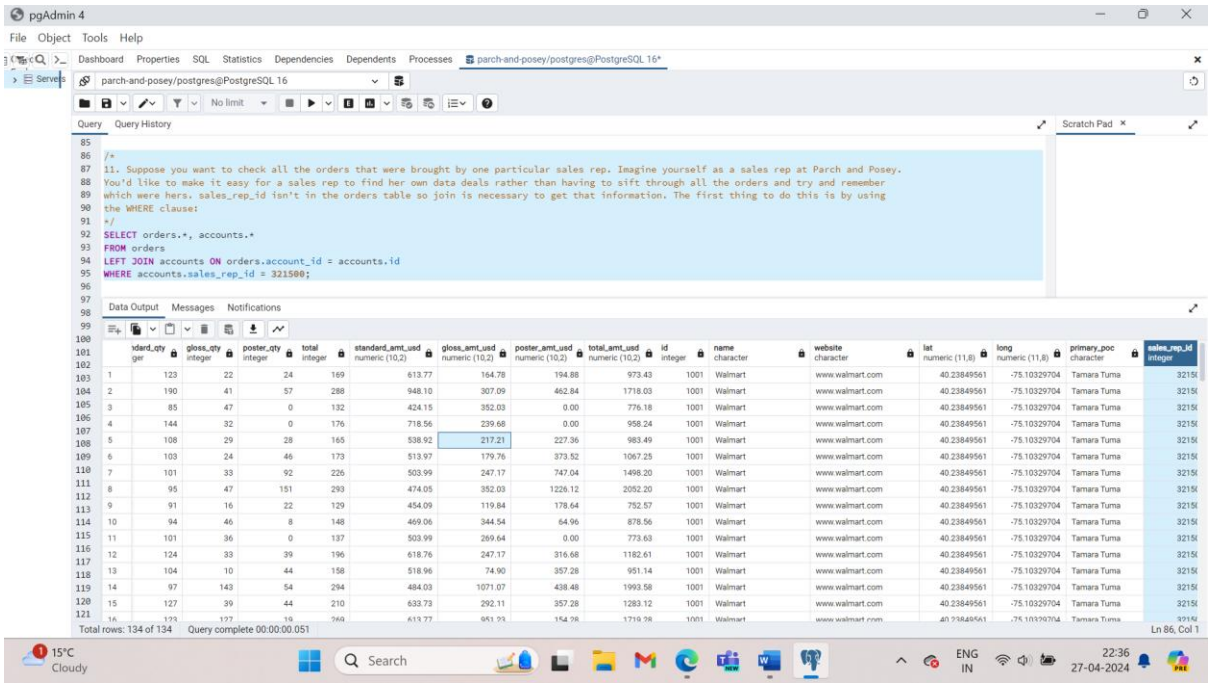
```
SELECT region.name AS region_name,accounts.name AS account_name,
(orders.total_amt_usd/(orders.total+ 0.01)) AS unit_price
FROM region
JOIN sales_reps ON region.id = sales_reps.region_id
JOIN accounts ON sales_reps.id = accounts.sales_rep_id
JOIN orders ON accounts.id = orders.account_id;
```



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.

