Aggregation

Sometimes we need to look at row level data of a salesperson to see how they're performing in each month. But other times row level data is overwhelming, and ultimately less valuable than aggregated data. Counting all the orders in each region in each month, for example, will be 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, the SQL database is great at aggregating data.

In practice, you will find yourself using row level output in the early exploratory work when searching your database to better understand this data. As you begin to get a sense of what the data looks like and begin to look for answers to your questions, aggregates become more helpful.

Nulls are a datatype that specifies where no data exists in SQL. It differs from a zero or a space. And there's good reason. From a business perspective, for Parch and Posey, a zero means that no paper was sold, which could mean that a sale was attempted but not made. A null could mean that no sale was even attempted. And that is a pretty meaningful difference.

Notice that NULL is different than zero – these are cells where data does not exist. When identifying NULLs 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.

There are two common ways in which you are likely to encounter NULLs:

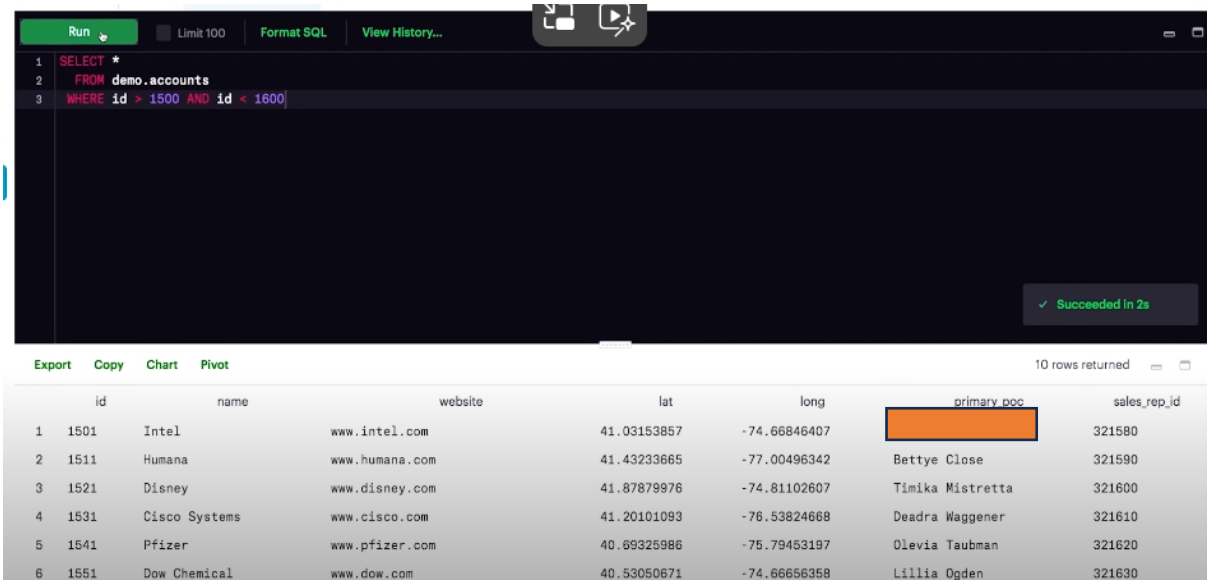
Nulls frequently occur when performing left or right join.

Nulls can also occur from simply missing data in our database.

SELECT *

FROM accounts

WHERE id > 1500 AND id < 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
5	1541	Pfizer	www.pfizer.com	40.69325986	-75.79453197	Olevia Taubman	321620
6	1551	Dow Chemical	www.dow.com	40.53050671	-74.66656358	Lillia Ogden	321630

Here we can see that the primary POC, is blank for the Intel account. This could simply be an error in the data. Maybe a point of contact that was accidently deleted at some point, or it could be that

Parch and Posey's point of contact left the company, and they don't have a new point of contact for that point. Either way there is no data in this cell. This cell is Null.

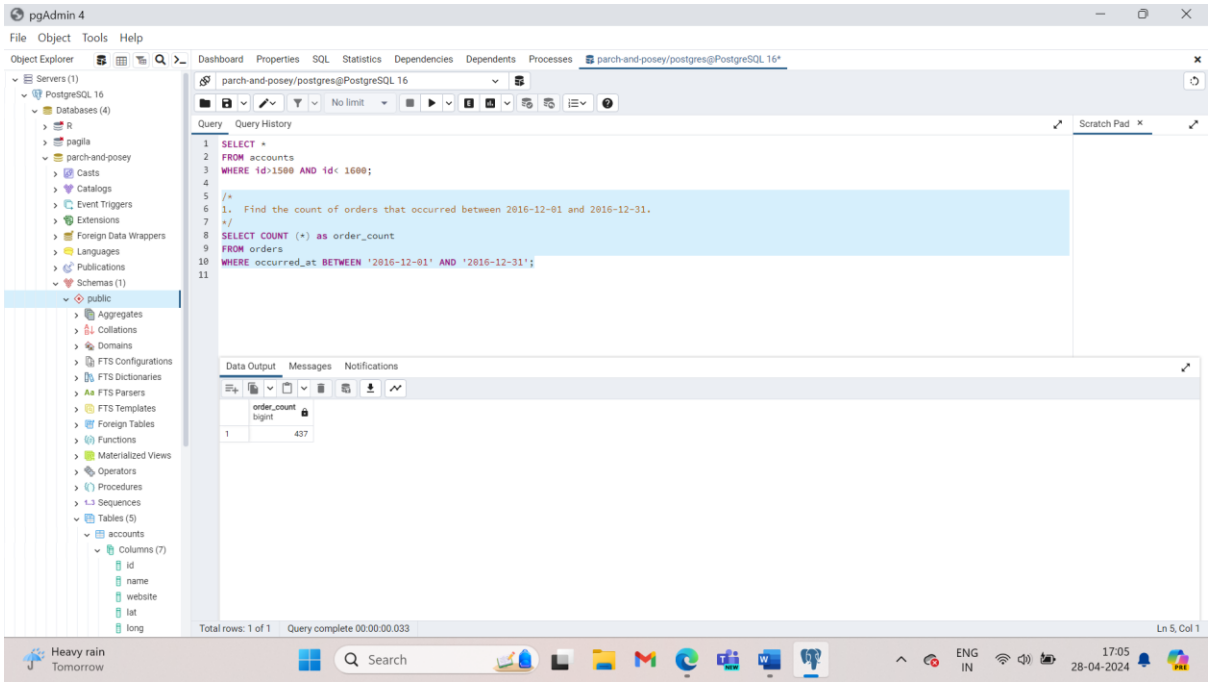
Imagining yourself as a sales manager at Parch and Posy, you might want to know all the accounts for which the primary POC is null. If you don't have a point of contact, chances are you're not going to be able to keep that customer for much longer. To find all these accounts, we'll have to use some special syntax in our WHERE clause. Turns out there are a few accounts without points of contact.

1. Find the count of orders that occurred between 2016-12-01 and 2016-12-31.

SELECT COUNT (*) as order_count

FROM orders

WHERE occurred_at BETWEEN '2016-12-01' AND '2016-12-31';



You can see that it returns the output in a single column.

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

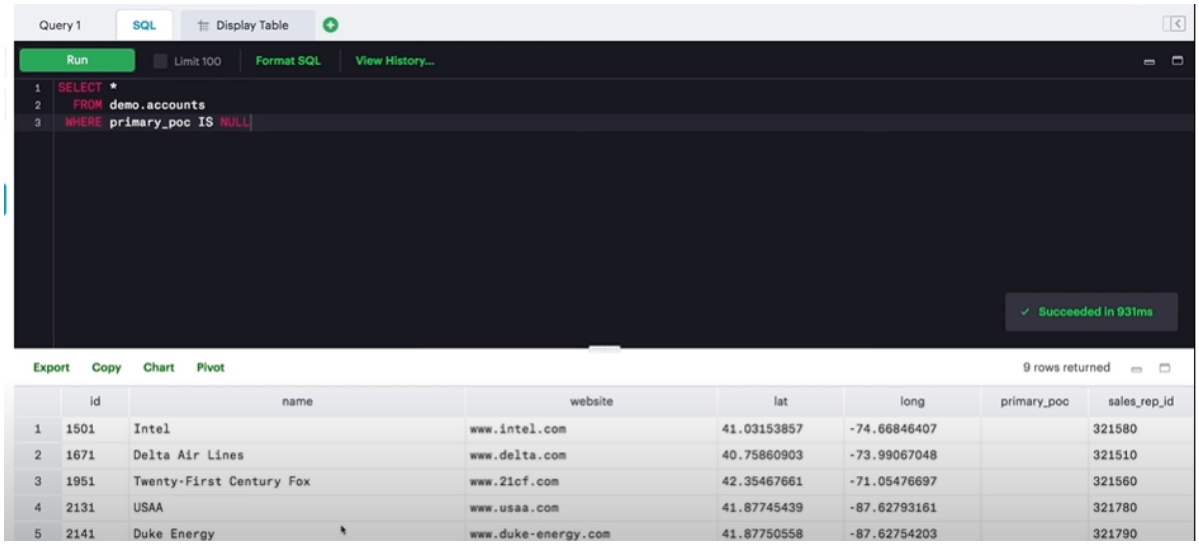
Note that COUNT does not consider rows that have NULL values. Therefore, this can be useful for quickly identifying which rows have missing values.

The COUNT function can also be used to count the number of null records in an individual column.

SELECT COUNT(primary_poc) AS primary_poc_count

FROM accounts

WHERE primary_poc IS NULL



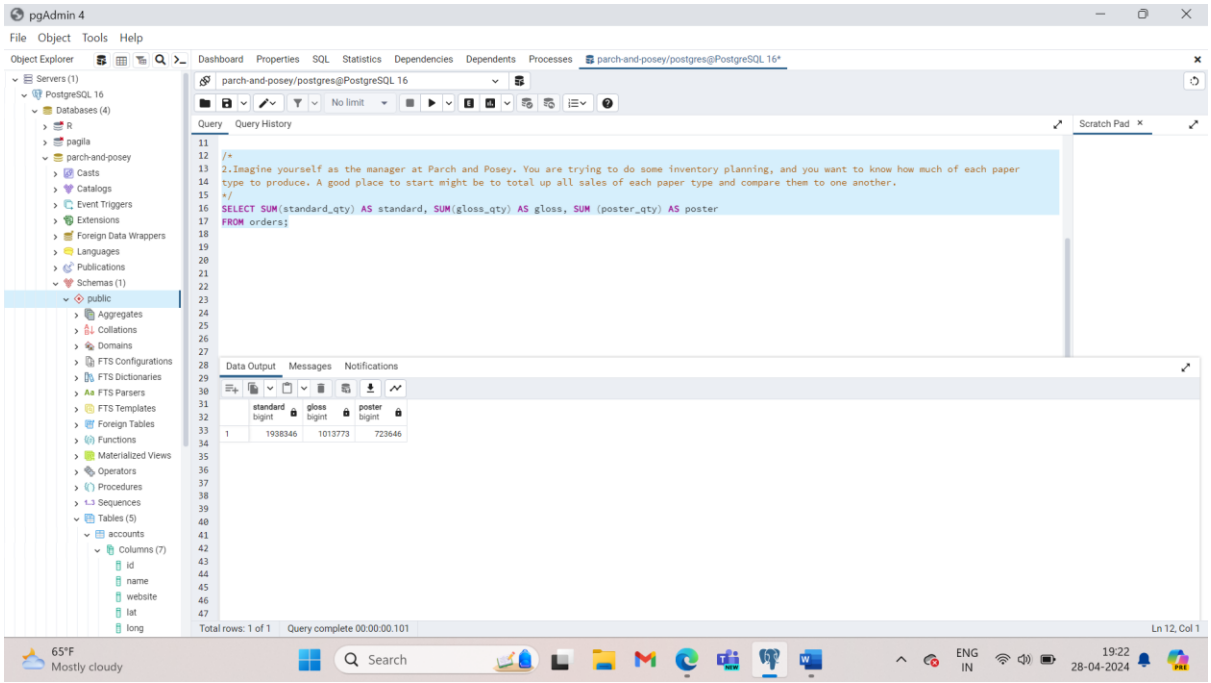
	id	name	website	lat	long	primary_poc	sales_rep_id
1	1501	Intel	www.intel.com	41.03153857	-74.66846407		321580
2	1671	Delta Air Lines	www.delta.com	40.75860903	-73.99067048		321510
3	1951	Twenty-First Century Fox	www.21cf.com	42.35467661	-71.05476697		321560
4	2131	USAA	www.usaa.com	41.87745439	-87.62793161		321780
5	2141	Duke Energy	www.duke-energy.com	41.87750558	-87.62754203		321790

COUNT can be used for columns with non-numerical values.

2. Imagine yourself as the manager at Parch and Posey. You are trying to do some inventory planning, and you want to know how much of each paper type to produce. A good place to start might be to total up all sales of each paper type and compare them to one another.

SELECT SUM(standard_qty) AS standard, SUM(gloss_qty) AS gloss, SUM (poster_qty) AS poster

FROM orders;

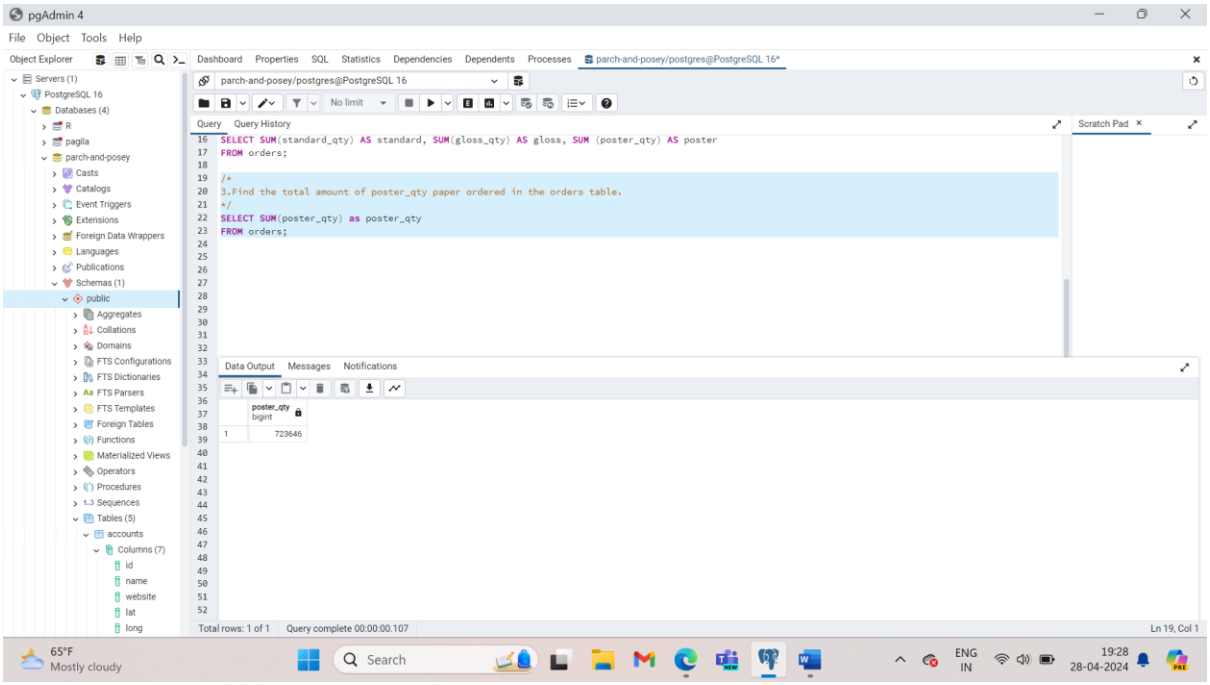


Looks like standard is more popular than both of the non-standard paper types combined.

Unlike COUNT, you can only use SUM function on columns containing numerical values. You need not worry about the nulls. The SUM function will treat the nulls as zeros.

