**SQL JOINS**

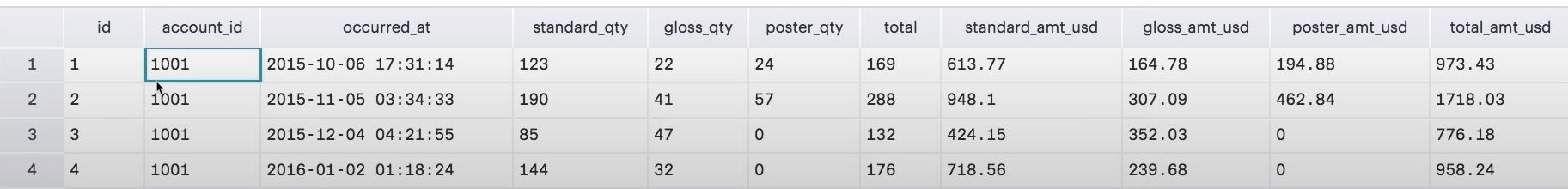
The real power of SQL comes from working with data across multiple tables at once.

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.



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.**

A screenshot of a computer

Description automatically generated

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:

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?

**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.**

**Introduction to joins**

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;

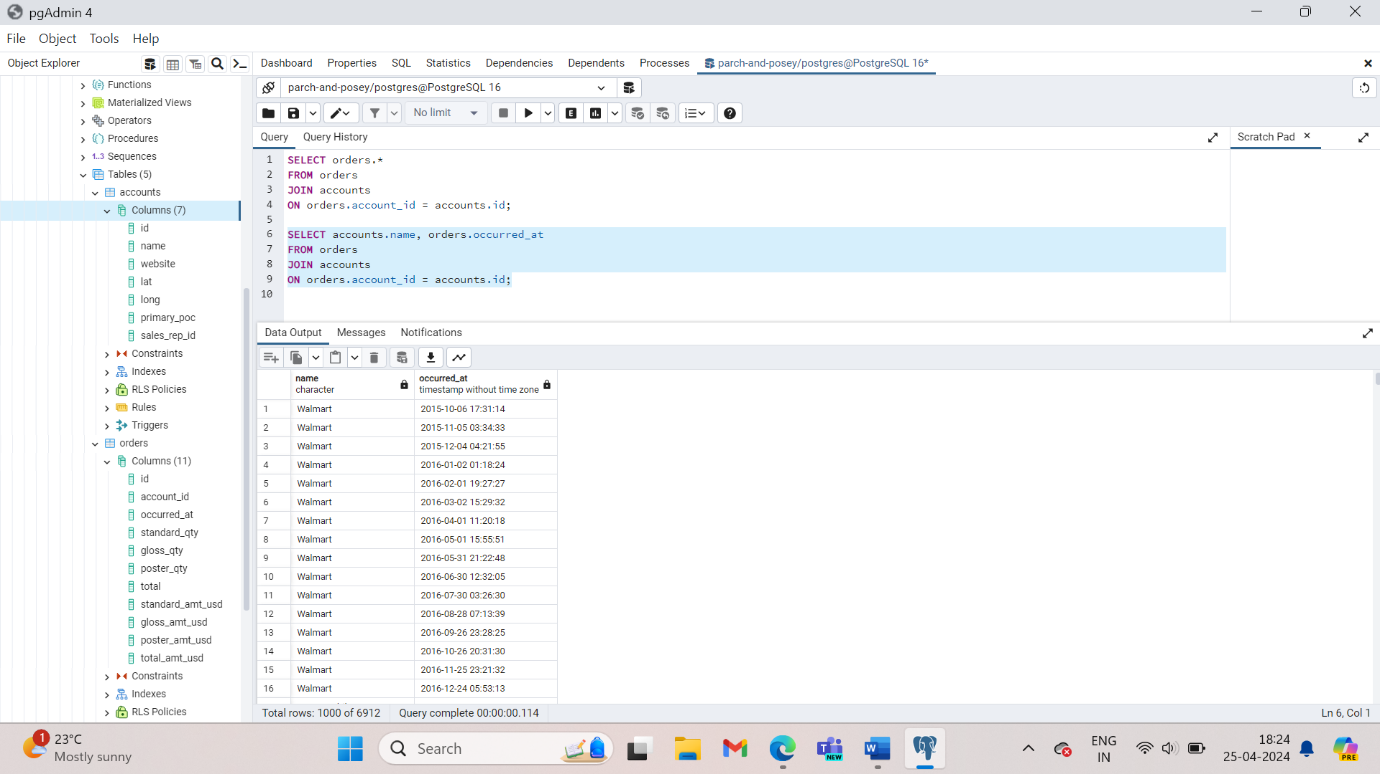
1. If we want to pull only the **account name** and the dates in which that account placed an order, but none of the other columns, we can do this with the following query:

SELECT accounts.name, orders.occurred\_at

FROM orders

JOIN accounts

ON orders.account\_id = accounts.id;