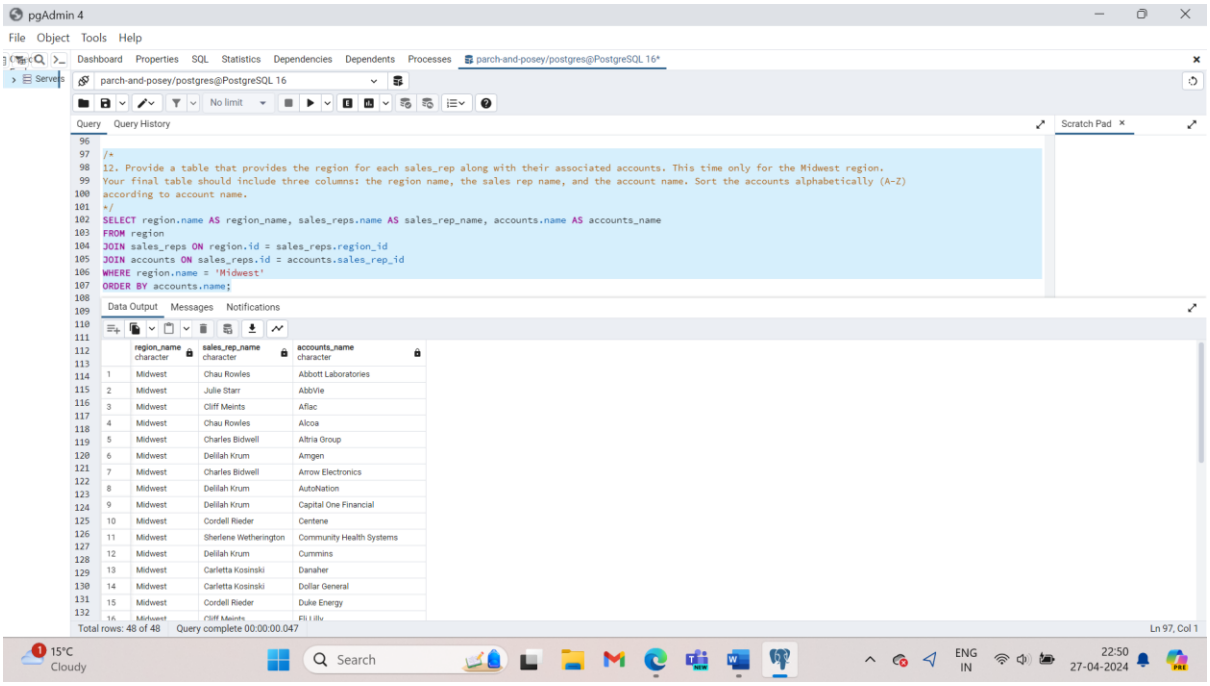
full outer join - will return the inner join result set, as well as any unmatched rows from either of the two tables being joined.

11. Suppose you want to check all the orders that were brought by one particular sales rep. Imagine yourself as a sales rep at Parch and Posey. You'd like to make it easy for a sales rep to find her own data deals rather than having to sift through all the orders and try and remember which were hers. sales_rep_id isn't in the orders table so join is necessary to get that information. The first thing to do this is by using the WHERE clause:
    SELECT orders.*, accounts.*
    FROM orders
    LEFT JOIN accounts ON orders.account_id = accounts.id
    WHERE accounts.sales_rep_id = 321500;