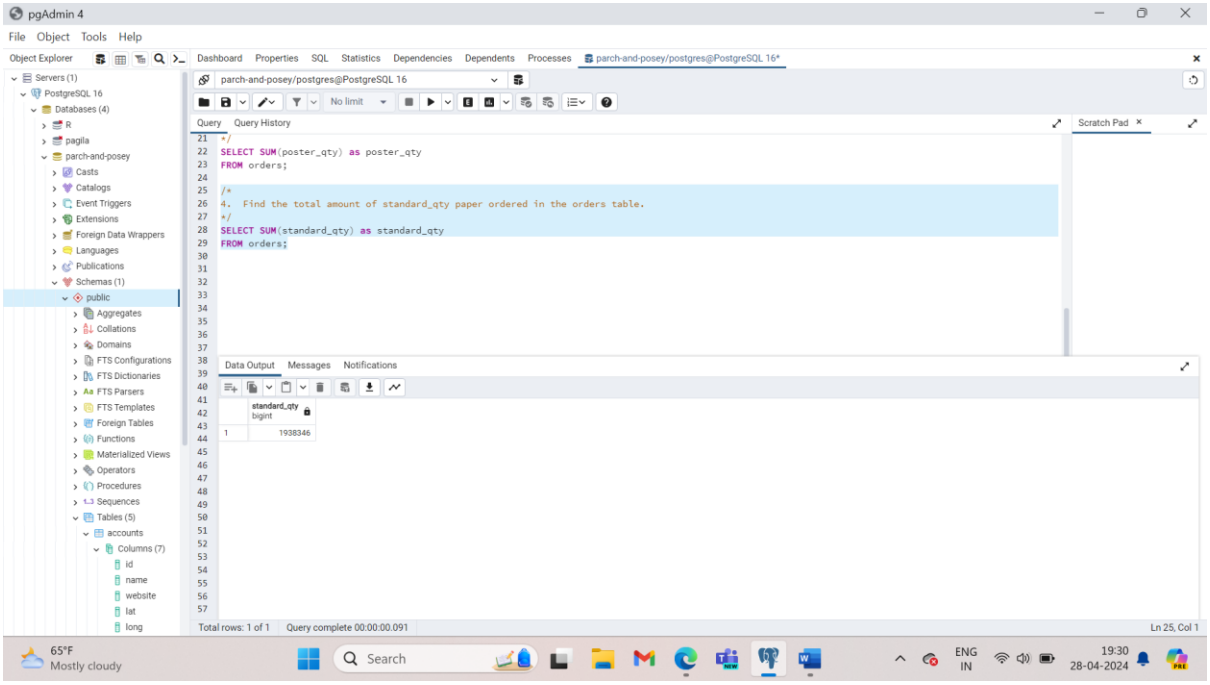
3. Find the total amount of poster_qty paper ordered in the orders table.

```
SELECT SUM(poster_qty) as total_poster_sales
FROM orders;
```



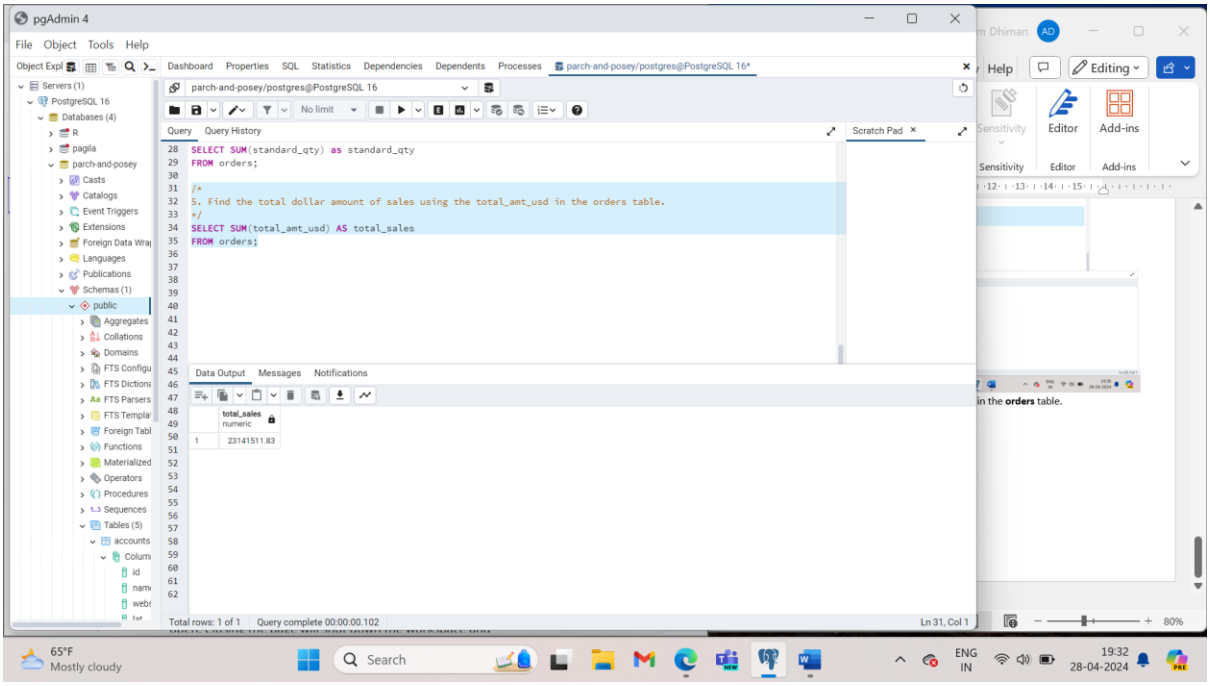
4. Find the total amount of standard_qty paper ordered in the orders table.

```
SELECT SUM(standard_qty) as total_standard_sales
FROM orders;
```

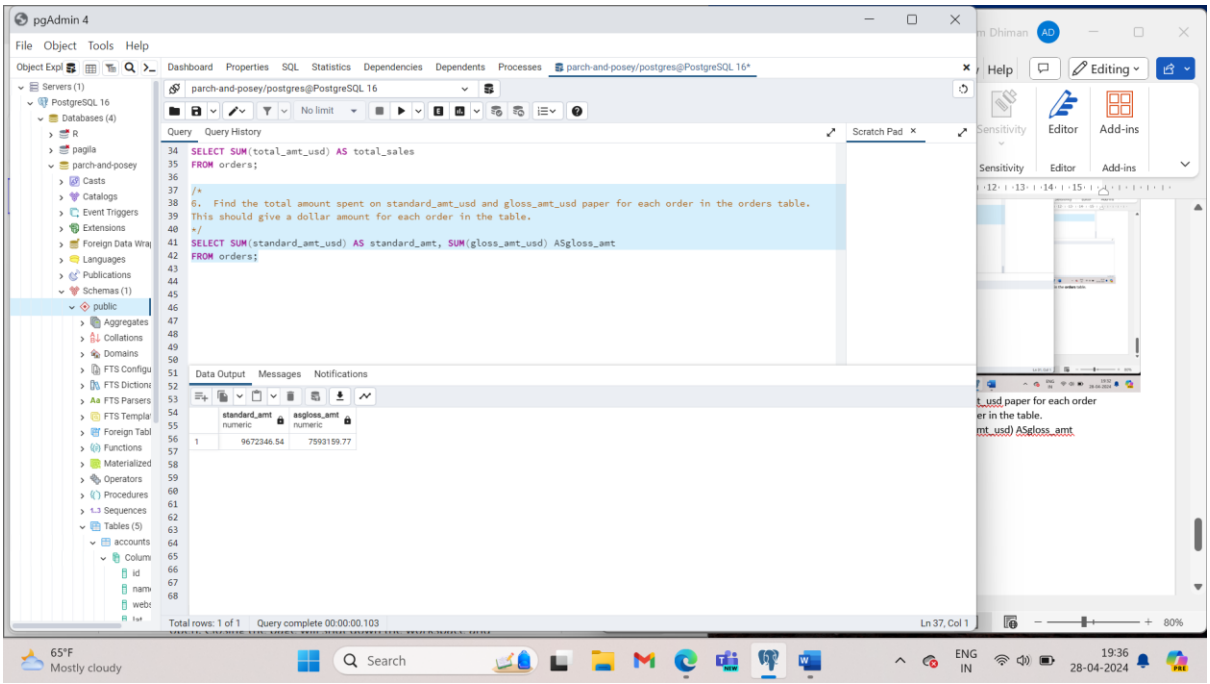


5. Find the total dollar amount of sales using the **total_amt_usd** in the **orders** table.

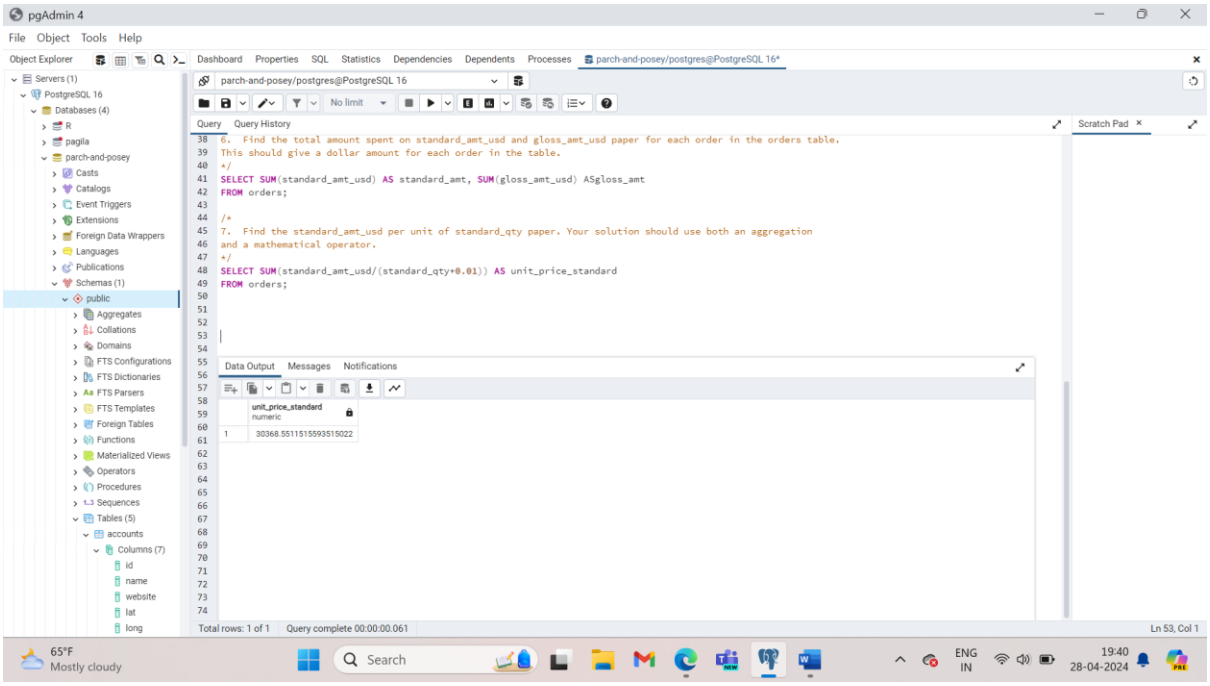
```
SELECT SUM(total_amt_usd) AS total_sales
FROM orders;
```



6. 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.
SELECT standard_amt_usd + gloss_amt_usd AS total_standard_gloss
FROM orders;



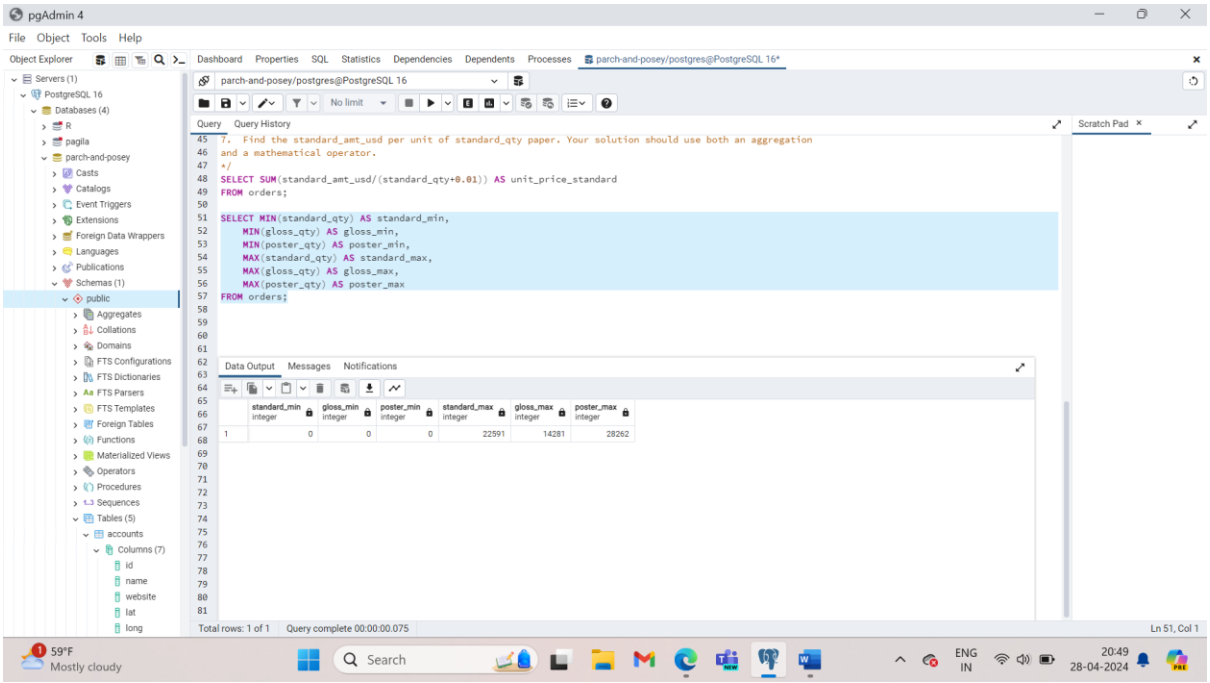
7. Find the standard_amt_usd per unit of standard_qty paper. Your solution should use both an aggregation and a mathematical operator.
SELECT SUM(standard_amt_usd/SUM(standard_qty) AS unit_price_standard
FROM orders;



The syntax for MIN and MAX is like COUNT.

There is no surprise that the MIN for each paper type is zero. Some customers order only one or two types of paper. What is surprising is that the largest single order is for poster paper even though it is the least popular overall. There are important implications here. Even though it's not the most popular item, Parch and Posey might want to produce enough poster paper to be able to fulfil big orders at any given time.

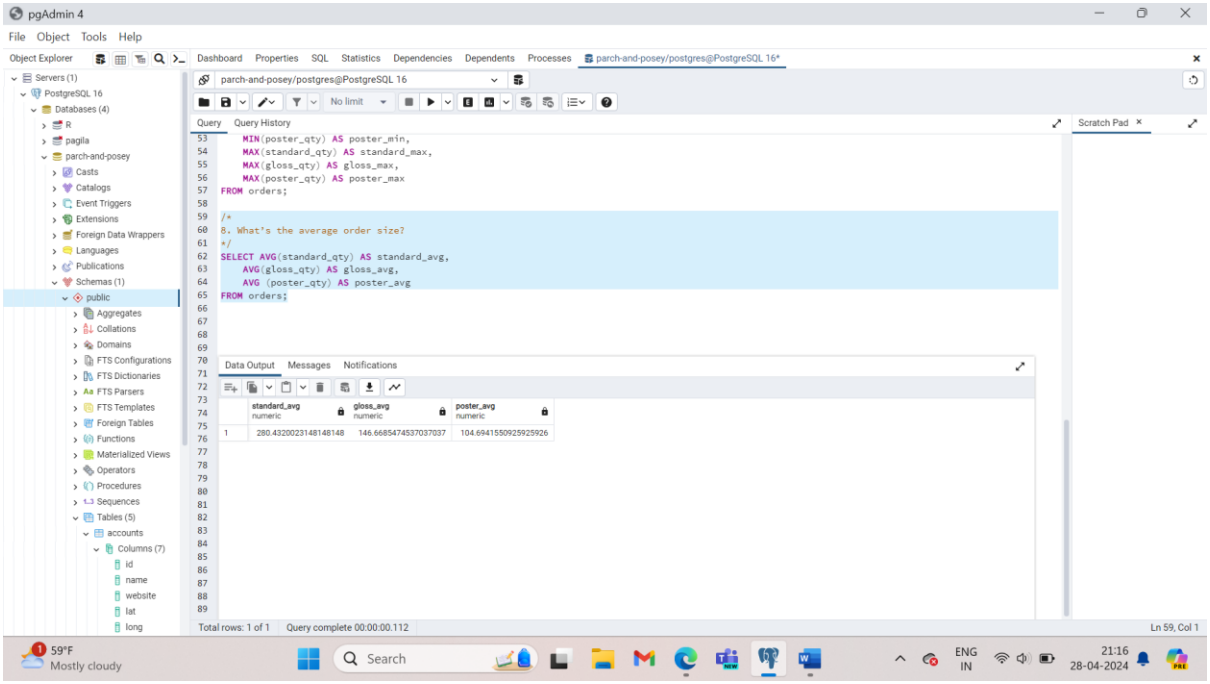
```
SELECT MIN(standard_qty) AS standard_min,
       MIN(gloss_qty) AS gloss_min,
       MIN(poster_qty) AS poster_min,
       MAX(standard_qty) AS standard_max,
       MAX(gloss_qty) AS gloss_max,
       MAX(poster_qty) AS poster_max
FROM orders;
```



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 values 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.