****

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

**SELECT** \*

**FROM** orders

**JOIN** accounts

**ON** orders.account\_id = accounts.id;



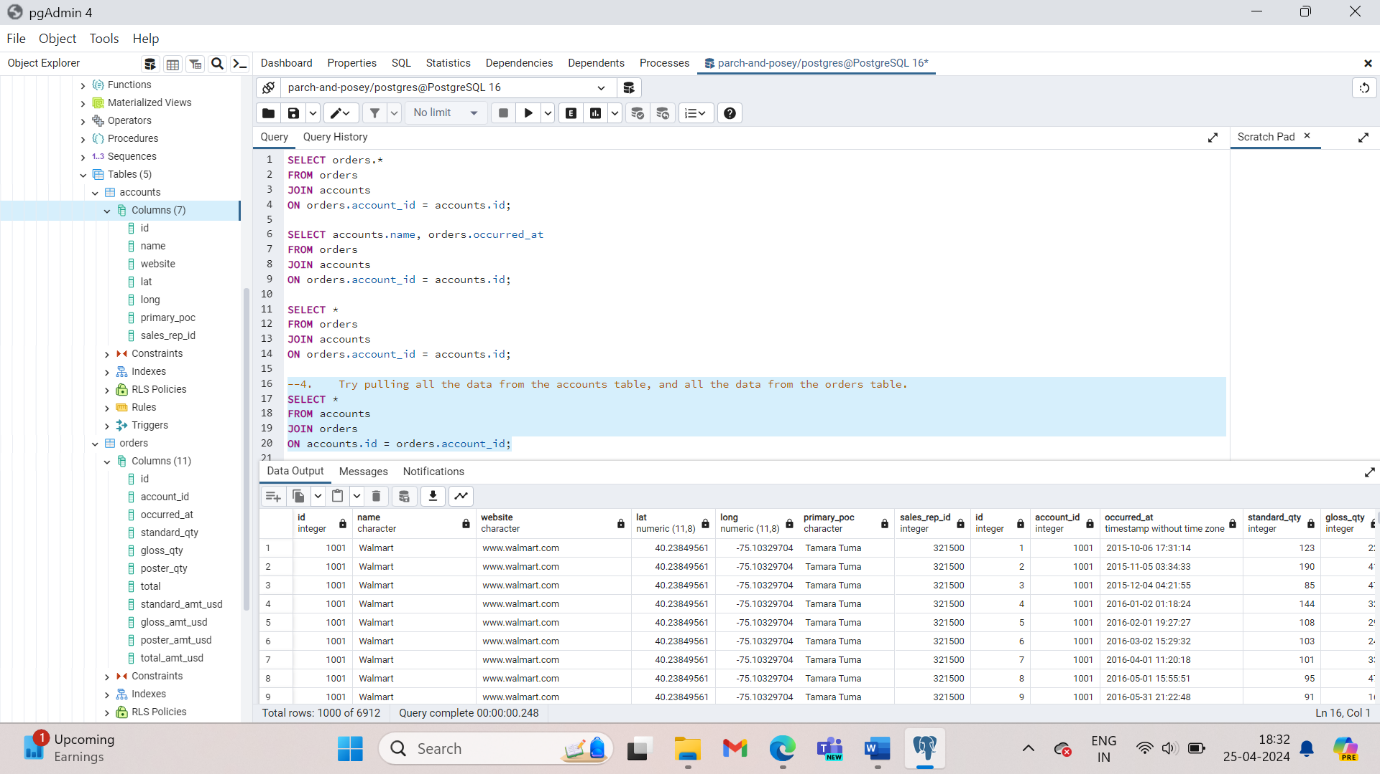
1. Try pulling all the data from the **accounts** table, and all the data from the **orders** table.

SELECT \*

FROM accounts

JOIN orders

ON accounts.id = orders.accounts\_id;



1. 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;

A screenshot of a computer

Description automatically generated

1. If you were to join the web\_events, accounts and orders table, how would you do it?

A diagram of a computer

Description automatically generated

SELECT \*

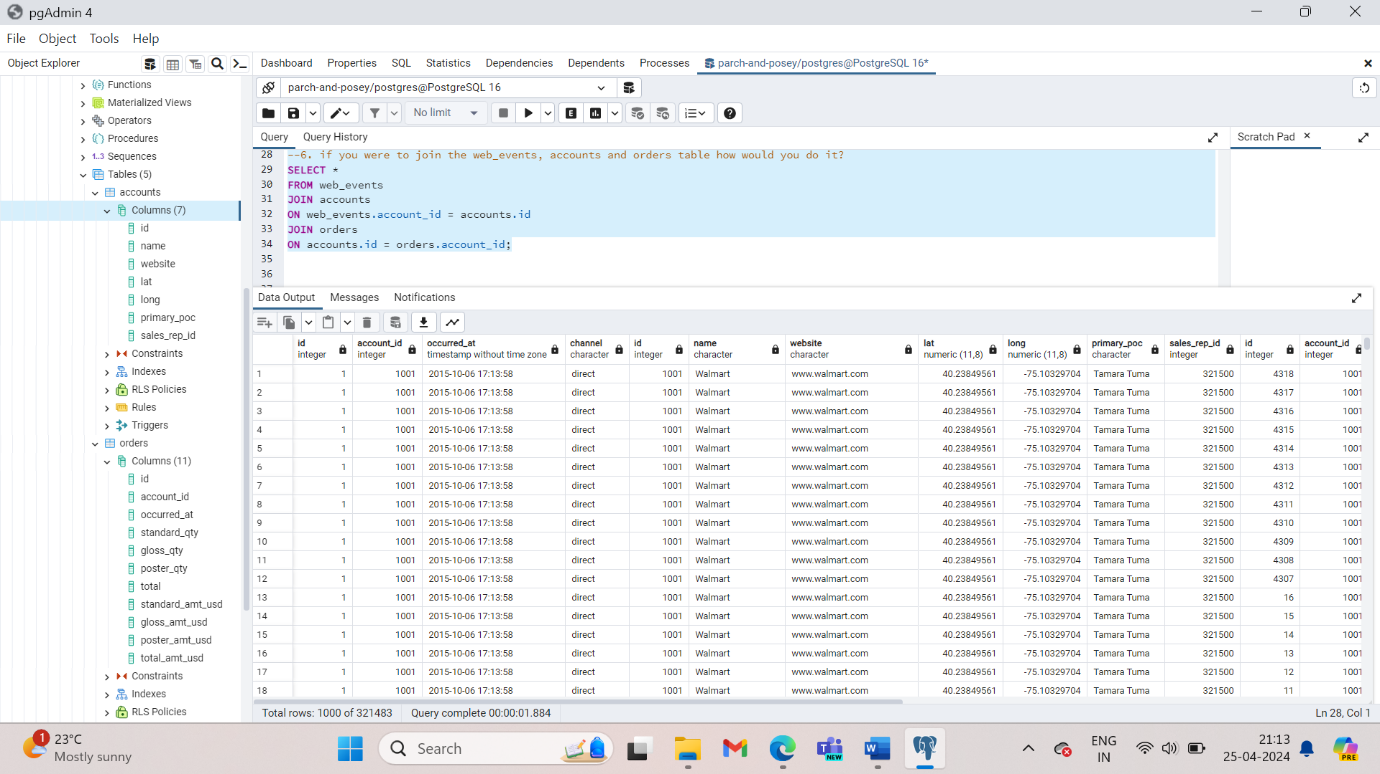
FROM web\_events

JOIN accounts

ON web\_events.account\_id = accounts.id

JOIN orders

ON accounts.id = orders.account\_id;