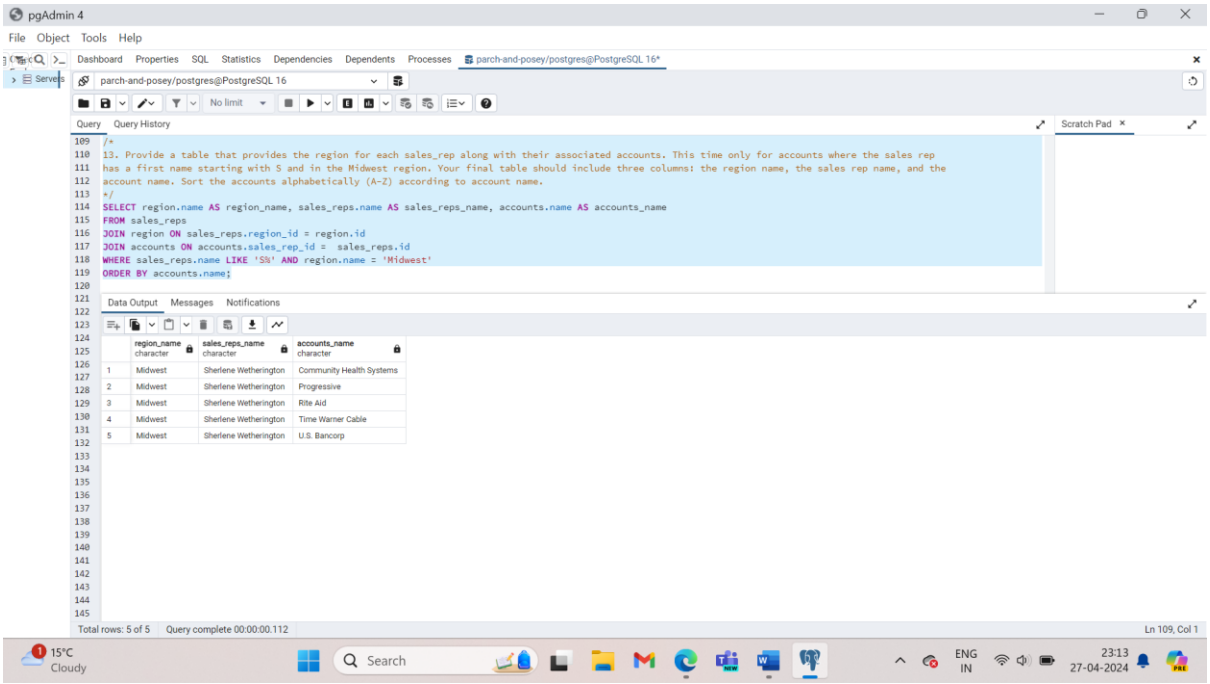
    As you can see the query will have only the orders that have 321500 in the sales_rep_id. But what if we need to keep all the other results order as well (BY changing the WHERE to AND)

12. 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.
    SELECT region.name AS region_name, sales_reps.name AS sales_rep_name, accounts.name AS accounts_name
    FROM region
    JOIN sales_reps ON region.id = sales_reps.region_id
    JOIN accounts ON sales_reps.id = accounts.sales_rep_id
    WHERE region.name = 'Midwest'
    ORDER BY accounts.name;



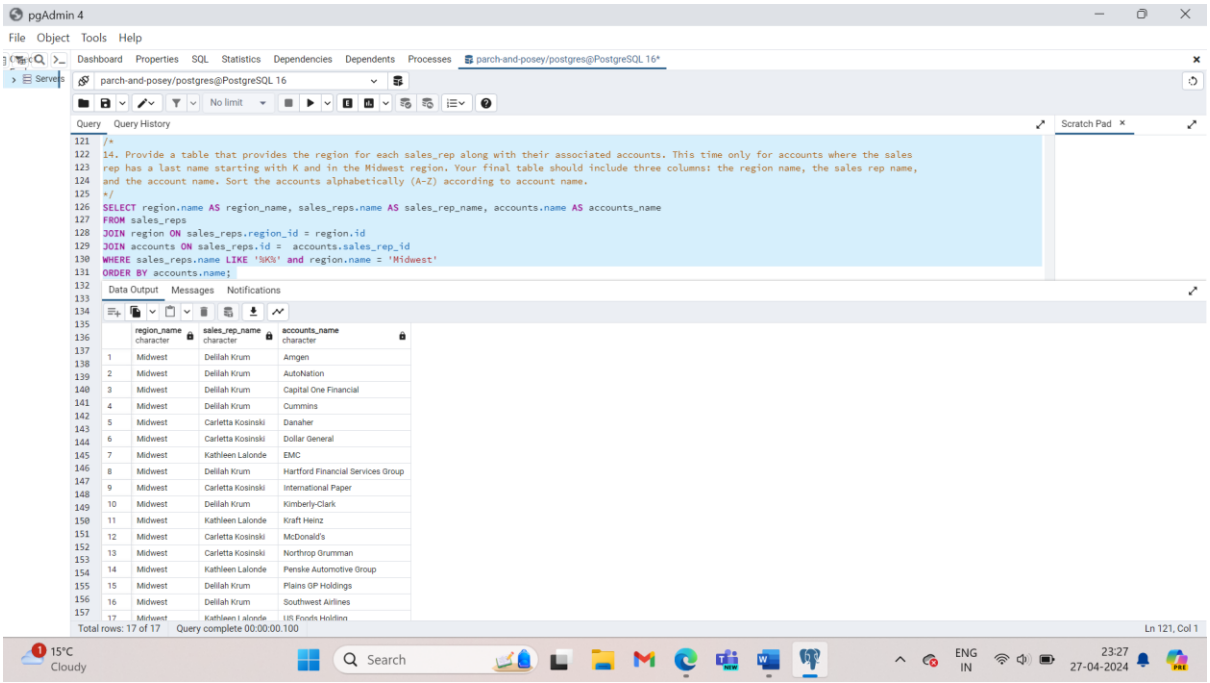
13. Provide a table that provides the **region** for each **sales_rep** along with their associated **accounts**. This time only for accounts where the sales rep has a first name starting with S and in the Midwest region. Your final table should include three columns: the region **name**, the sales rep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

```
SELECT region.name AS region_name, sales_reps.name AS sales_reps_name, accounts.name AS accounts_name
FROM sales_reps
JOIN region ON sales_reps.region_id = region.id
JOIN accounts ON accounts.sales_rep_id =  sales_reps.id
WHERE sales_reps.name LIKE 'S%' AND region.name = 'Midwest'
ORDER BY accounts.name;
```