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

```
SELECT AVG(standard_qty) AS standard_avg,
       AVG(gloss_qty) AS gloss_avg,
       AVG (poster_qty) AS poster_avg
FROM orders;
```

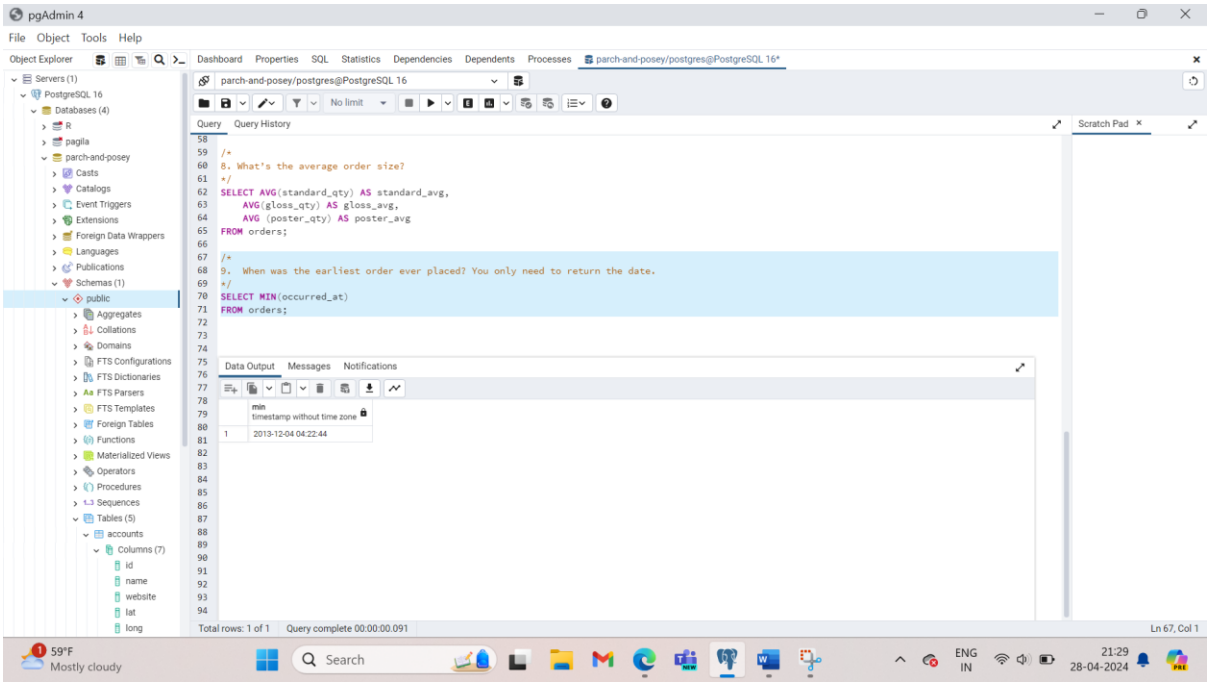


EXPERT TIP

Typically, to **calculate the median** in SQL, you would need to **employ techniques such as using window functions or subqueries to rank the data and then selecting the middle value or interpolating between the two middle values if the dataset has an even number of elements.** This process can be quite intricate, especially when dealing with large datasets.

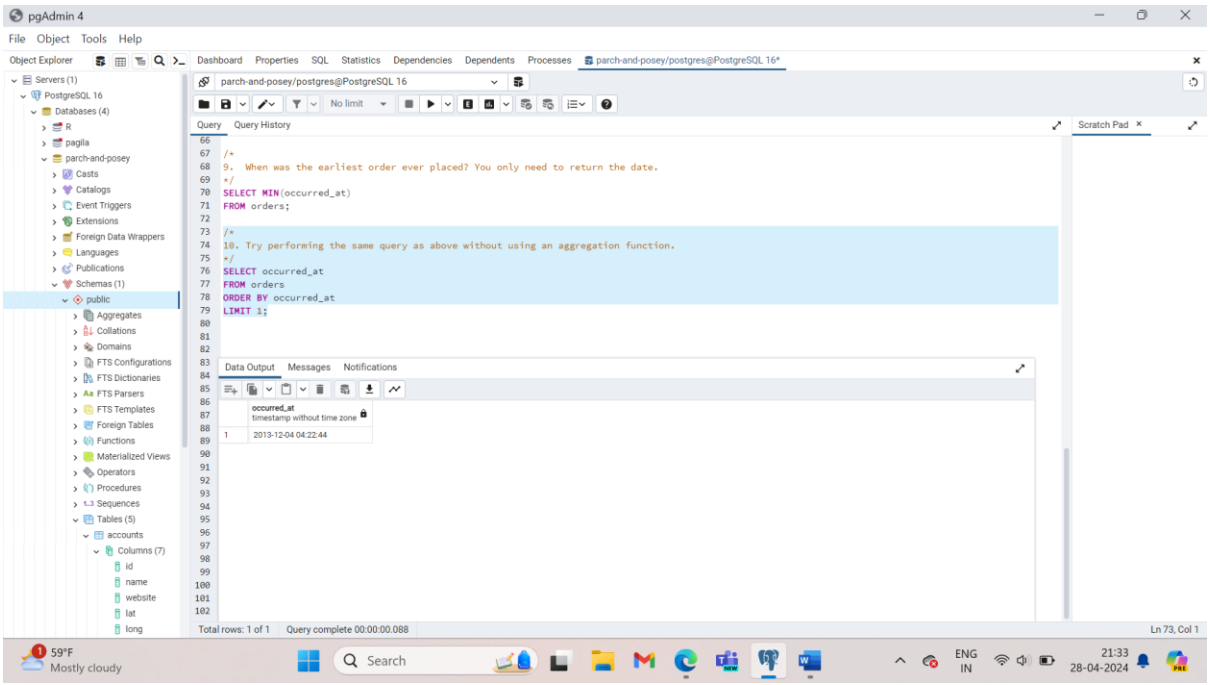
9. When was the earliest order ever placed? You only need to return the date.

```
SELECT MIN(occurred_at)
FROM orders;
```



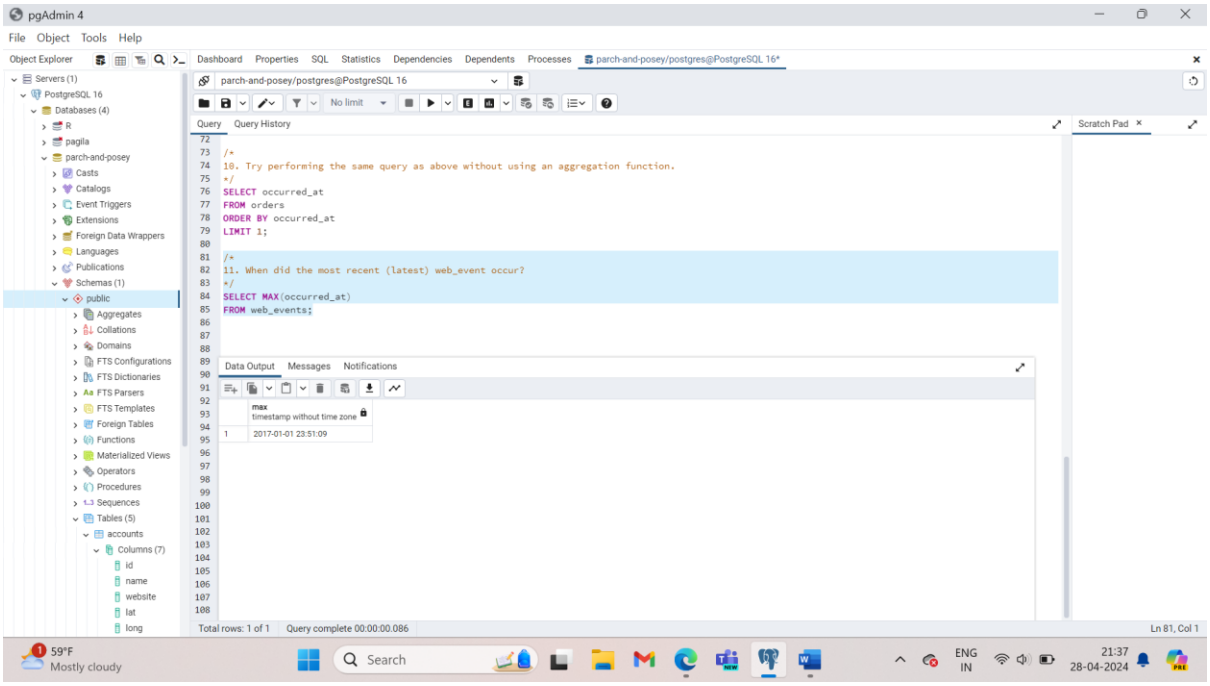
10. Try performing the same query as above without using an aggregation function.

```
SELECT occurred_at
FROM orders
ORDER BY occurred_at
LIMIT 1;
```



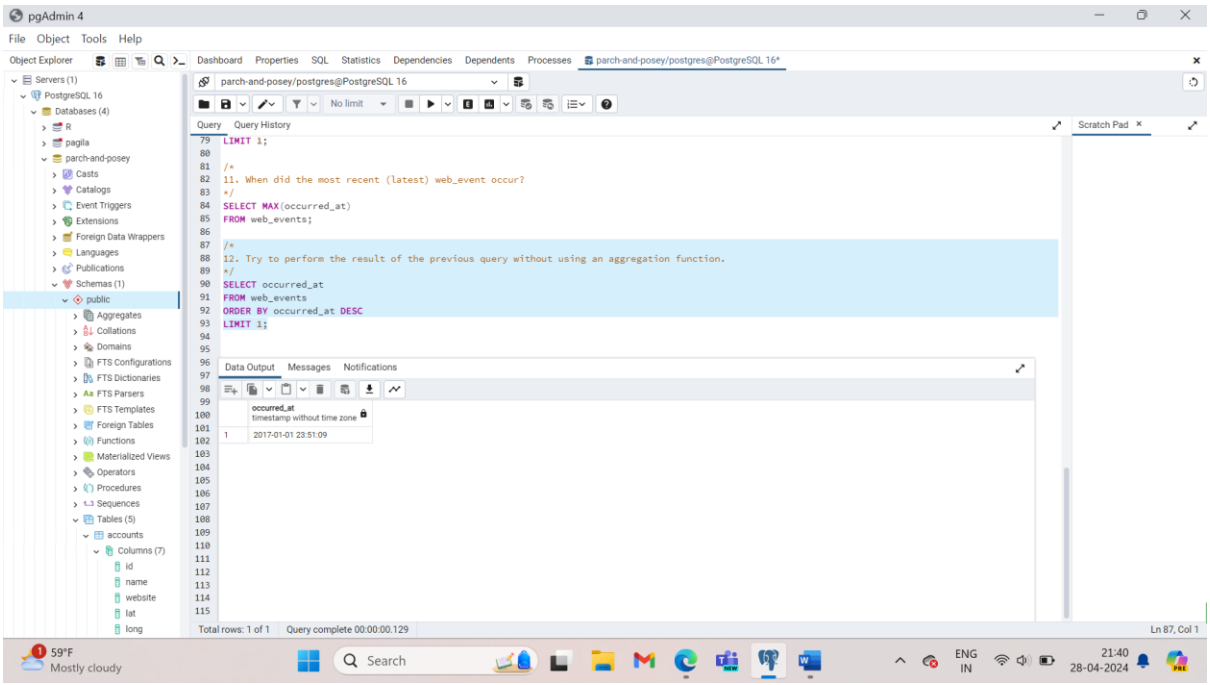
11. When did the most recent (latest) **web_event** occur?

```
SELECT MAX(occurred_at)
FROM web_events;
```



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

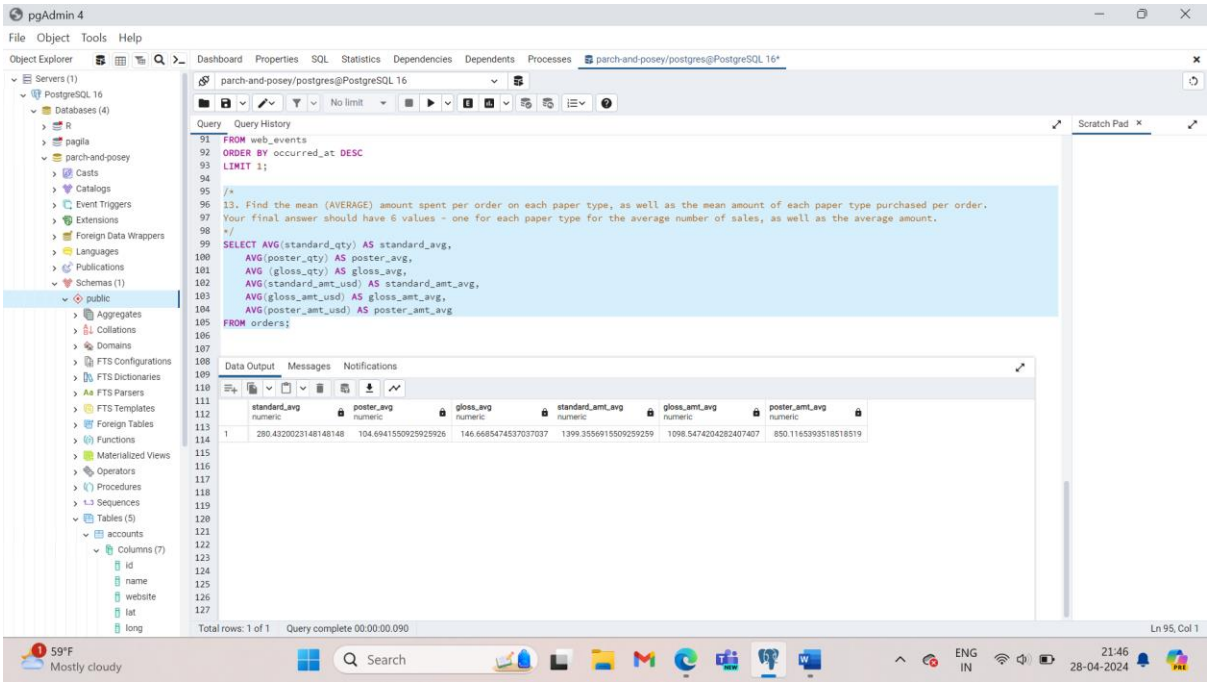
```
SELECT occurred_at
FROM web_events
ORDER BY occurred_at DESC
LIMIT 1;
```



13. 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.

```
SELECT AVG(standard_qty) AS standard_avg,
       AVG(poster_qty) AS poster_avg,
       AVG (gloss_qty) AS gloss_avg,
       AVG(standard_amt_usd) AS standard_amt_avg,
       AVG(gloss_amt_usd) AS gloss_amt_avg,
```

AVG(poster_amt_usd) AS poster_amt_avg
FROM orders;



14. what is the MEDIAN **total_usd** spent on all **orders**?

SELECT *
FROM (SELECT total_amt_usd
FROM orders
ORDER BY total_amt_usd
LIMIT 3457) AS Table1

ORDER BY total_amt_usd DESC

LIMIT 2;

Since there are 6912 orders - we want the average of the 3457 and 3456 order amounts when ordered. This is the average of 2483.16 and 2482.55. This gives the median of **2482.855**. This obviously isn't an ideal way to compute. If we obtain new orders, we will have to change the limit. SQL didn't even calculate the median for us. The above used a SUBQUERY, but you could use any method to find the two necessary values, and then you just need the average of them.

As a sales manager, you might want to sum all of the sales of each paper type for each account. Group by allows you to create segments that will aggregate independently of one another. In other words, group by allows you to take the sum of data limited to each account rather than across the entire dataset.

We want to tell the query to aggregate into segments, where each segment is one of the values in the account id column. Do this using GROUP BY clause.