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

SELECT web\_events.channel, accounts.name, orders.total

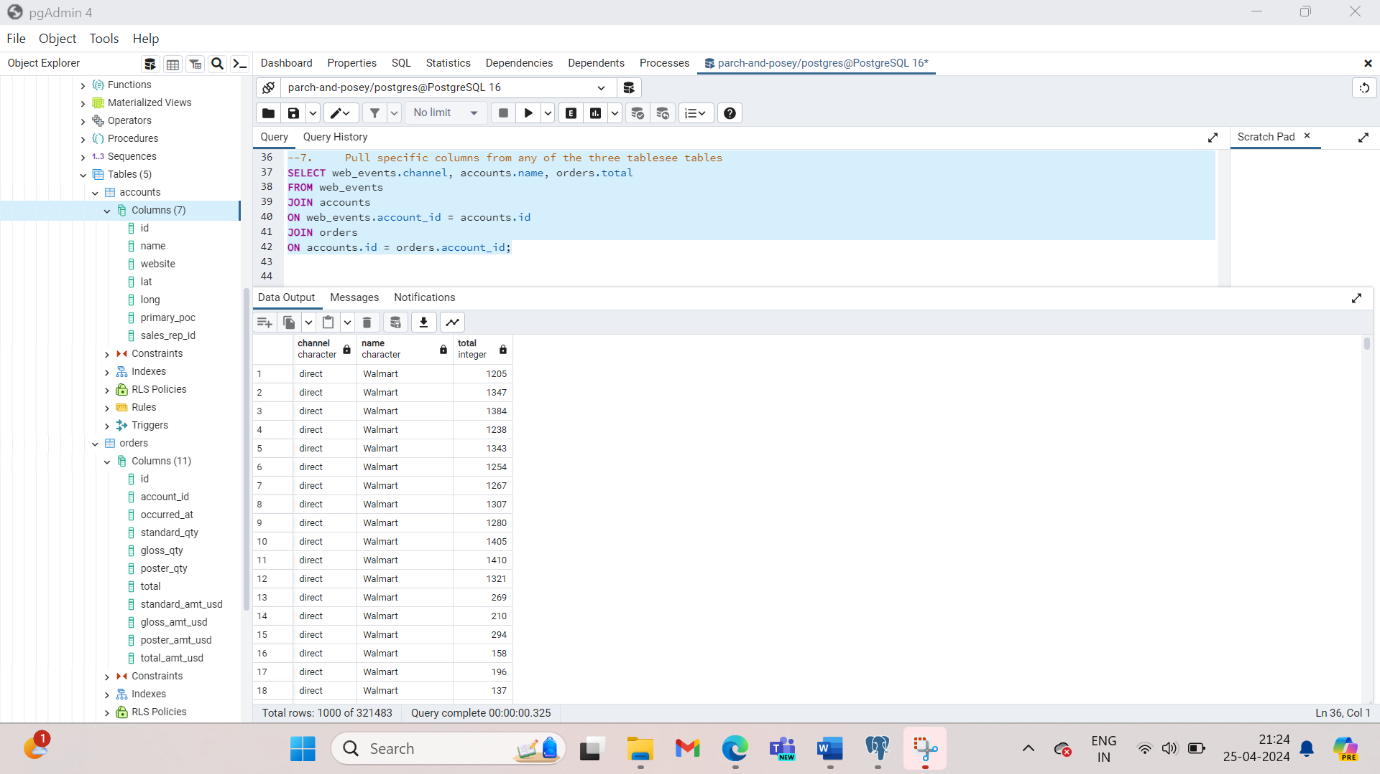
FROM web\_events

JOIN accounts

ON web\_events.account\_id = accounts.id

JOIN orders

ON accounts.id = orders.account\_id;



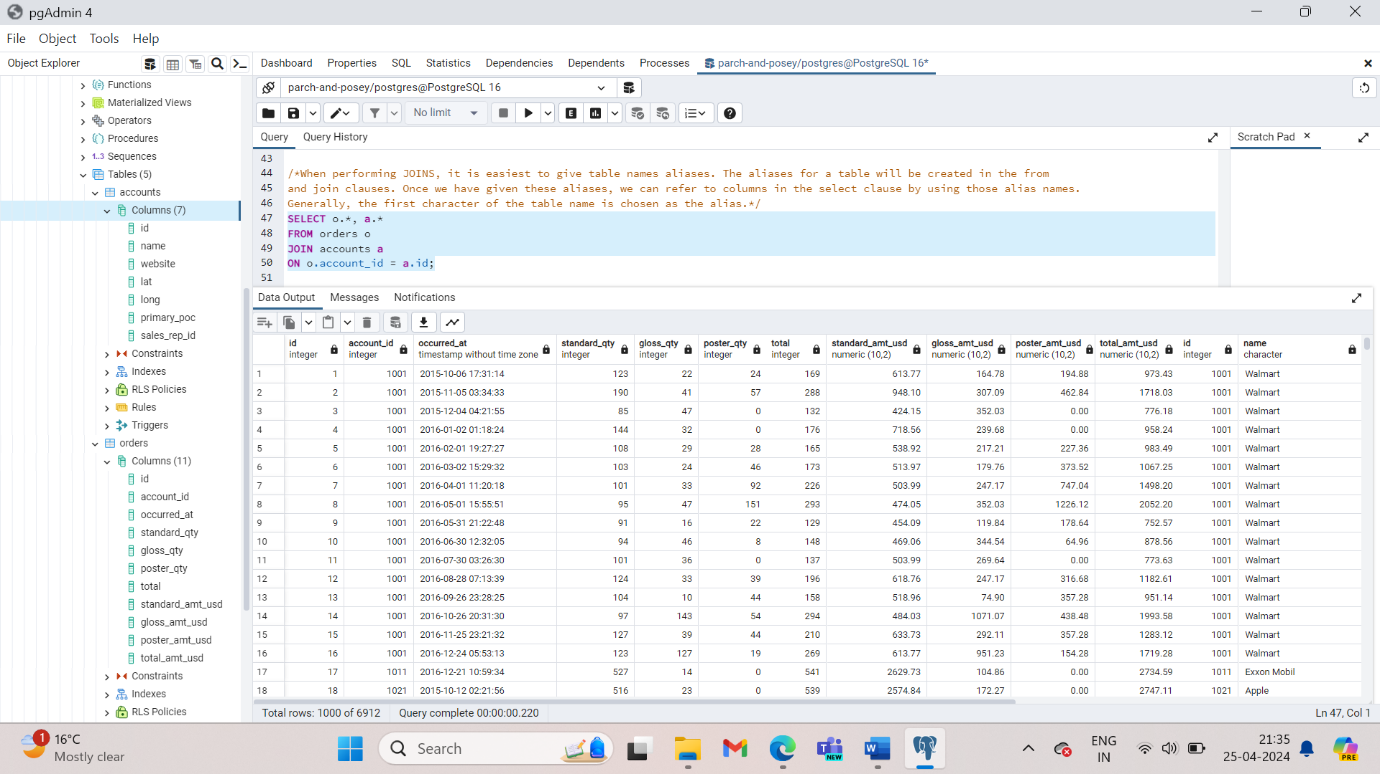
When performing JOINS, it is easiest to give table names aliases. The aliases for a table will be created in the from and join clauses. Once we have given these aliases, we can refer to columns in the select clause by using those alias names. Generally, the first character of the table name is chosen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