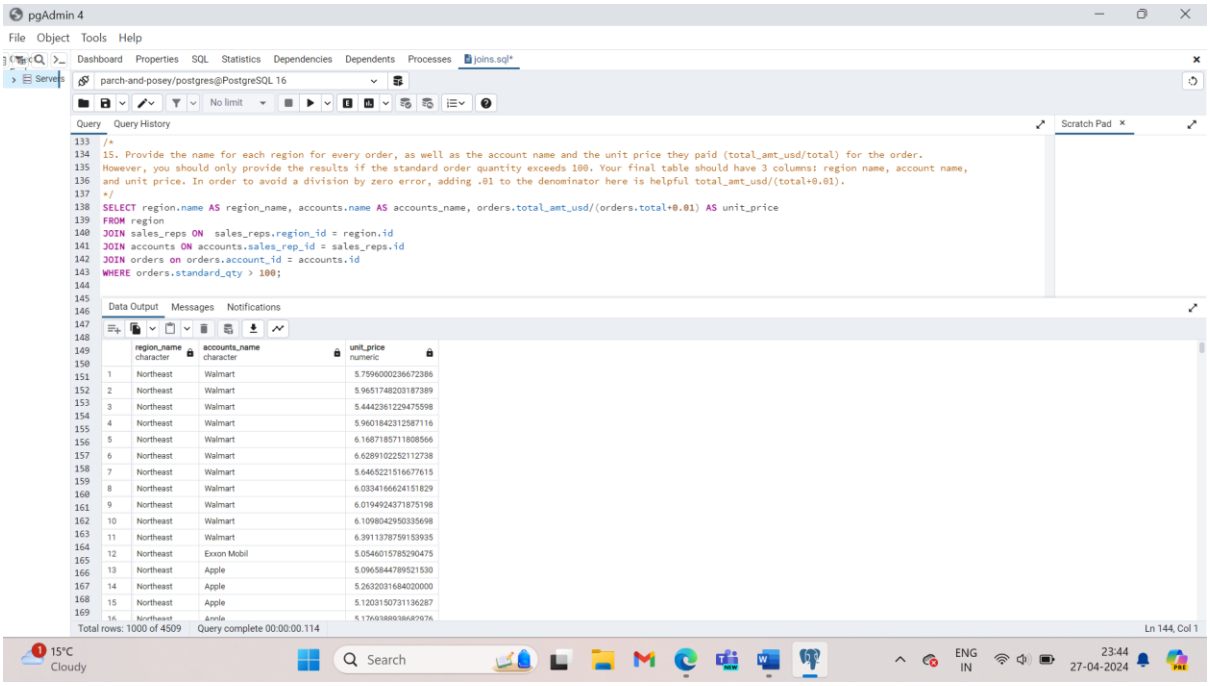
14. Provide a table that provides the **region** for each **sales_rep** along with their associated **accounts**. This time only for accounts where the sales rep has a **last** name starting with K and in the Midwest region. Your final table should include three columns: the region **name**, the sales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

```
SELECT region.name, sales_reps.name, accounts.name
FROM sales_reps
JOIN region ON sales_reps.region_id = region.id
JOIN accounts ON sales_reps.id =  accounts.sales_rep_id
WHERE sales_reps.name LIKE '%K%' and region.name = 'Midwest'
ORDER BY accounts.name;
```