SELECT account_id,

SUM (standard_qty) AS standard_sum,

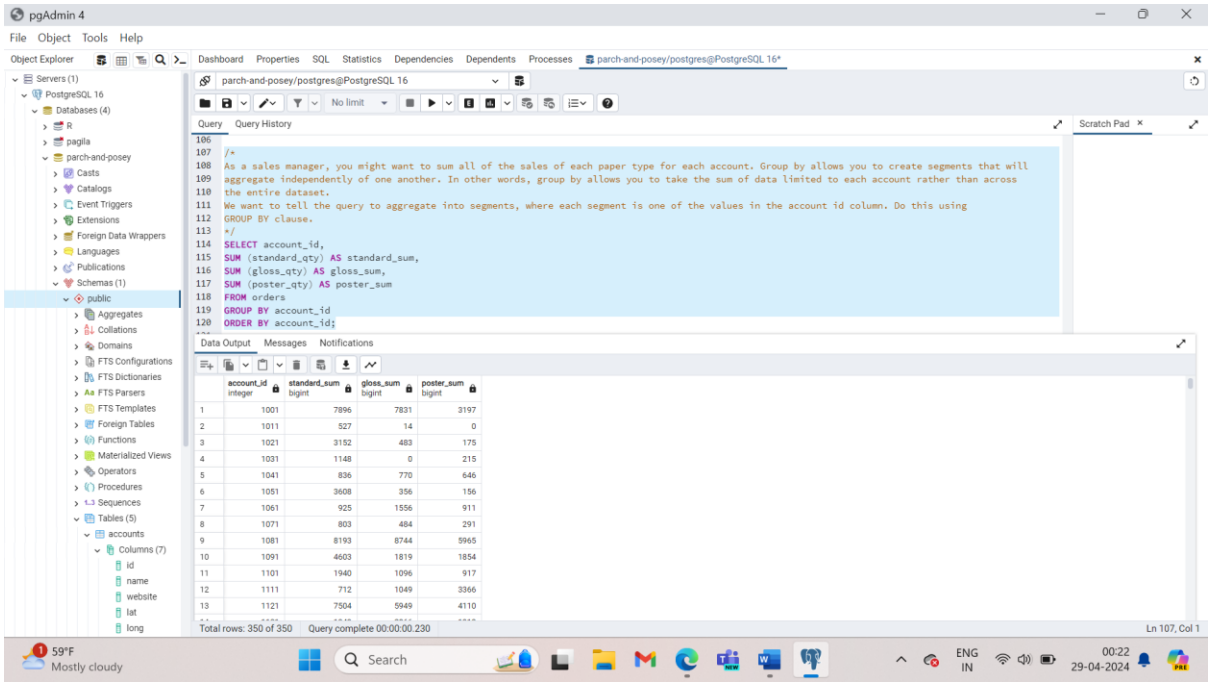
SUM (gloss_qty) AS gloss_sum,

SUM (poster_qty) AS poster_sum

FROM orders

GROUP BY account_id

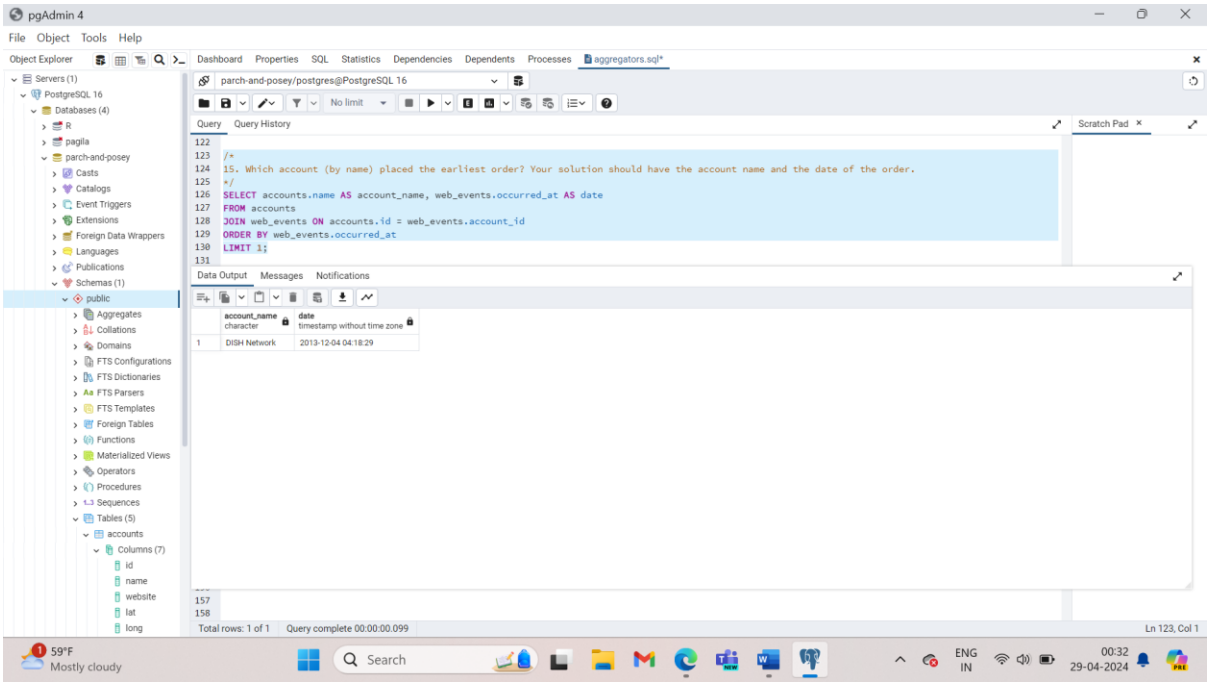
ORDER BY account_id;



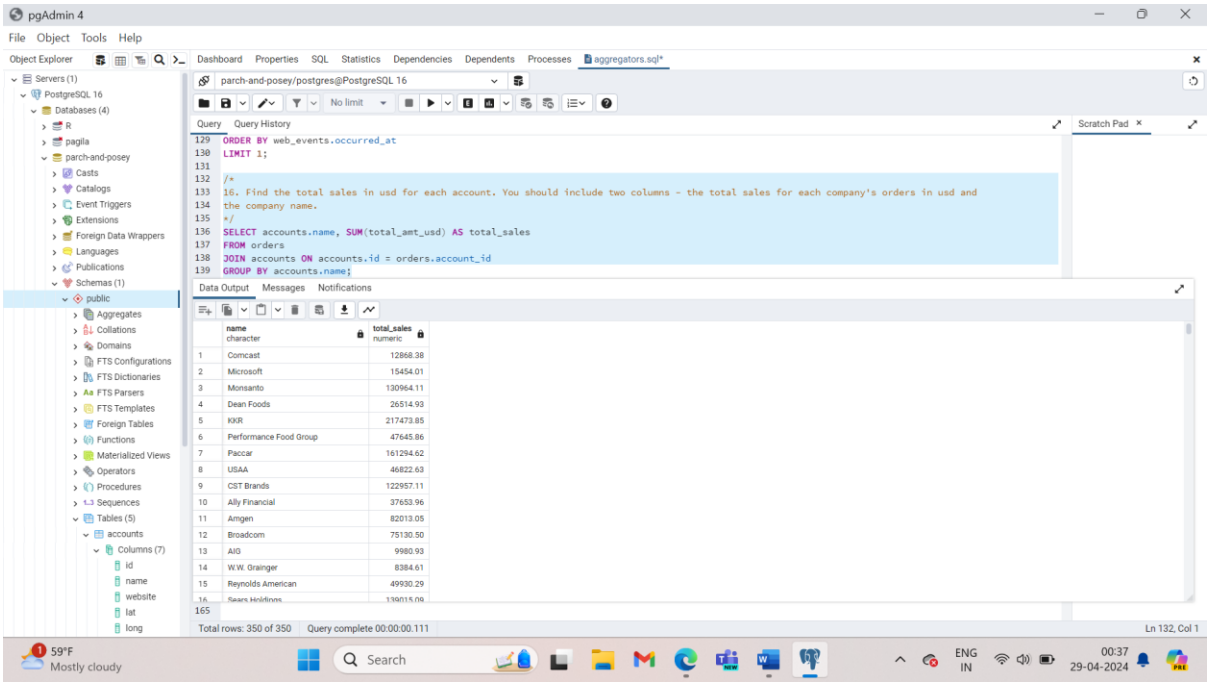
The key takeaways here:

- **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.

15. Which **account** (by name) placed the earliest order? Your solution should have the **account name** and the **date** of the order.
    SELECT accounts.name AS account_name, web_events.occurred_at AS date
    FROM accounts
    JOIN web_events ON accounts.id = web_events.account_id
    ORDER BY web_events.occurred_at
    LIMIT 1;

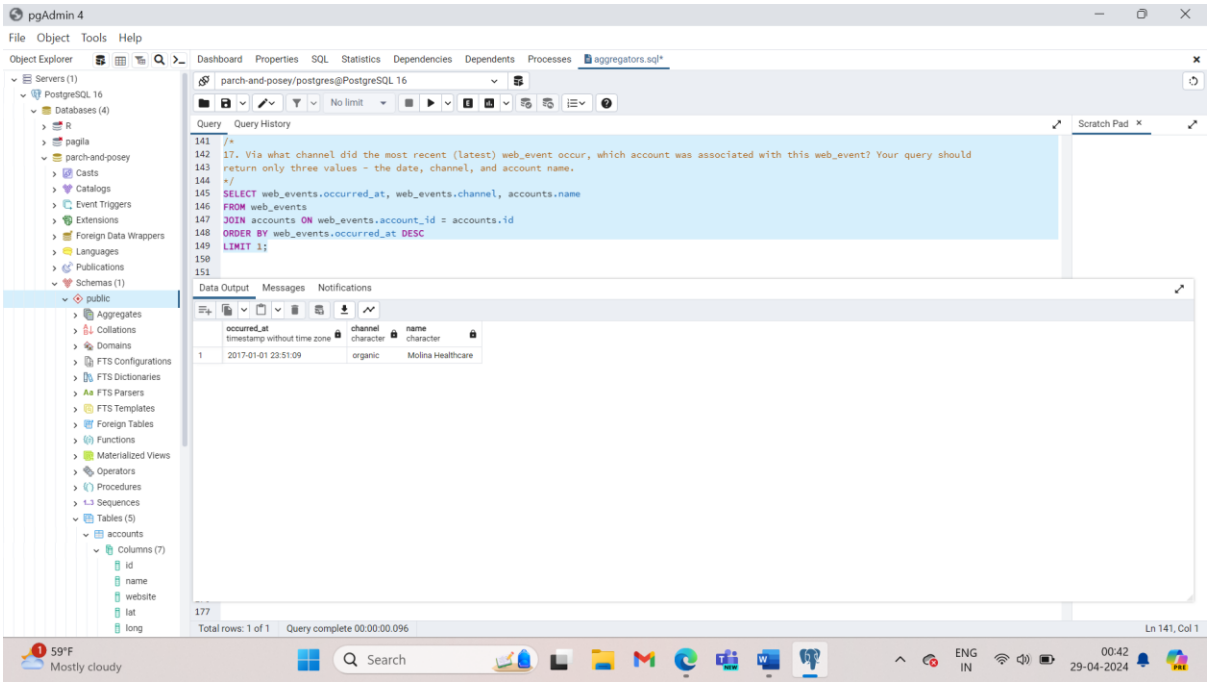


16. Find the total sales in usd for each account. You should include two columns - the total sales for each company's orders in usd and the company name.
SELECT accounts.name, SUM(total_amt_usd) AS total_sales
FROM orders
JOIN accounts ON accounts.id = orders.account_id
GROUP BY accounts.name;

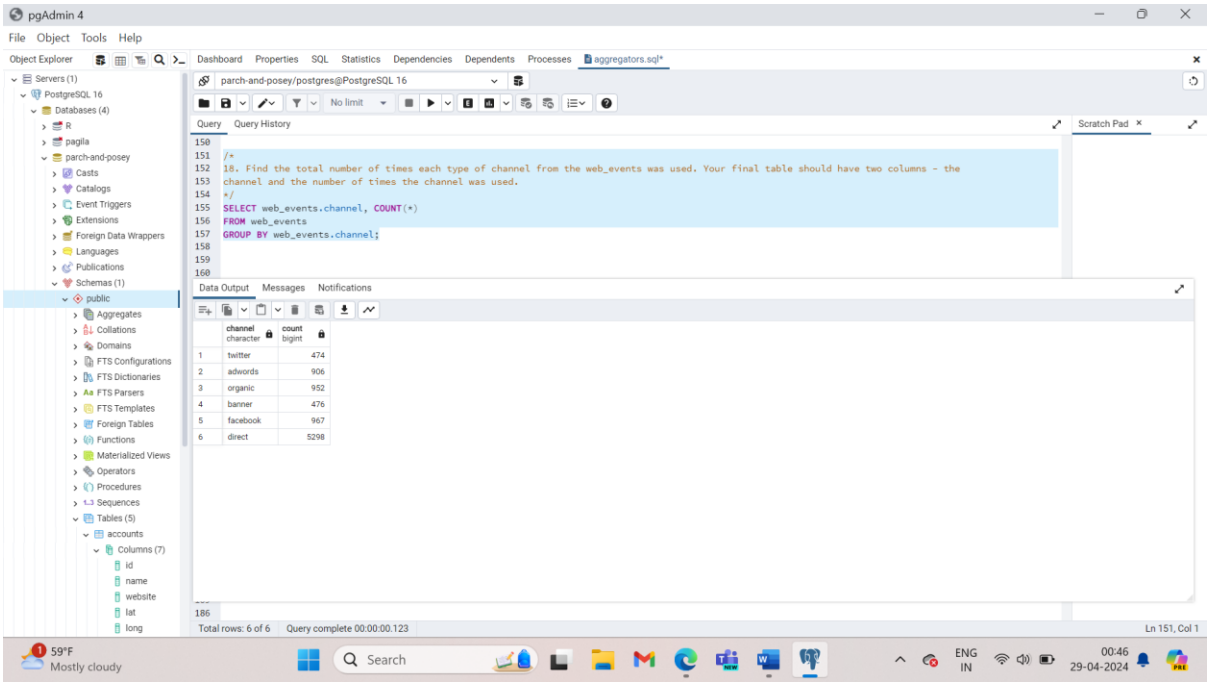


17. Via what channel did the most recent (latest) web_event occur, which account was associated with this web_event? Your query should return only three values - the date, channel, and account name.
SELECT web_events.occurred_at, web_events.channel, accounts.name
FROM web_events
JOIN accounts ON web_events.account_id = accounts.id
ORDER BY web_events.occurred_at DESC

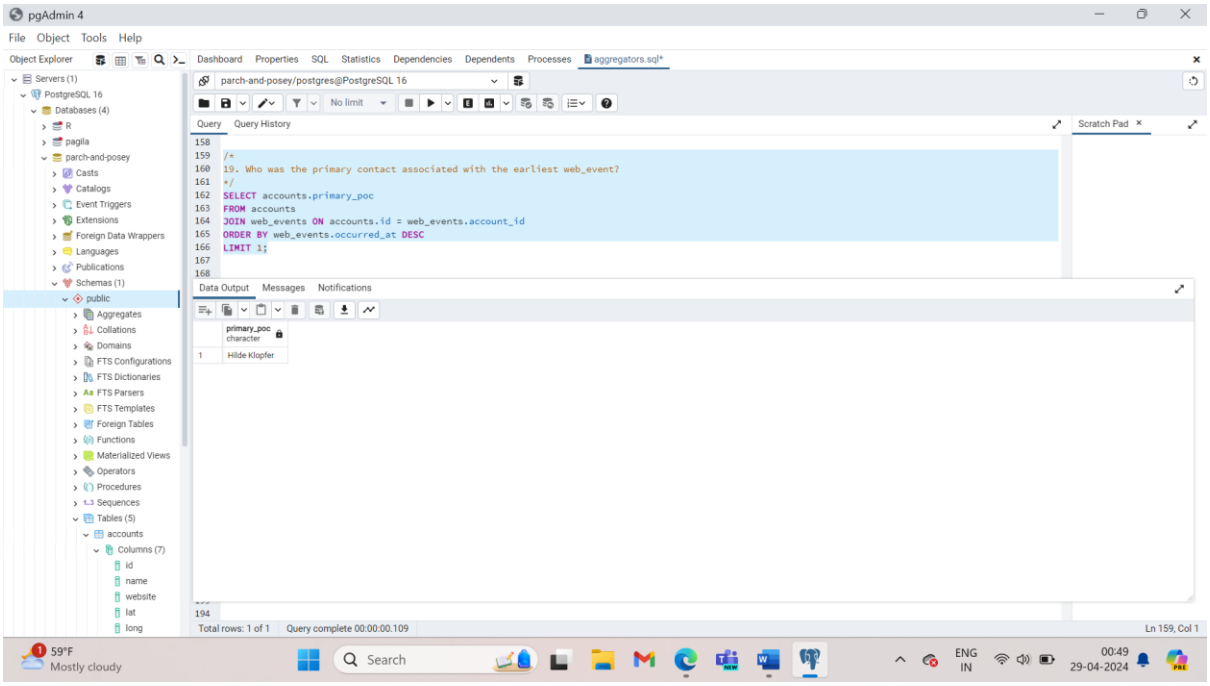
LIMIT 1;