SELECT o.\*, a.\*

FROM orders o

JOIN accounts a

ON o.account\_id = a.id;



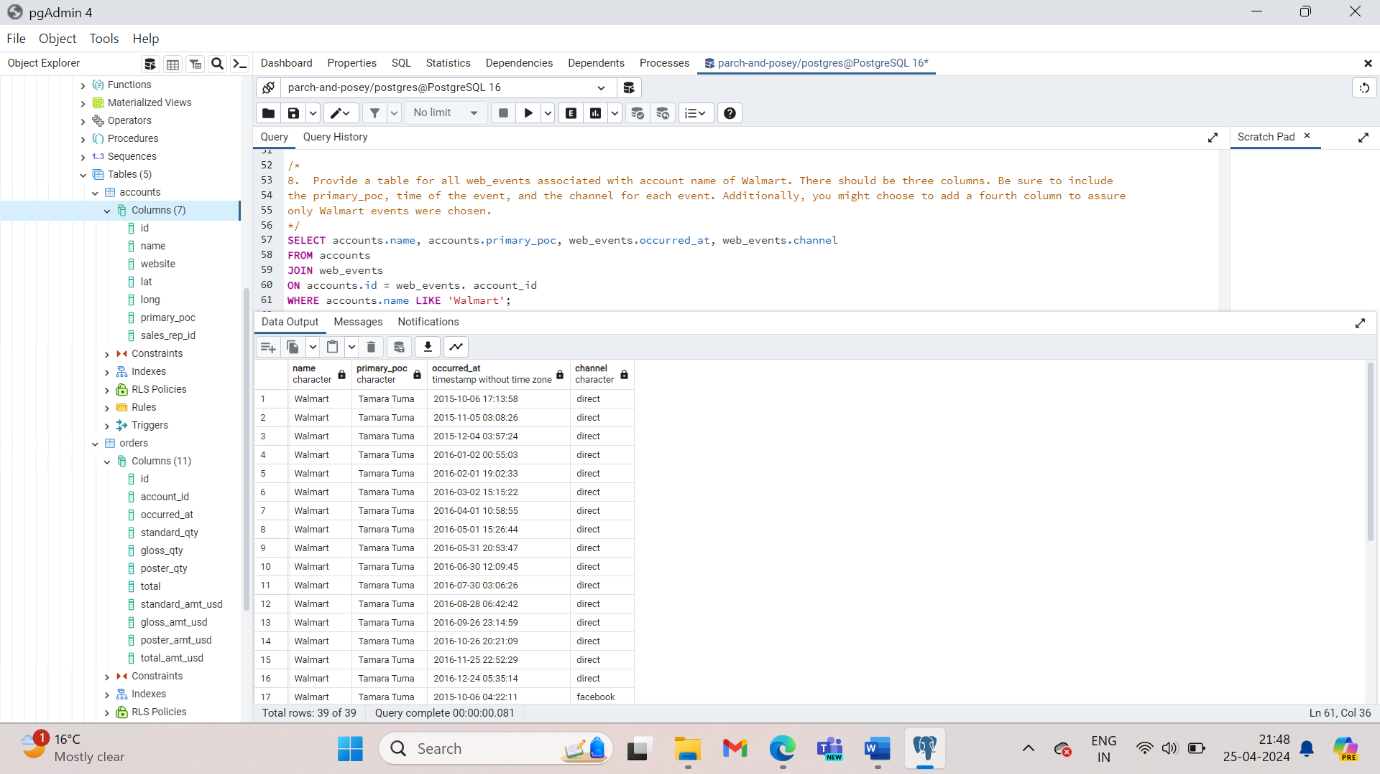
1. Provide a table for all **web\_events** associated with account **name** of Walmart. There should be three columns. Be sure to include the primary\_poc, time of the event, and the channel for each event. Additionally, you might choose to add a fourth column to assure only Walmart events were chosen.

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;



1. 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;

A screenshot of a computer

Description automatically generated

1. 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.

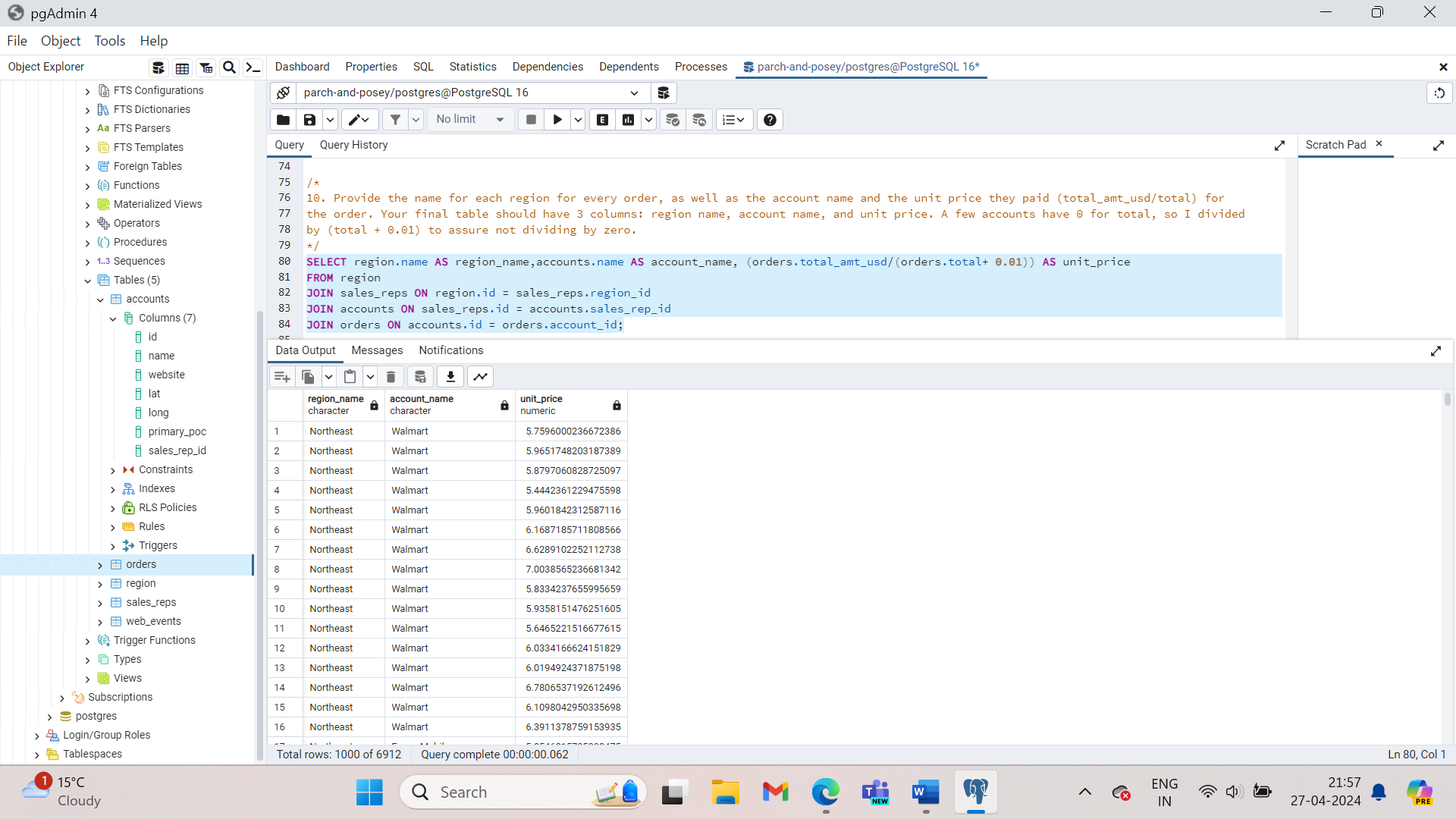
SELECT region.name AS region\_name,accounts.name AS account\_name, (orders.total\_amt\_usd/(orders.total+ 0.01)) AS unit\_price