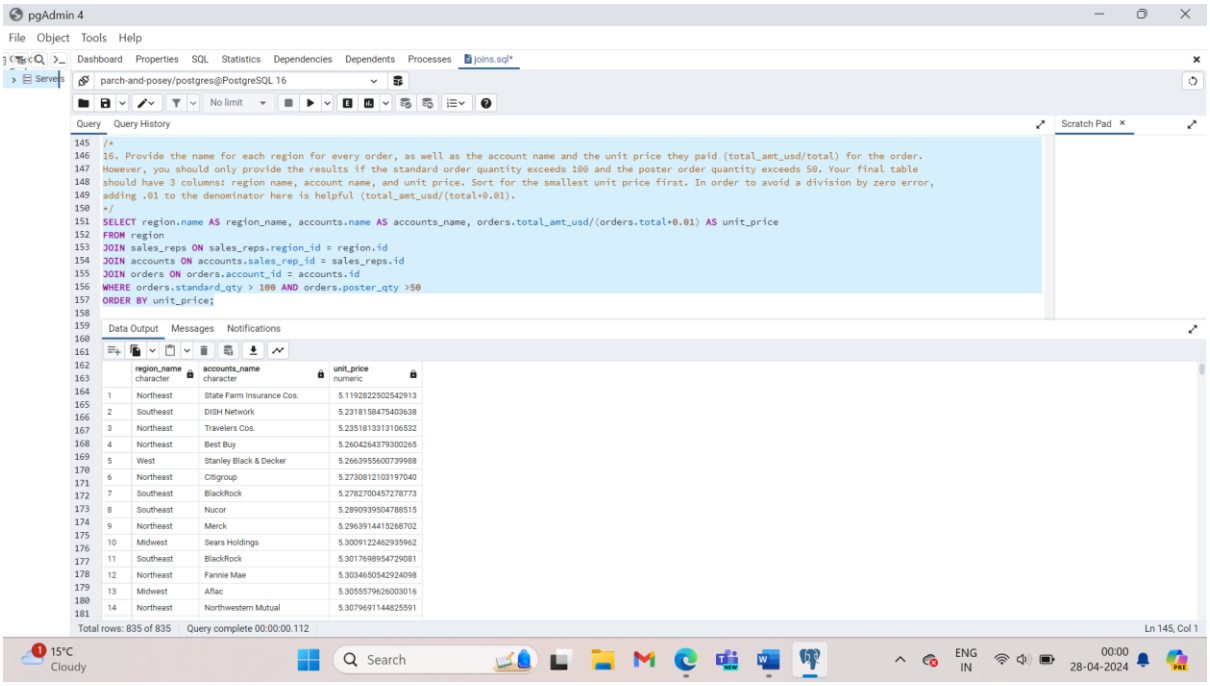
15. Provide the **name** for each region for every **order**, as well as the account **name** and the **unit price** they paid (total_amt_usd/total) for the order. However, you should only provide the results if the **standard order quantity** exceeds 100. Your final table should have 3 columns: **region 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).

```
SELECT region.name AS region_name, accounts.name AS accounts_name,
orders.total_amt_usd/(orders.total+0.01) AS unit_price
FROM region
JOIN sales_reps ON  sales_reps.region_id = region.id
JOIN accounts ON accounts.sales_rep_id = sales_reps.id
JOIN orders on orders.account_id = accounts.id
WHERE orders.standard_qty > 100;
```