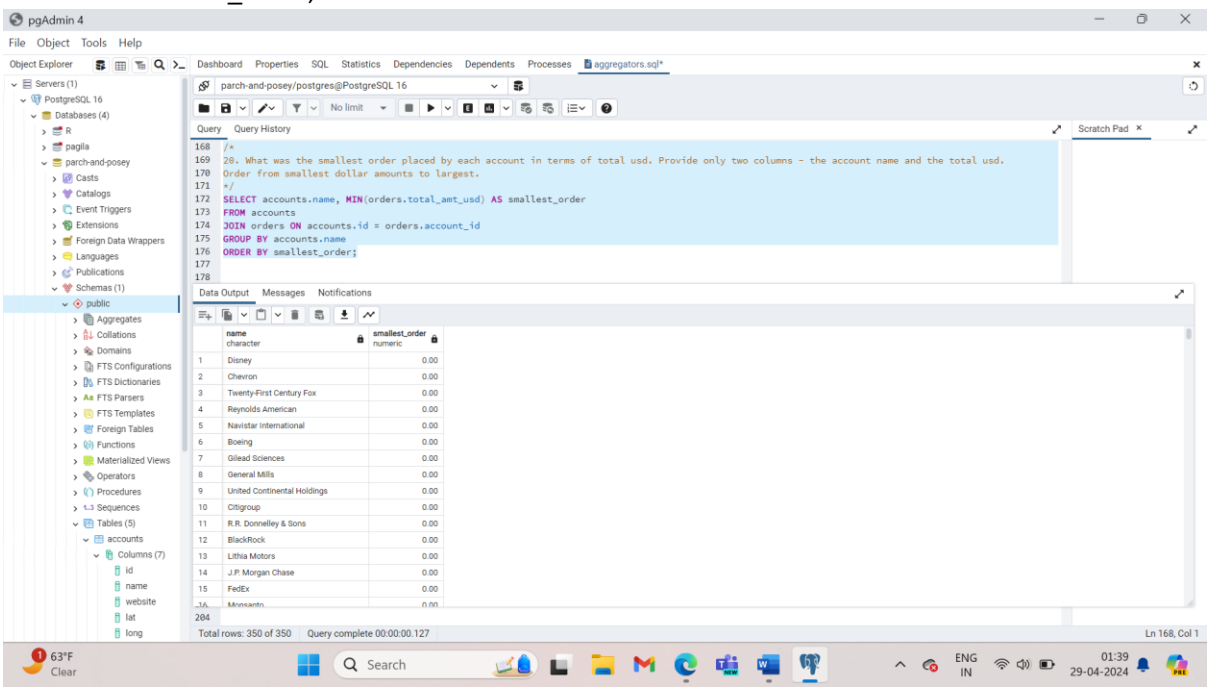
18. Find the total number of times each type of channel from the web_events was used. Your final table should have two columns - the channel and the number of times the channel was used.
SELECT w.channel, COUNT(*)
FROM web_events
GROUP BY web_events.channel;



19. Who was the primary contact associated with the earliest web_event?
SELECT accounts.primary_poc
FROM accounts
JOIN web_events ON accounts.id = web_events.account_id
ORDER BY web_events.occurred_at DESC
LIMIT 1;

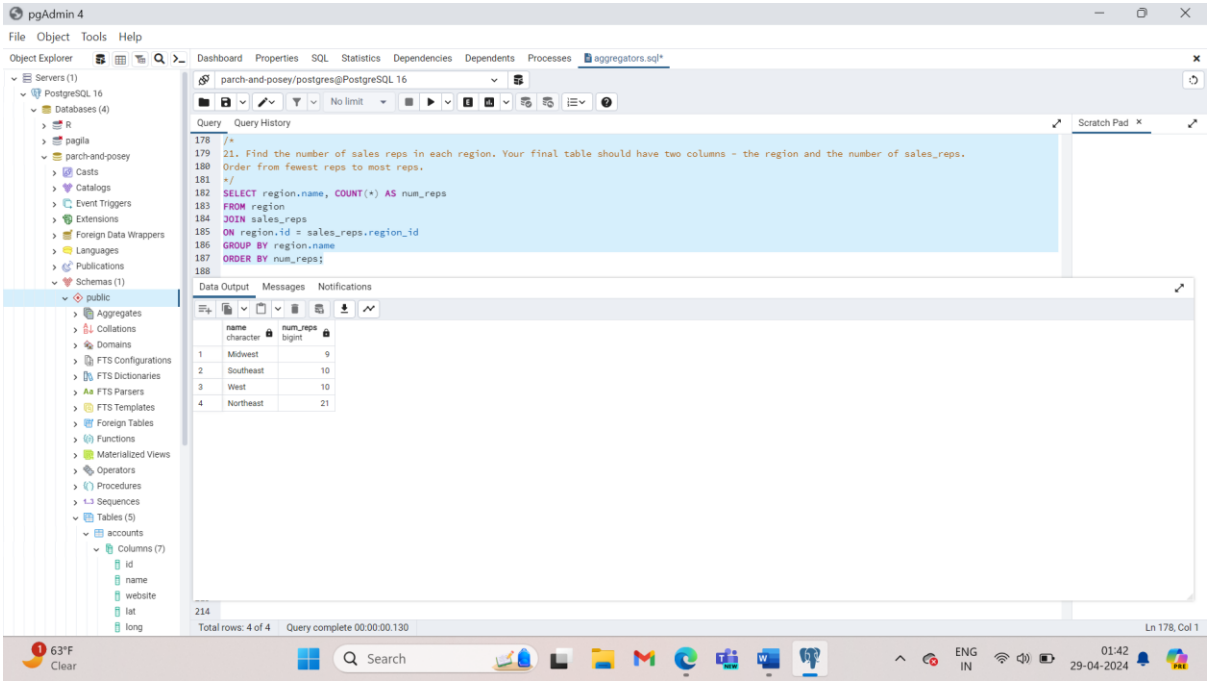


20. What was the smallest order placed by each account in terms of total usd. Provide only two columns - the account name and the total usd. Order from smallest dollar amounts to largest.
SELECT accounts.name, MIN(orders.total_amt_usd) AS smallest_order
FROM accounts
JOIN orders ON accounts.id = orders.account_id
GROUP BY accounts.name
ORDER BY smallest_order;



21. 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.
SELECT region.name, COUNT(*) AS num_reps
FROM region
JOIN sales_reps

```
ON region.id = sales_reps.region_id
GROUP BY region.name
ORDER BY num_reps;
```

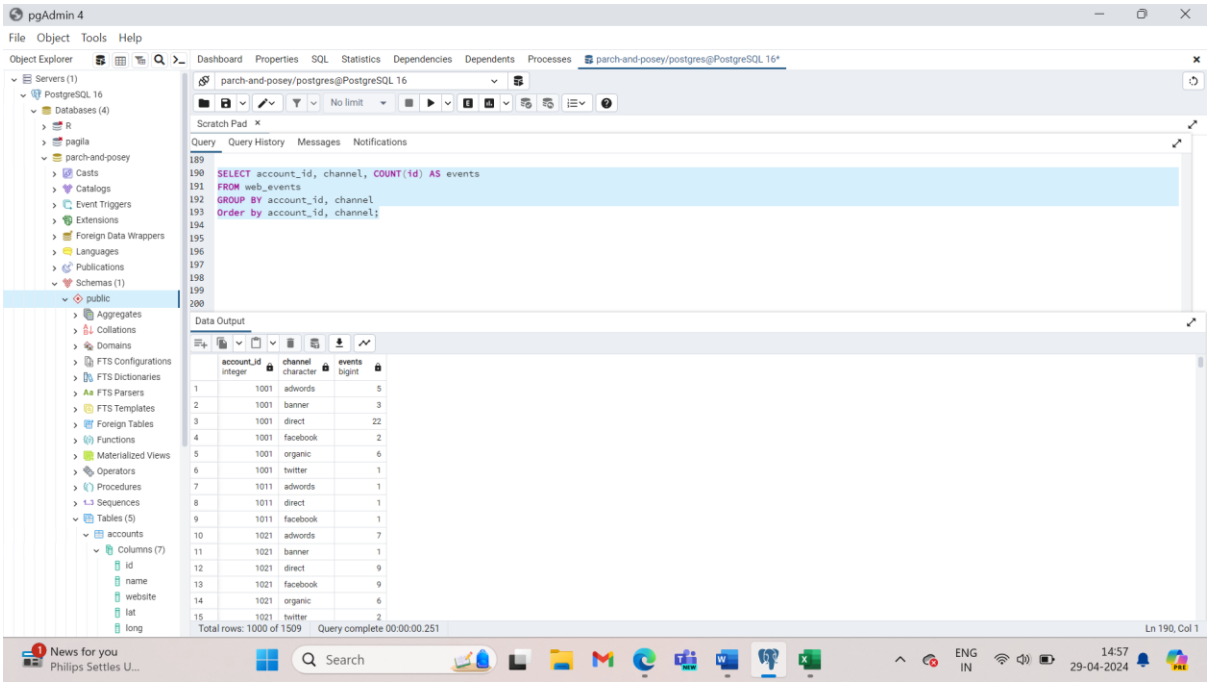


Imagine yourself as a marketing manager at Parch and Posey, trying to understand how each account interacted with various advertising channels. Which channels are driving traffic and leading to purchases? Are we investing in channels that aren't worth the cost? How much traffic are we obtaining from each channel?

One way we can investigate this is to count all the events for each channel, for each account ID. It looks like the useful information here is the events column. Let's reorder this to highlight the highest volume channels for each account.

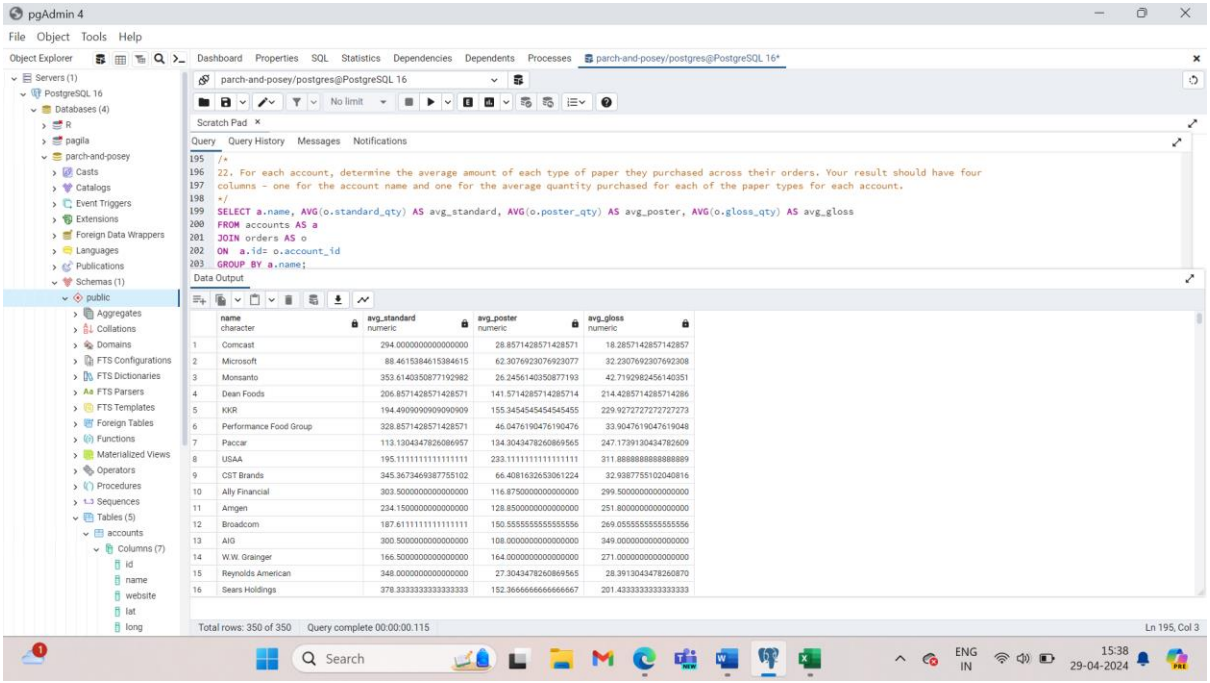
A reminder here that any column that is not within an aggregation must show up in your GROUP BY statement. If you forget, you will likely get an error. However, in the off chance that your query does work, you might not like the results!

```
SELECT account_id, channel, COUNT(id) AS events
FROM web_events
GROUP BY account_id, channel
Order by account_id, channel;
```



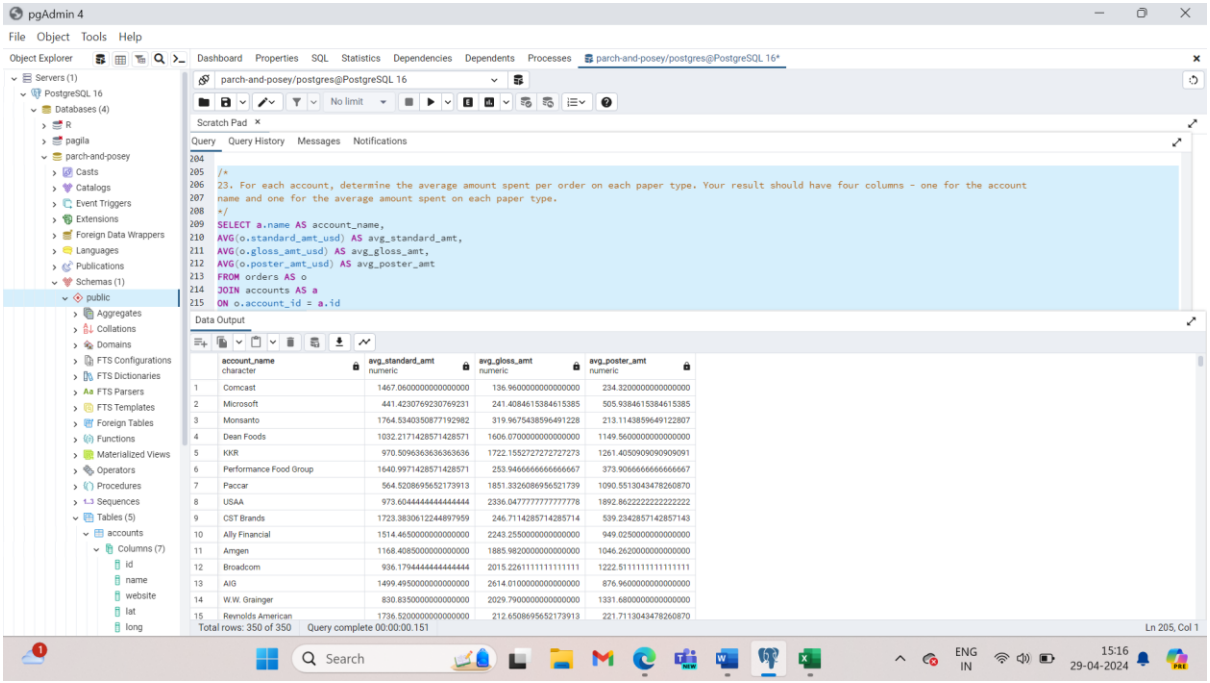
22. For each account, determine the average amount of each type of paper they purchased across their orders. Your result should have four columns - one for the account name and one for the average quantity purchased for each of the paper types for each account.

```
SELECT a.name, AVG(o.standard_qty) AS avg_standard, AVG(o.poster_qty) AS avg_poster,
AVG(o.gloss_qty) AS avg_gloss
FROM accounts AS a
JOIN orders AS o
ON  a.id= o.account_id
GROUP BY a.name;
```



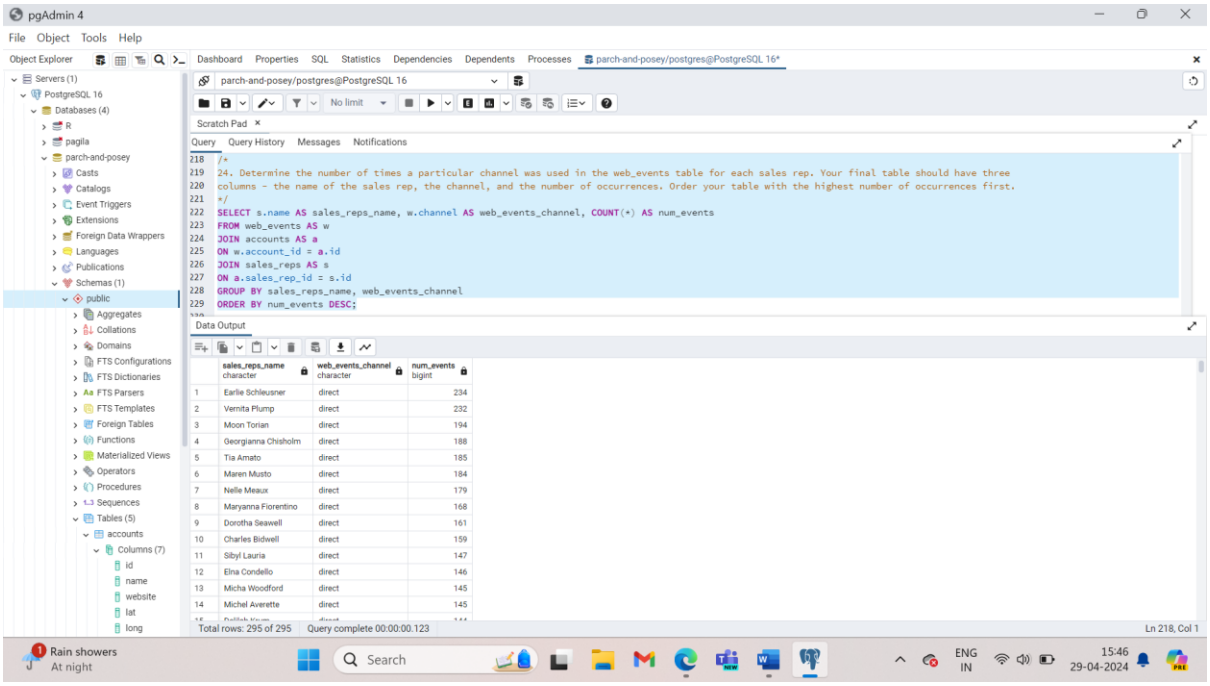
23. For each account, determine the average amount spent per order on each paper type. Your result should have four columns - one for the account name and one for the average amount spent on each paper type.

```
SELECT a.name AS account_name,
        AVG(o.standard_amt_usd) AS avg_standard_amt,
        AVG(o.gloss_amt_usd) AS avg_gloss_amt,
        AVG(o.poster_amt_usd) AS avg_poster_amt
FROM orders AS o
JOIN accounts AS a
ON o.account_id = a.id
GROUP BY A.name;
```