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.

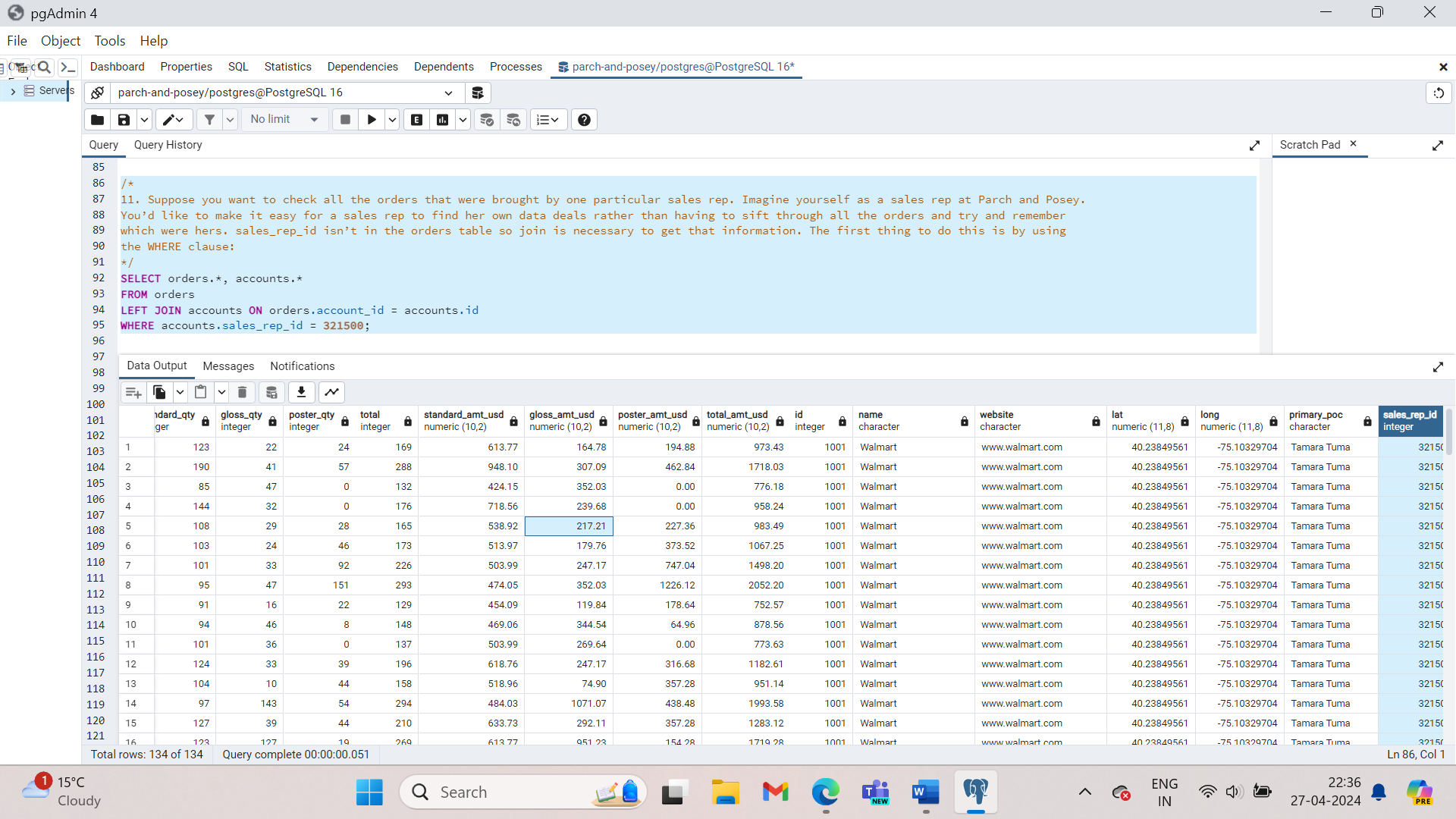
1. Suppose you want to check all the orders that were brought by one particular sales rep. Imagine yourself as a sales rep at Parch and Posey. You’d like to make it easy for a sales rep to find her own data deals rather than having to sift through all the orders and try and remember which were hers. 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)

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.