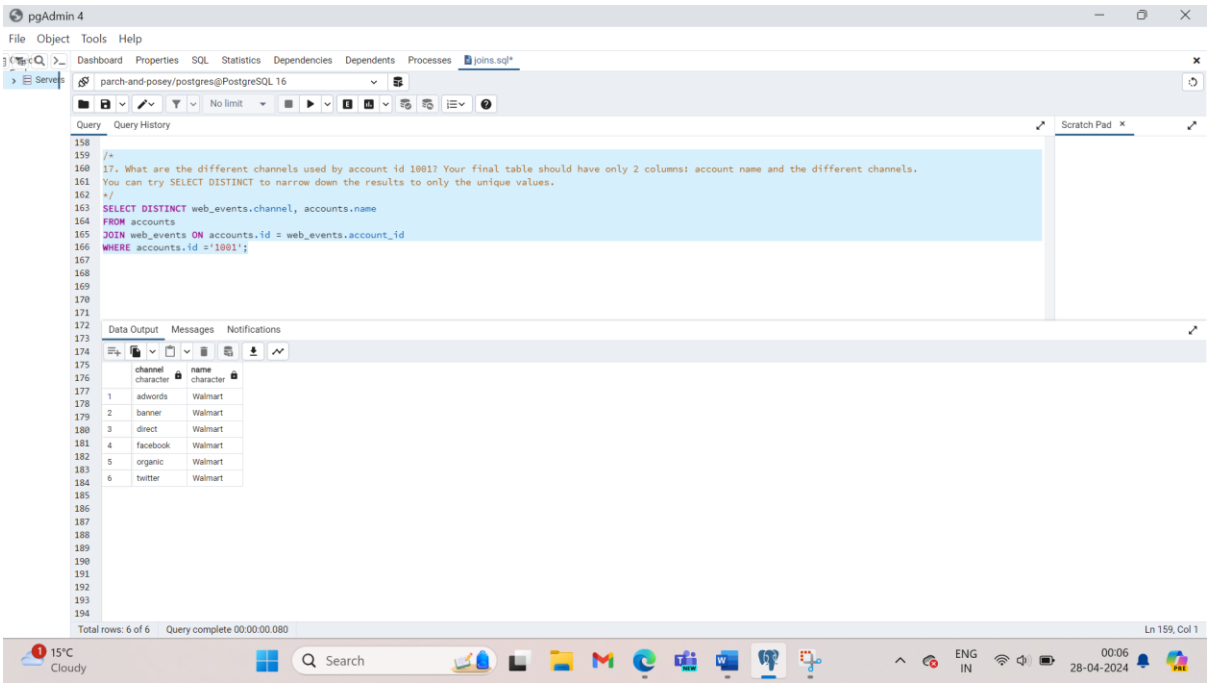
16. Provide the **name** for each region for every **order**, as well as the account **name** and the **unit price** they paid (total_amt_usd/total) for the order. However, you should only provide the results if the **standard order quantity** exceeds 100 and the **poster order quantity** exceeds 50. 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).

```
SELECT region.name AS region_name, accounts.name AS accounts_name,
orders.total_amt_usd/(orders.total+0.01) AS unit_price
FROM region
JOIN sales_reps ON sales_reps.region_id = region.id
JOIN accounts ON accounts.sales_rep_id = sales_reps.id
JOIN orders ON orders.account_id = accounts.id
WHERE orders.standard_qty > 100 AND orders.poster_qty >50
ORDER BY unit_price;
```



17. 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.