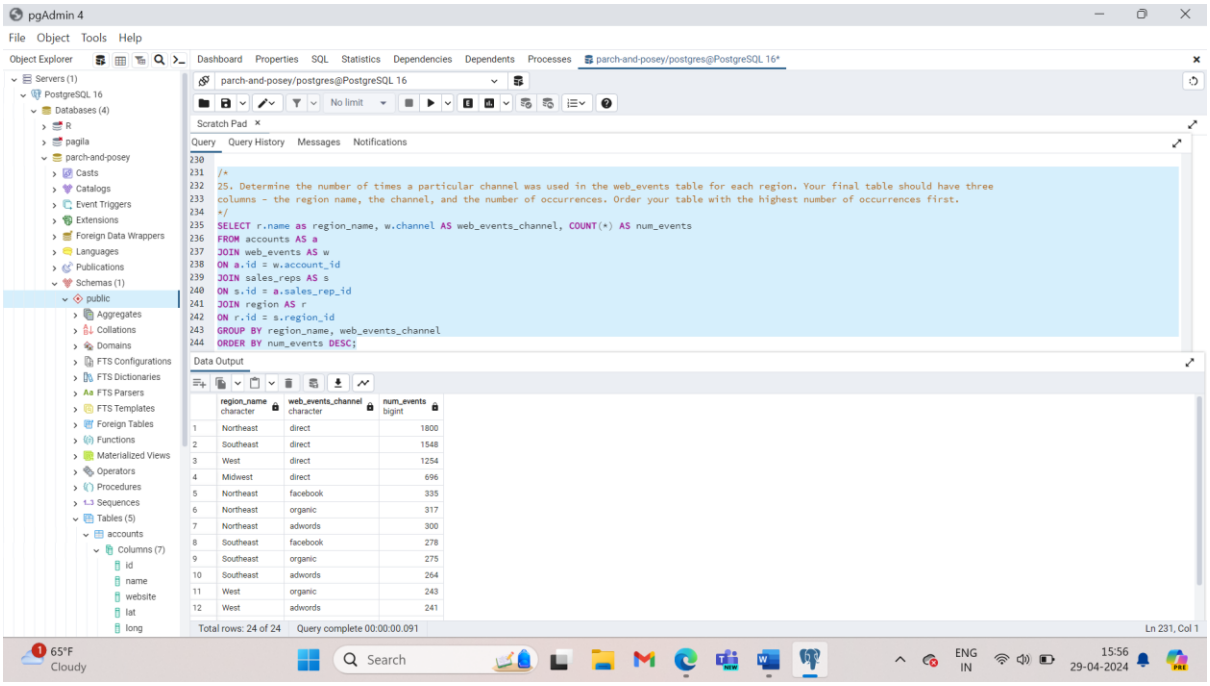
24. Determine the number of times a particular channel was used in the web_events table for each sales rep. Your final table should have three columns - the name of the sales rep, the channel, and the number of occurrences. Order your table with the highest number of occurrences first.

```
SELECT s.name AS sales_reps_name, w.channel AS web_events_channel, COUNT(*) AS
num_events
FROM web_events AS w
JOIN accounts AS a
ON w.account_id = a.id
JOIN sales_reps AS s
ON a.sales_rep_id = s.id
GROUP BY sales_reps, web_events_channel
ORDER BY num_events DESC;
```



25. 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.

```
SELECT r.name as region_name, w.channel AS web_events_channel, COUNT(*) AS
num_events
FROM accounts AS a
JOIN web_events AS w
ON a.id = w.account_id
JOIN sales_reps AS s
ON s.id = a.sales_rep_id
JOIN regions AS r
ON r.id = s.region_id
GROUP BY region_name, web_events_channel
ORDER BY num_events DESC;
```



DISTINCT

You can think of distinct this way: if you want to group by some columns but you don't necessarily want to include ant aggregations, you can use Distinct instead.

Eg: let's revisit the count of events by channel by account we looked in the Group BY segment.

SELECT DISTINCT account_id, channel
FROM web_events
Order by account_id;

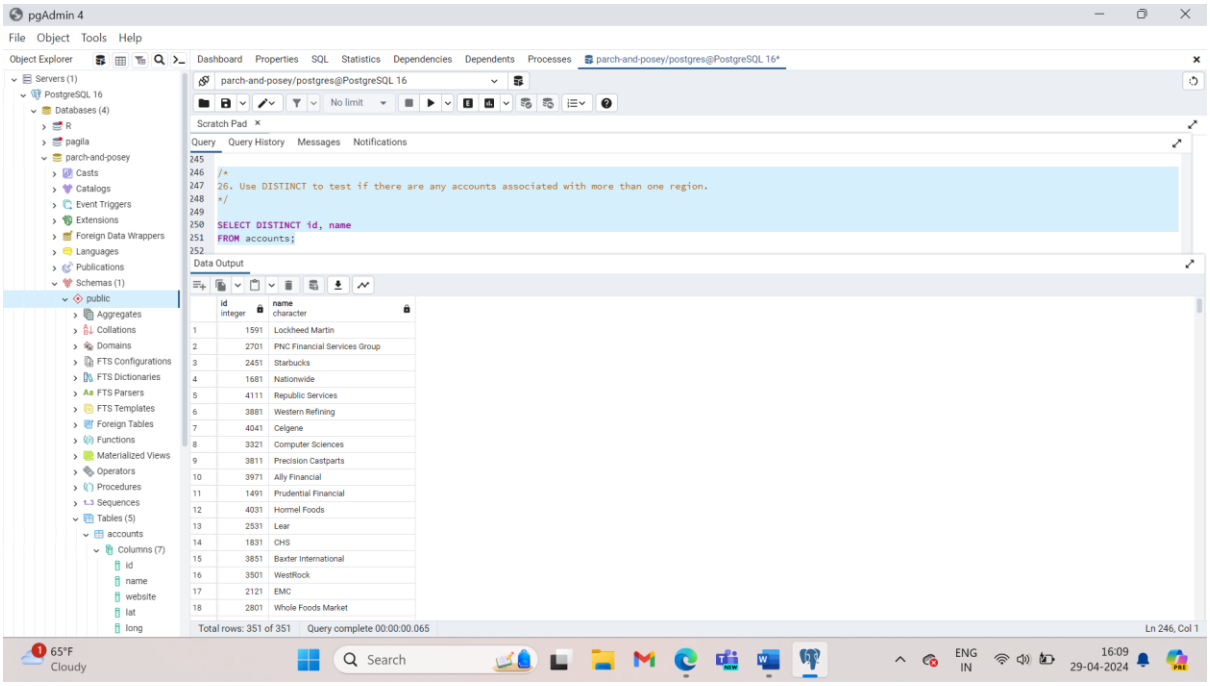
DISTINCT is always used in SELECT statements, and it provides the unique rows for all columns written in the SELECT statement. Therefore, you only use DISTINCT once in any particular SELECT statement.

You could write:

SELECT DISTINCT column1, column2, column3
FROM table1;
Which would return the distinct rows across all three columns.

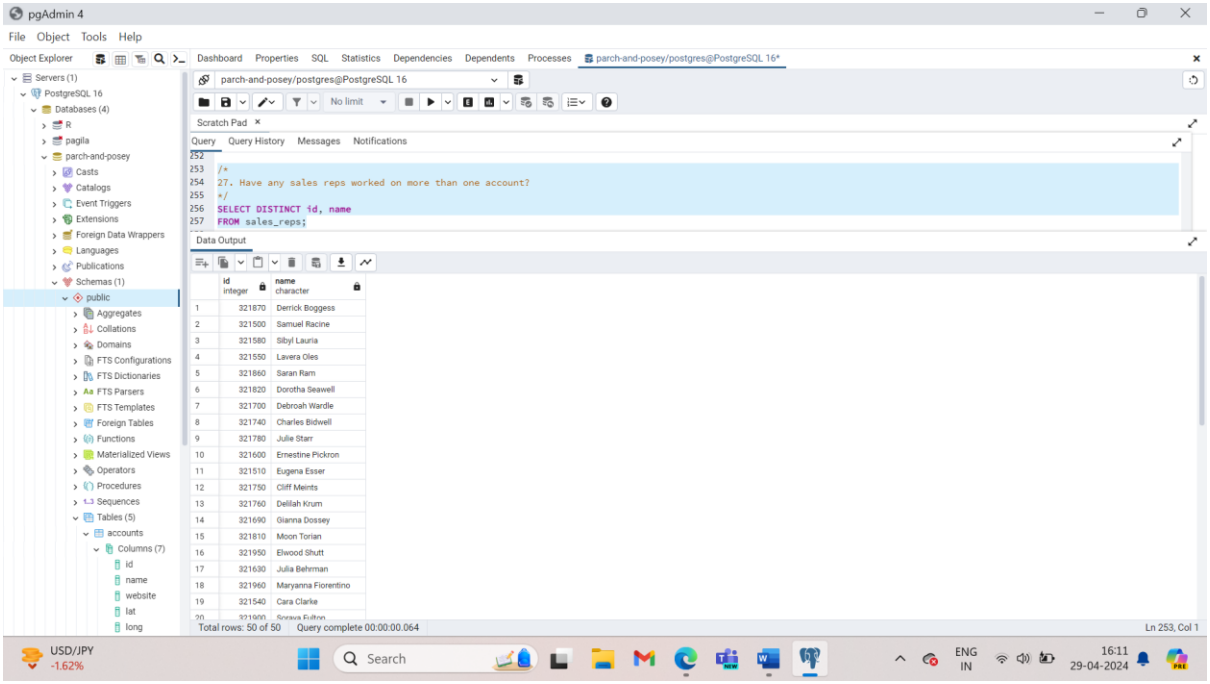
26. Use DISTINCT to test if there are any accounts associated with more than one region.

SELECT DISTINCT id, name
FROM accounts;



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

```
SELECT DISTINCT id, name
FROM sales_reps;
```



HAVING

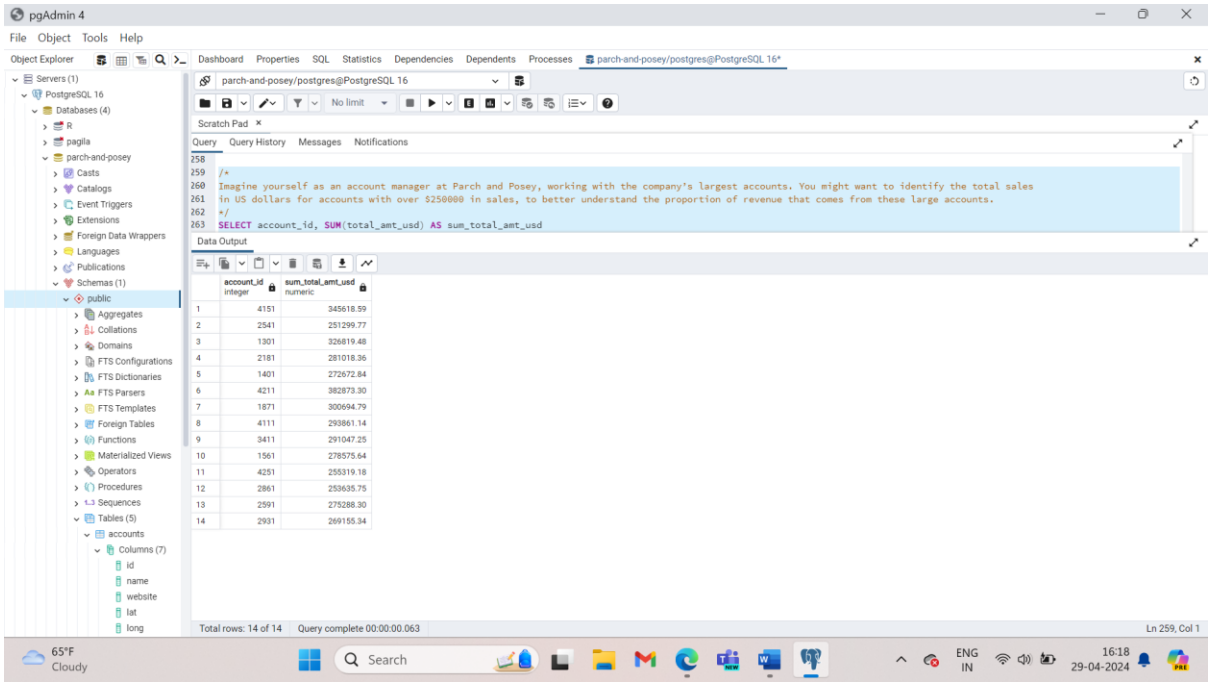
Imagine yourself as an account manager at Parch and Posey, working with the company's largest accounts. You might want to identify the total sales in US dollars for accounts with over $250000 in sales, to better understand the proportion of revenue that comes from these large accounts.

SELECT account_id, SUM(total_amt_usd) AS sum_total_amt_usd

FROM orders

GROUP BY 1

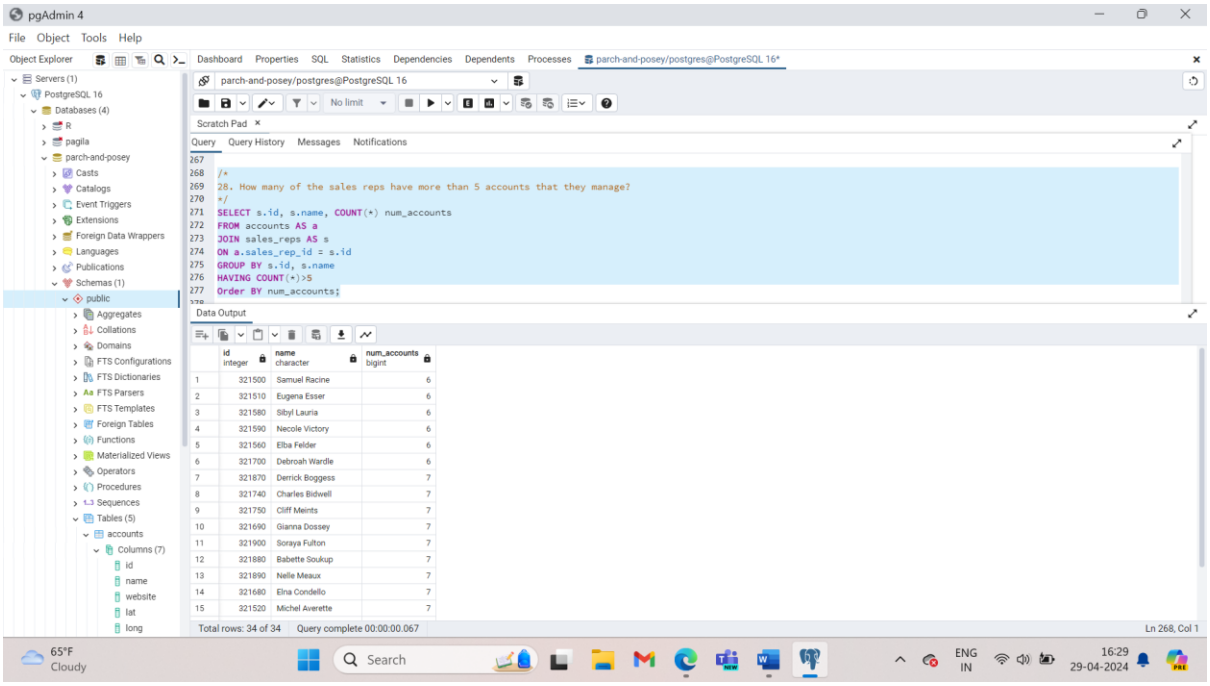
HAVING SUM(total_amt_usd) >= 250000;



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.