SELECT region.name AS region\_name, sales\_reps.name AS sales\_rep\_name, accounts.name AS accounts\_name

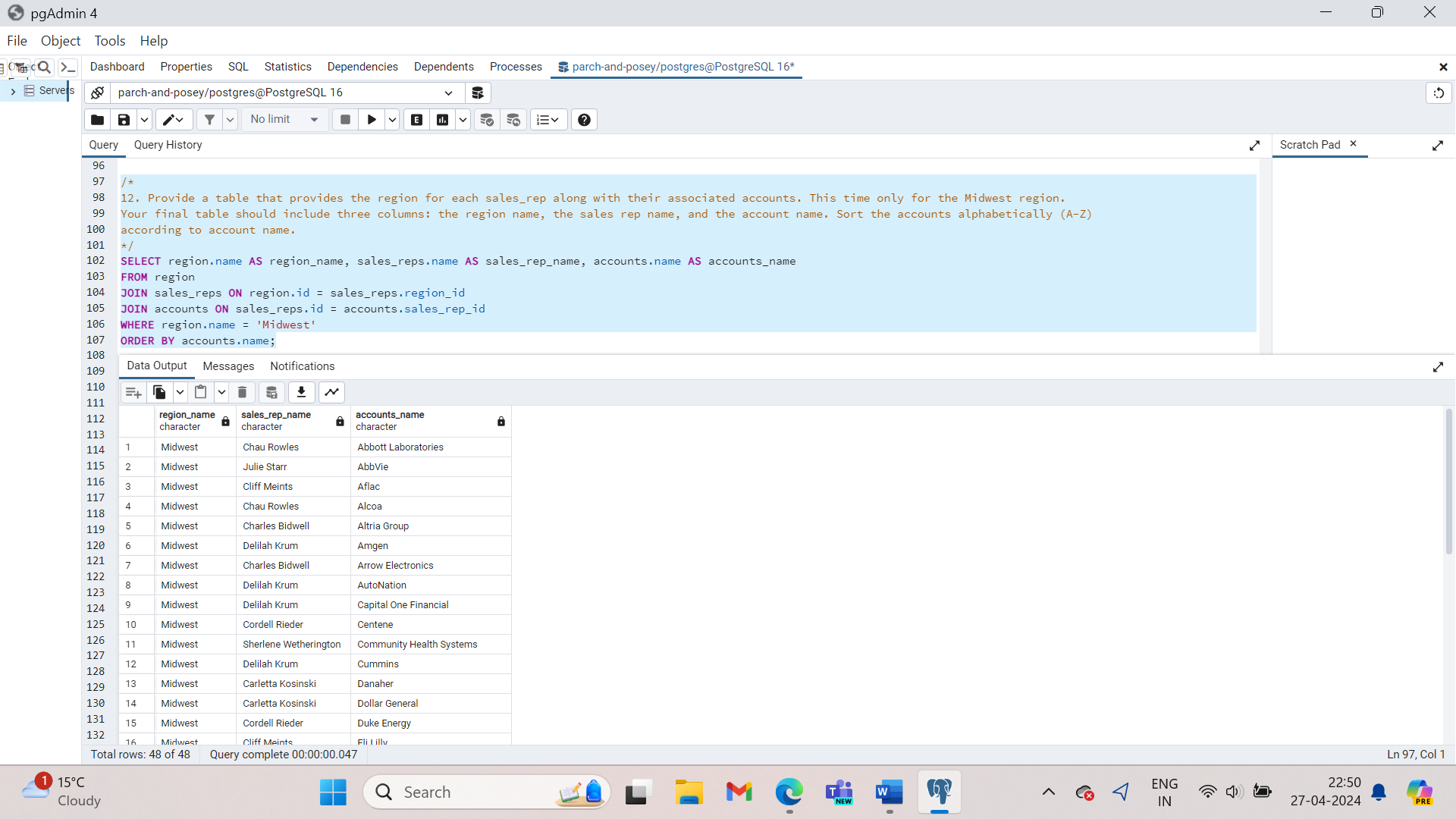
FROM region

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;



1. 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

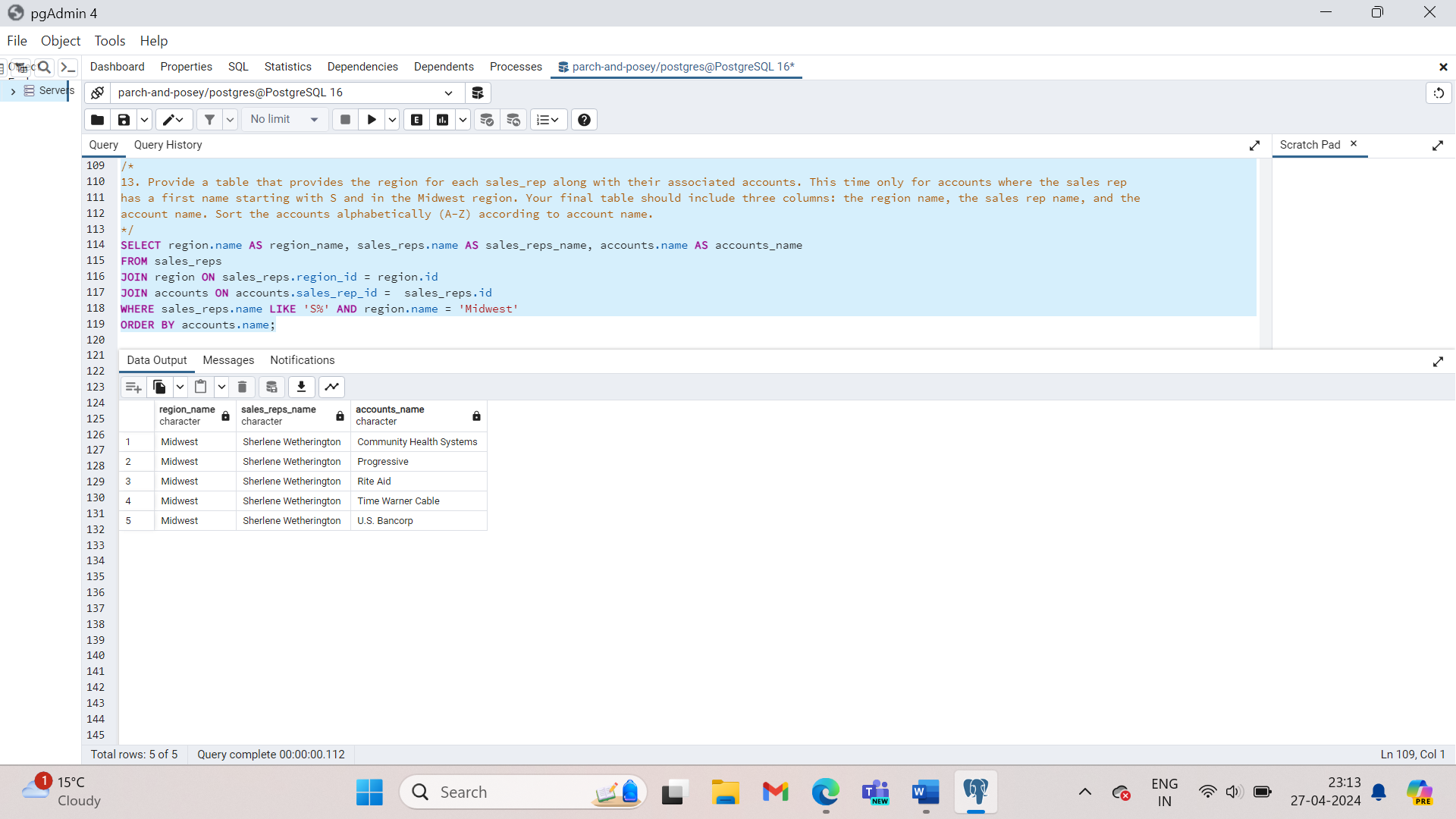
FROM sales\_reps

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;