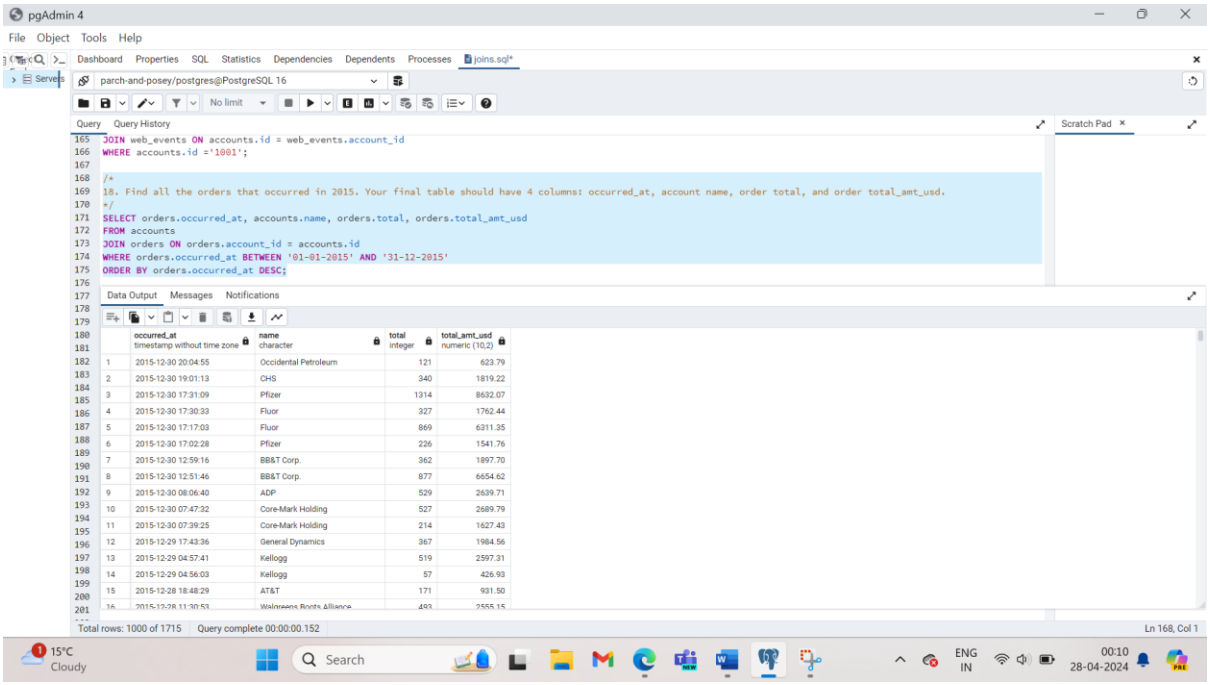
```
SELECT DISTINCT web_events.channel, accounts.name
FROM accounts
JOIN web_events ON accounts.id = web_events.account_id
WHERE accounts.id =’1001’;
```



18. 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.

```
SELECT orders.occurred_at, accounts.name, orders.total, orders.total_amt_usd
FROM accounts
JOIN orders ON orders.account_id = accounts.id
```

WHERE orders.occurred_at BETWEEN '01-01-2015' AND '31-12-2015'
ORDER BY orders.occurred_at DESC;



RECAP

Primary keys- are unique for every row in a table. These are generally the first column in our database

Foreign keys – are the primary keys appearing in another table, which allows the rows to be non-unique.

JOIN – an INNER JOIN that only pulls the data that exists in both tables.

LEFT JOIN- pulls all the data that exists in both tables, as well as all the rows from the table in the FROM even if they do not exist in the JOIN statement.

RIGHT JOIN- pulls all the data that exists in both tables, as well as all the rows from the table in the join even if they do not exist in the FROM statement.

ALIAS – We can alias tables and olumns 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 assume that the column headings are informative of the data in your table.

UNION, UNION ALL, CROSS JOIN, SELF JOIN