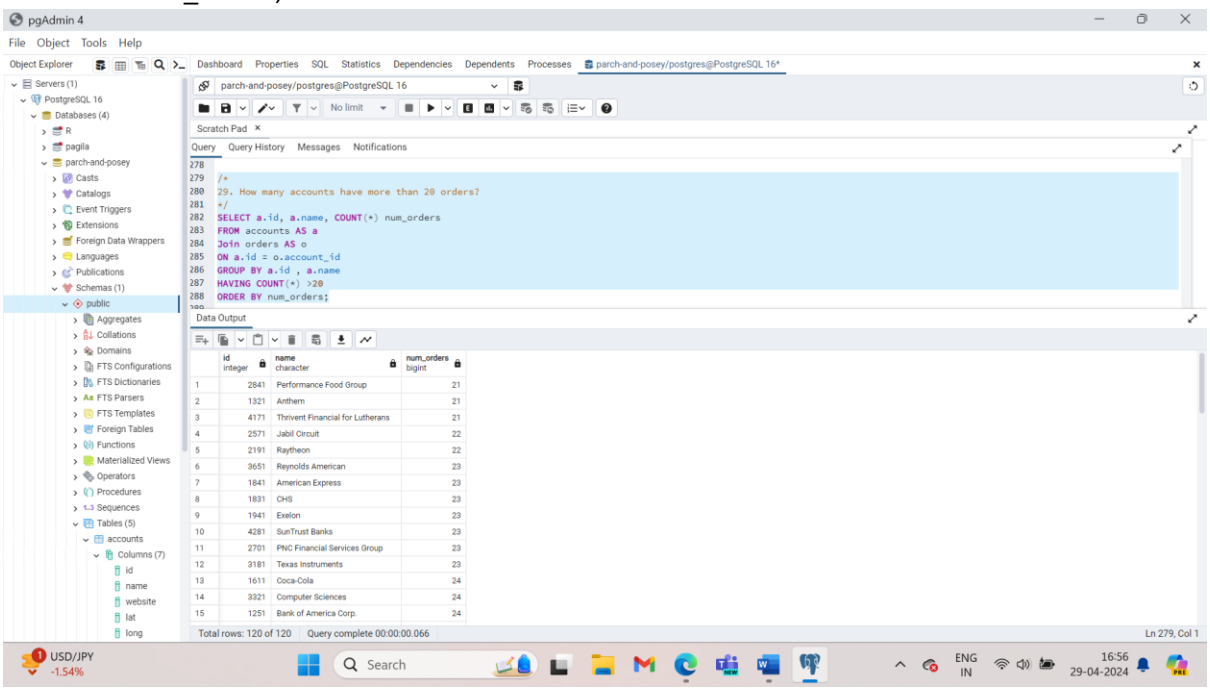
28. How many of the sales reps have more than 5 accounts that they manage?

```
SELECT s.id, s.name, COUNT(*) num_accounts
FROM accounts AS a
JOIN sales_reps AS s
ON a.sales_rep_id = s.id
GROUP BY s.id, s.name
HAVING COUNT(*)>5
Order BY num_accounts;
```



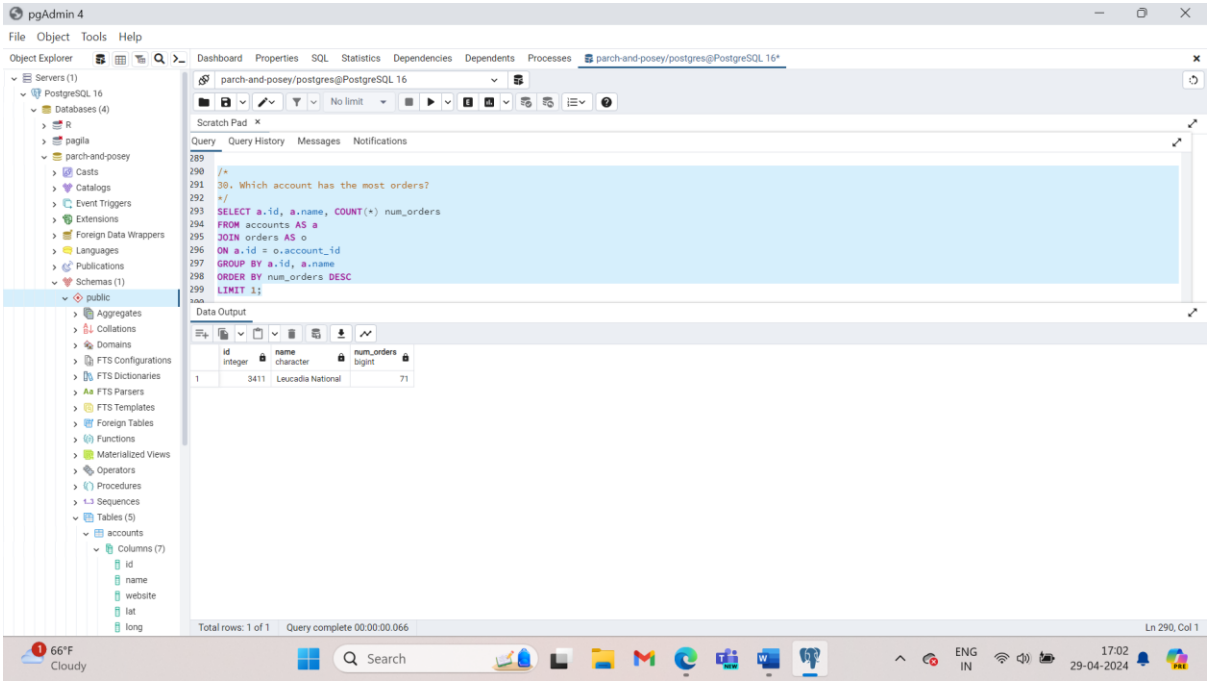
29. How many accounts have more than 20 orders?

```
SELECT a.id, a.name, COUNT(*) num_orders
FROM accounts AS a
Join orders AS o
ON a.id = o.account_id
GROUP BY a.id , a.name
HAVING COUNT(*) >20
ORDER BY num_orders;
```



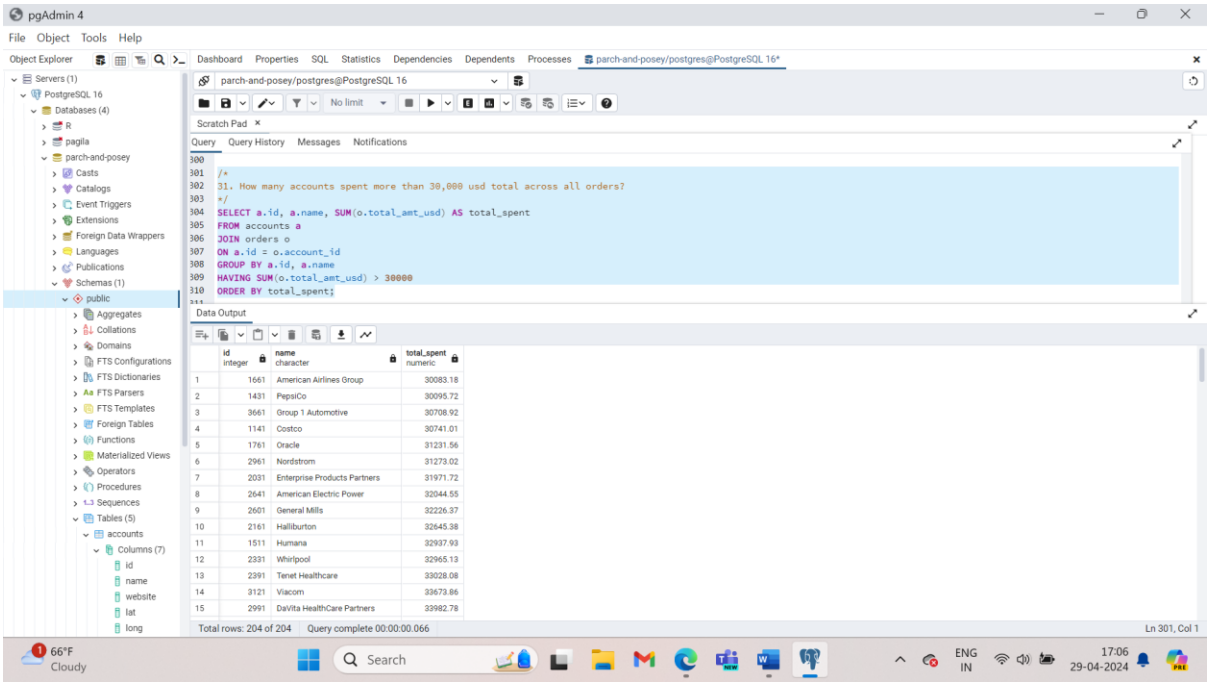
30. Which account has the most orders?

```
SELECT a.id, a.name, COUNT(*) num_orders
FROM accounts AS a
JOIN orders AS o
ON a.id = o.account_id
GROUP BY a.id, a.name
ORDER BY num_orders DESC
LIMIT 1;
```



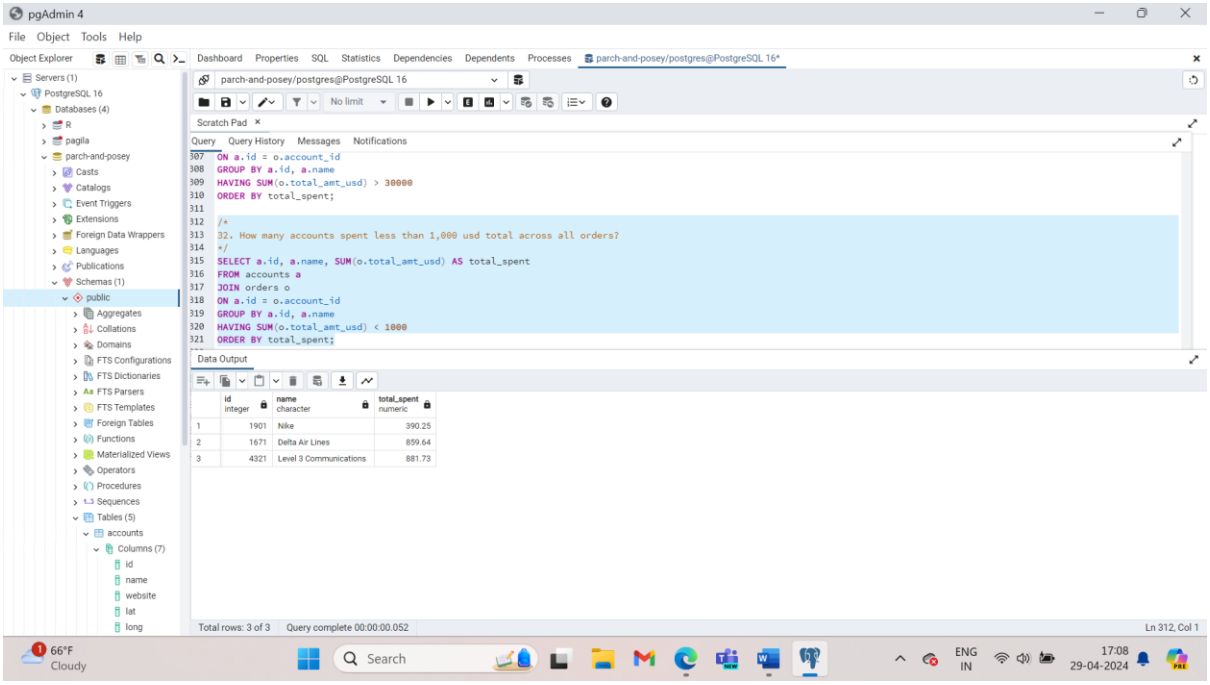
31. How many accounts spent more than 30,000 usd total across all orders?

```
SELECT a.id, a.name, SUM(o.total_amt_usd) AS total_spent
FROM accounts a
JOIN orders o
ON a.id = o.account_id
GROUP BY a.id, a.name
HAVING SUM(o.total_amt_usd) > 30000
ORDER BY total_spent;
```



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

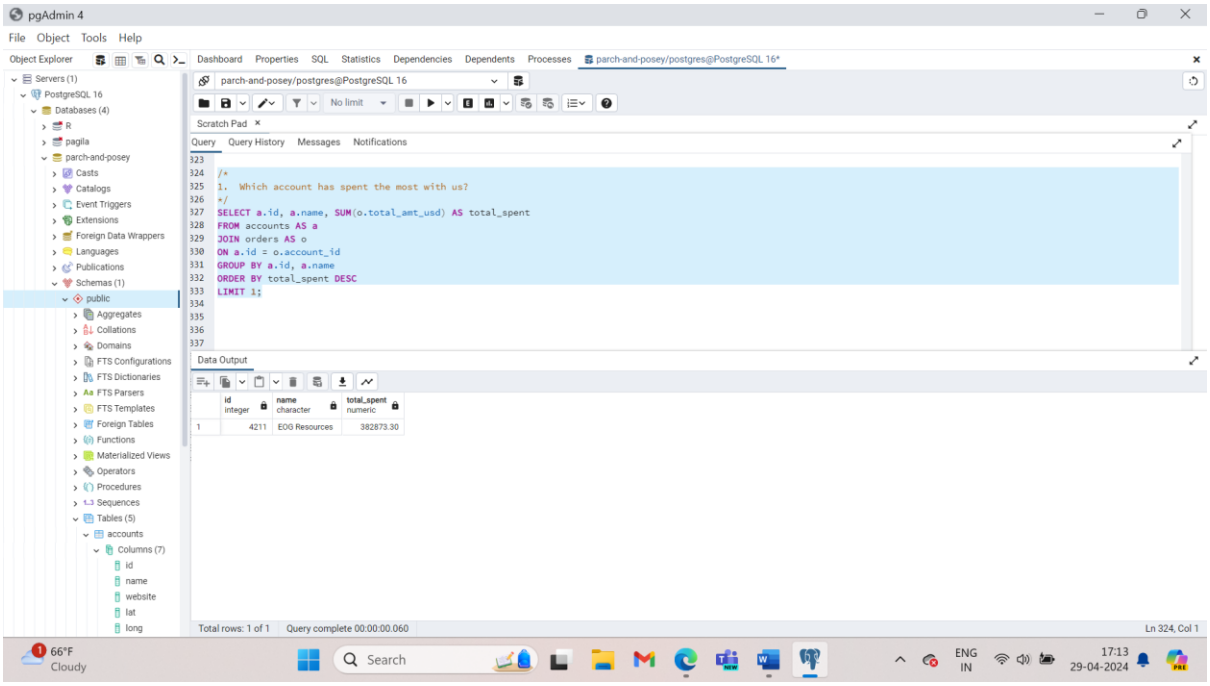
```
SELECT a.id, a.name, SUM(o.total_amt_usd) AS total_spent
FROM accounts a
JOIN orders o
ON a.id = o.account_id
GROUP BY a.id, a.name
HAVING SUM(o.total_amt_usd) < 1000
ORDER BY total_spent;
```



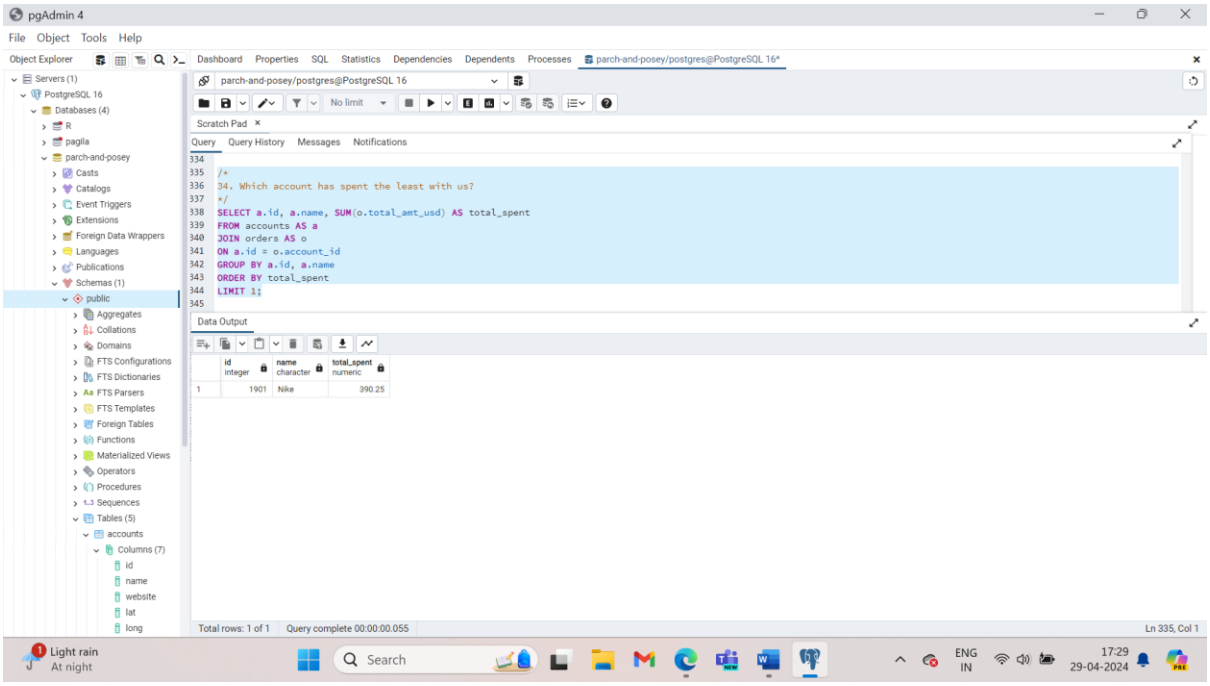
33. Which account has spent the most with us?

```
SELECT a.id, a.name, SUM(o.total_amt_usd) AS total_spent
FROM accounts AS a
```

JOIN orders AS o
ON a.id = o.account_id
GROUP BY a.id, a.name
ORDER BY total_spent DESC
LIMIT 1;