1. 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

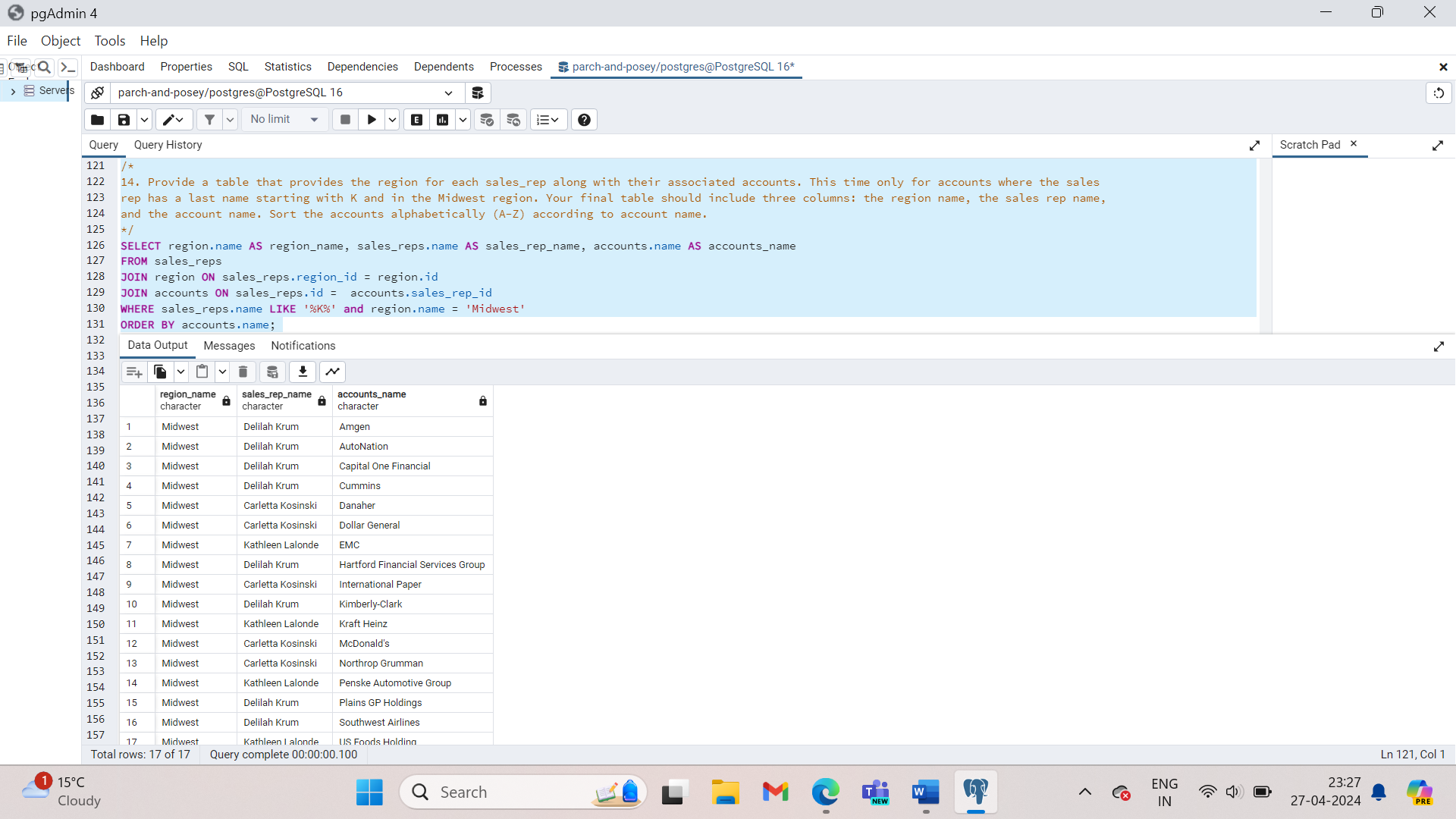
FROM sales\_reps

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;



1. 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

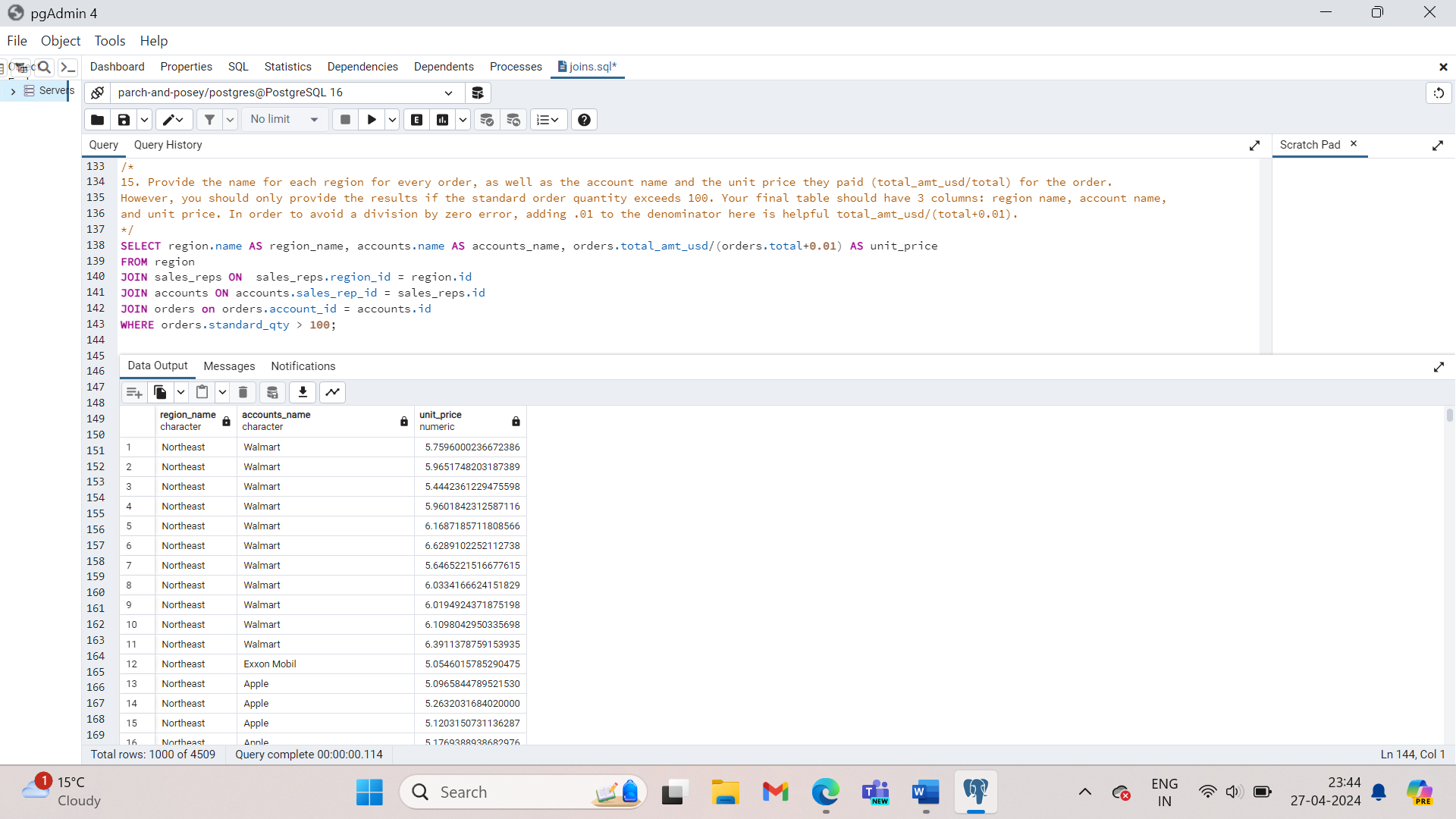
FROM region

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;



1. 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

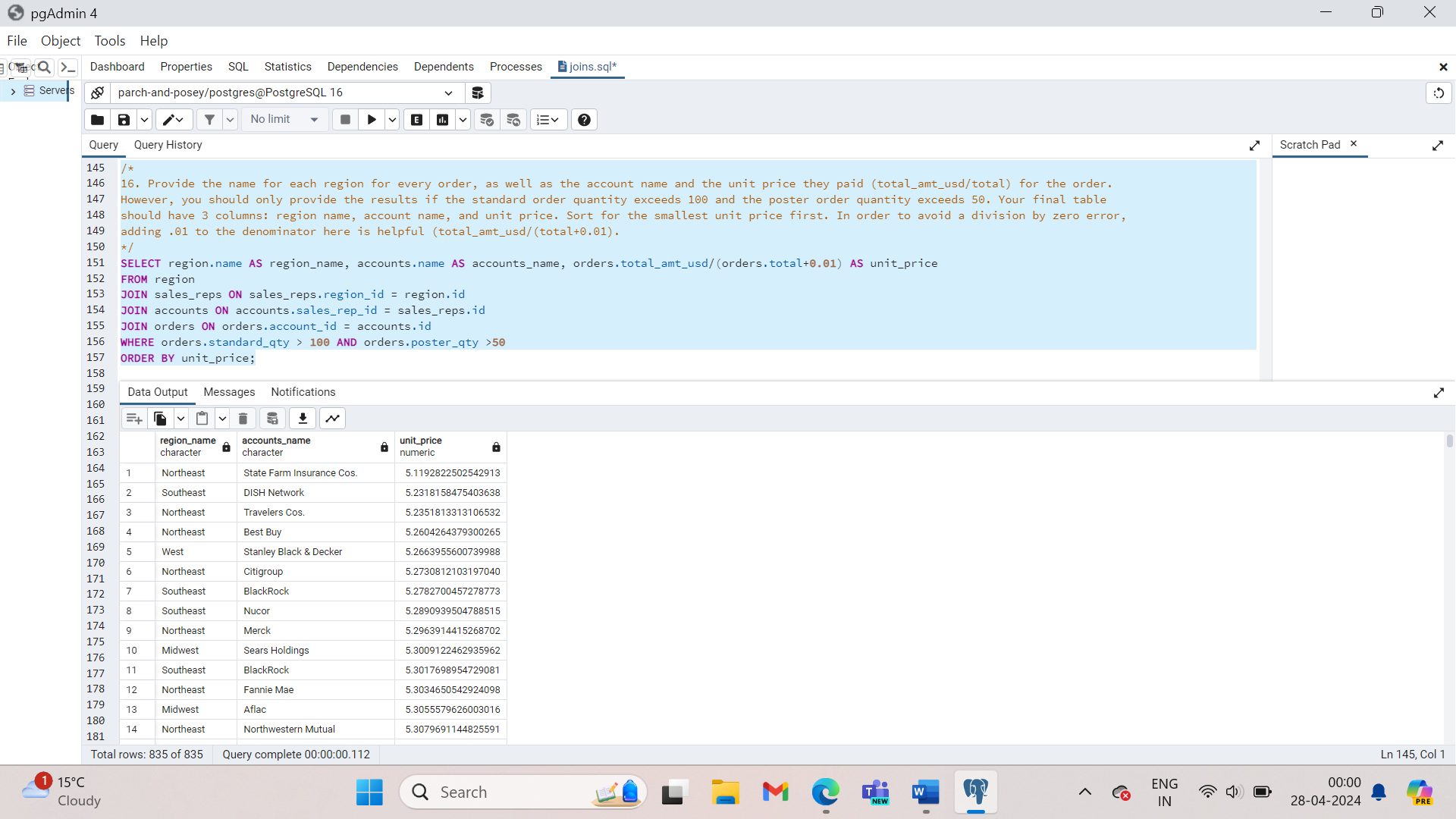
JOIN sales\_reps ON sales\_reps.region\_id = region.id

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;



1. 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’;

A screenshot of a computer

Description automatically generated

1. 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;

A screenshot of a computer

Description automatically generated

**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](https://www.w3schools.com/sql/sql_union.asp), [CROSS JOIN](https://www.w3resource.com/sql/joins/cross-join.php), [SELF JOIN](https://www.w3schools.com/sql/sql_join_self.asp)