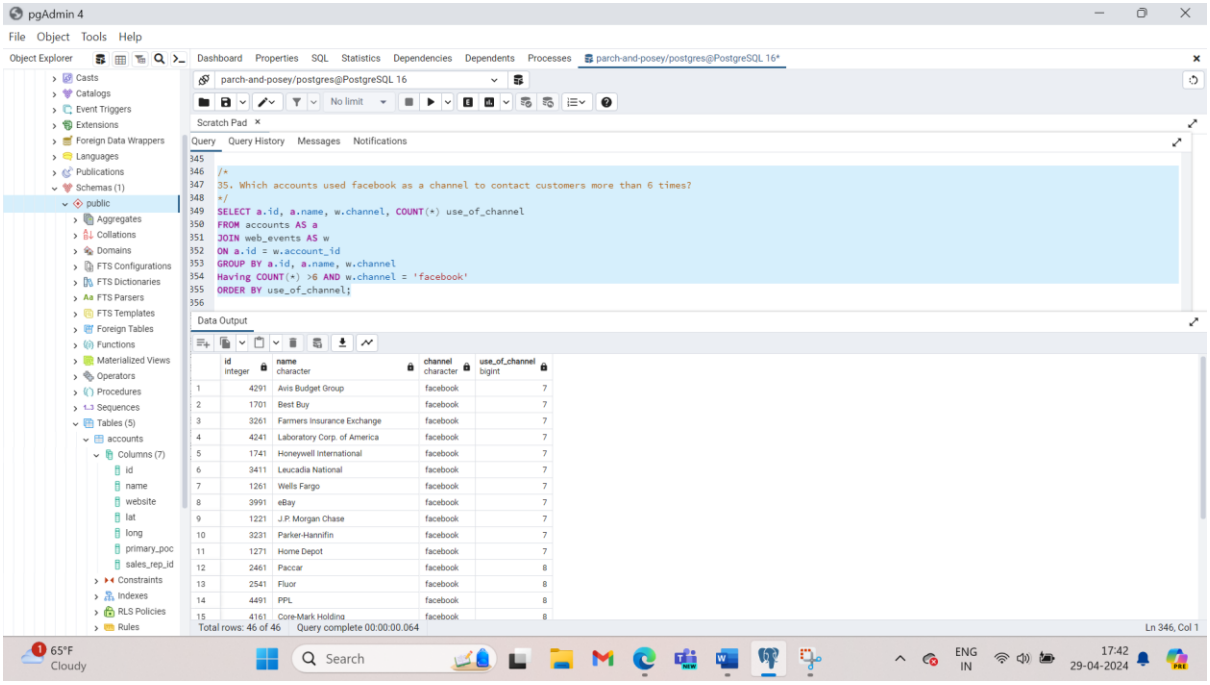


34. Which account has spent the least with us?
SELECT a.id, a.name, SUM(o.total_amt_usd) AS total_spent
FROM accounts AS a
JOIN orders AS o
ON a.id = o.account_id
GROUP BY a.id, a.name
ORDER BY total_spent
LIMIT 1;



35. Which accounts used facebook as a channel to contact customers more than 6 times?

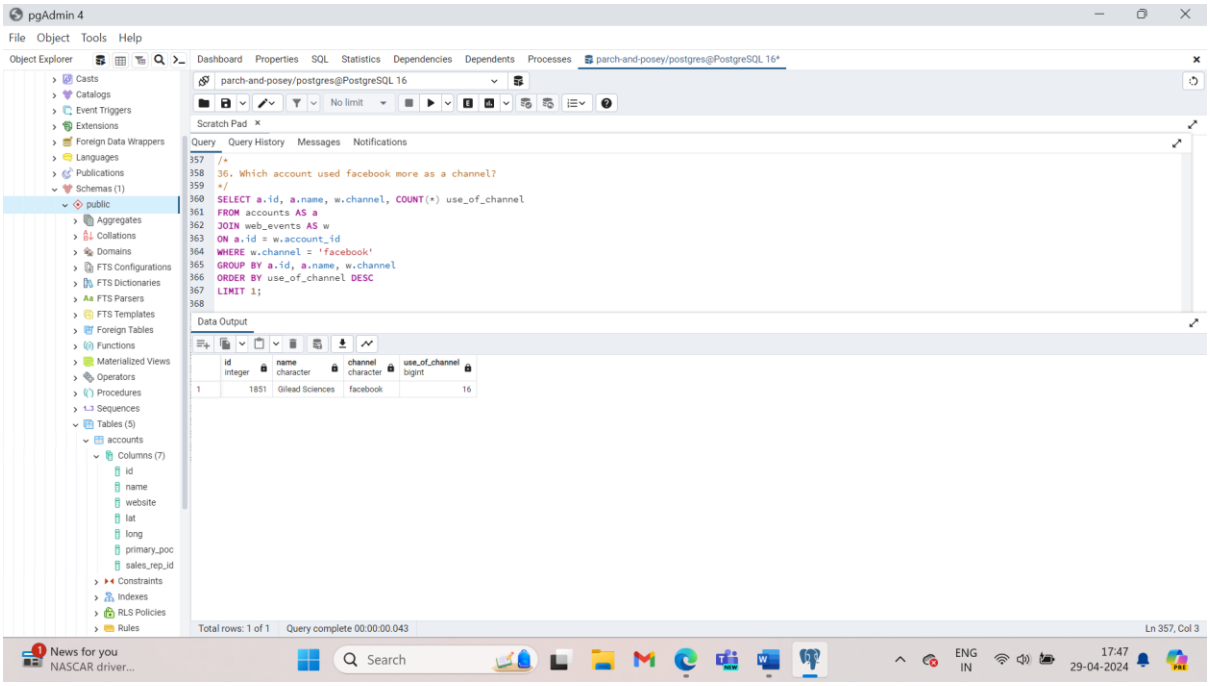
```
SELECT a.id, a.name, w.channel, COUNT(*) use_of_channel
FROM accounts AS a
JOIN web_events AS w
ON a.id = w.account_id
GROUP BY a.id, a.name, w.channel
Having COUNT(*) >6 AND w.channel = 'facebook'
ORDER BY use_of_channel;
```



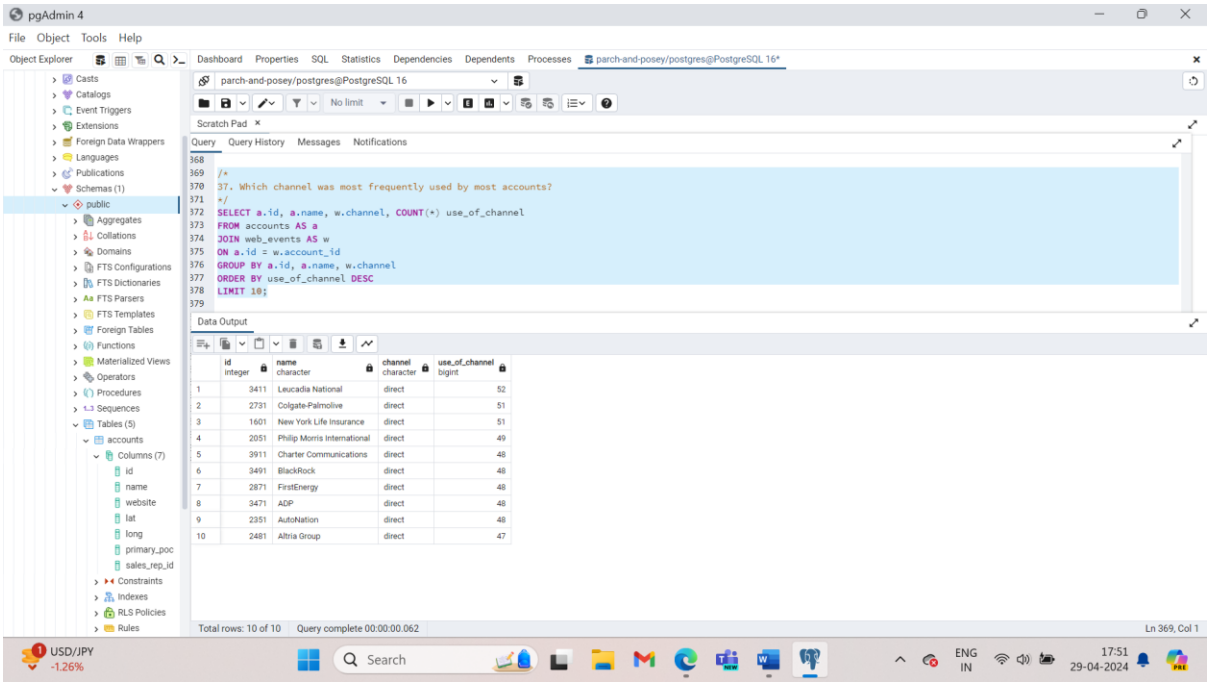
36. Which account used facebook more as a channel?

```
SELECT a.id, a.name, w.channel, COUNT(*) use_of_channel
FROM accounts AS a
JOIN web_events AS w
```

ON a.id = w.account_id
WHERE w.channel = ‘facebook’
GROUP BY a.id, a.name, w.channel
ORDER BY use_of_channel DESC
LIMIT 1;

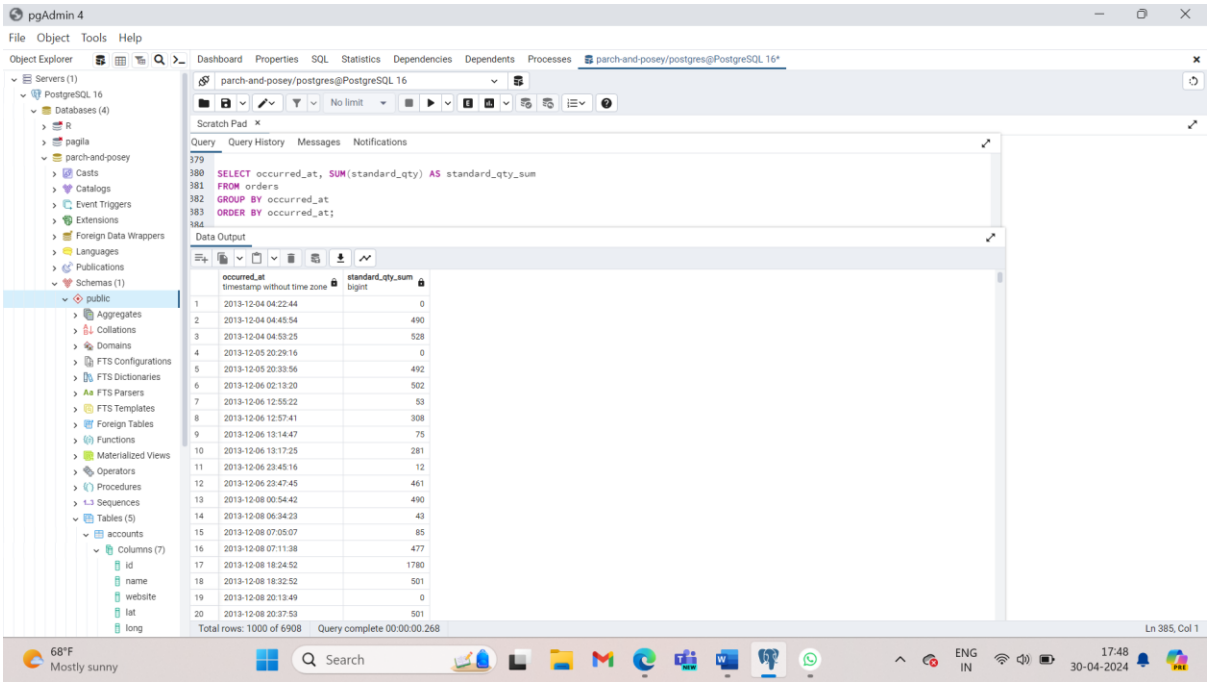


37. Which channel was most frequently used by most accounts?
SELECT a.id, a.name, w.channel, COUNT(*) use_of_channel
FROM accounts AS a
JOIN web_events AS w
ON a.id = w.account_id
GROUP BY a.id, a.name, w.channel
ORDER BY use_of_channel DESC
LIMIT 10;



DATE Functions

By now you might have noticed that dates are a bit hard to work with. Aggregating by date fields, doesn't work in a practical way. It treats each time stamp as unique. When it would be more practical to round to a particular day, week, or even 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 looking at the raw data.



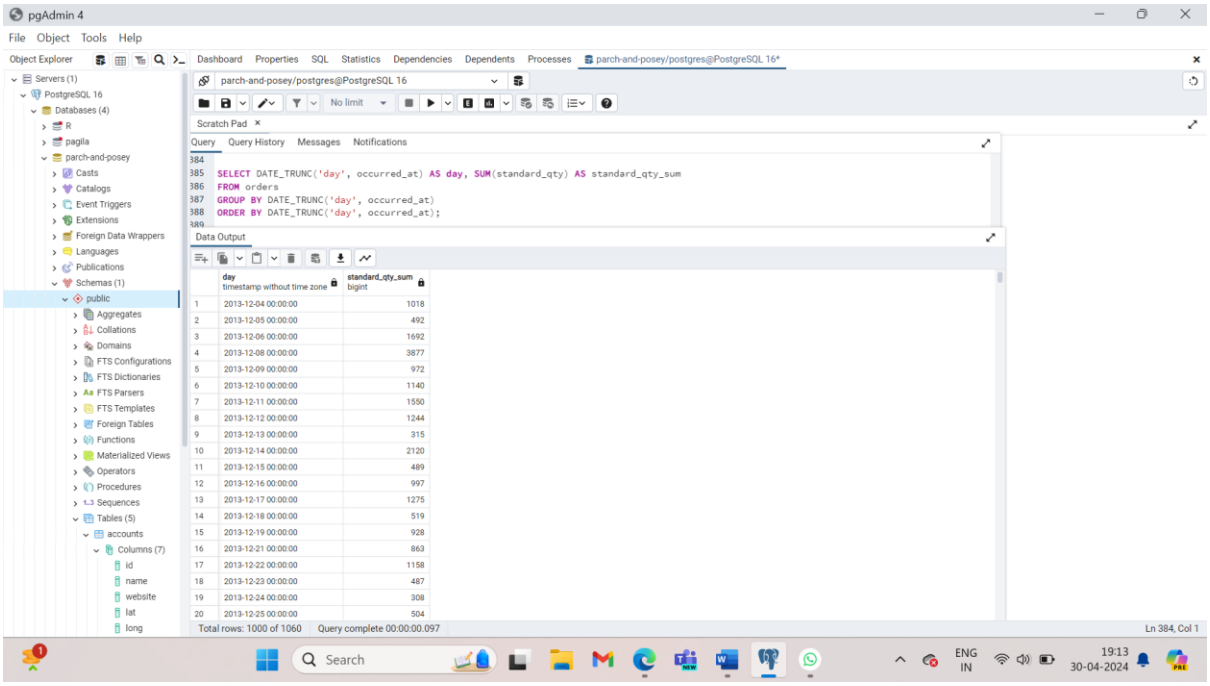
This aggregates from six 6,912 rows in the raw data down to 6908. Almost all of the data in this table are unique. Good news though: there are plenty of special functions to make dates easier to work with. But before we dig into date and time functions, Lets take a look at how dates are stored. If you live in US, you're probably used to seeing dates formatted as MM-DD-YYYY. It is not a convention compared to the rest of the world's standard. Most of the places follow DD-MM-YYYY format. This isn't necessarily better or worse, it's just different.

Databases do it yet another way. Ordering from the least to the most granular part of the date, YYYY-MM-DD. This is a very specific utility, and basis of that date sorted alphabetically are also in chronological order. In other words date ordering is the same whether you think of them as date or as bits of text. With this example here, we can see that the year first way the database stores dates, is ideal for sorting the way we'll want to retrieve this information in the future. Whether we want the most recent of oldest information, day first and month first date formats sort of funny ways that don't make a ton of sense. Another benefit id the date can easily be truncated in order to group them for analysis. Take the date 2017-04-01 12:15:01. If you want to group that with other events that occurred on the same day, we can't do that with the date in its current format. Group by every event that occurred on April 1st, at 12:15 and 1 second. That won't help us very much.

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.

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 by using **DATE_TRUNC** function.
In the previous query we saw that grouping by occurred_at doesn't help us much. If we replace each instance of occurred at with a truncated version, we'll get a result set that sums the quantities of standard paper by day.

```
SELECT DATE_TRUNC('day', occurred_at) AS day, SUM(standard_qty) AS standard_qty_sum
FROM orders
GROUP BY DATE_TRUNC('day', occurred_at)
ORDER BY DATE_TRUNC('day', occurred_at);
```



DATE_TRUNC is useful even when you need the data at a granular level like seconds. The use case for this is when we are working on server logs where multiple events are happening in a given 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,year.

There may be times where you might want to just pull out a given part of the day. For example, if you want to know what day of the week Parch and Posey's website sees the most traffic, you wouldn't want to use DATE_TRUNC. To get the day of the week, you'd have to use DATE_PART. Notice that a DATE_PART would provide the same month for an event that happens in April 2016 and April 2017 where a date trunc would differentiate these events.

Let's explore this example using Parch and Posey's data.

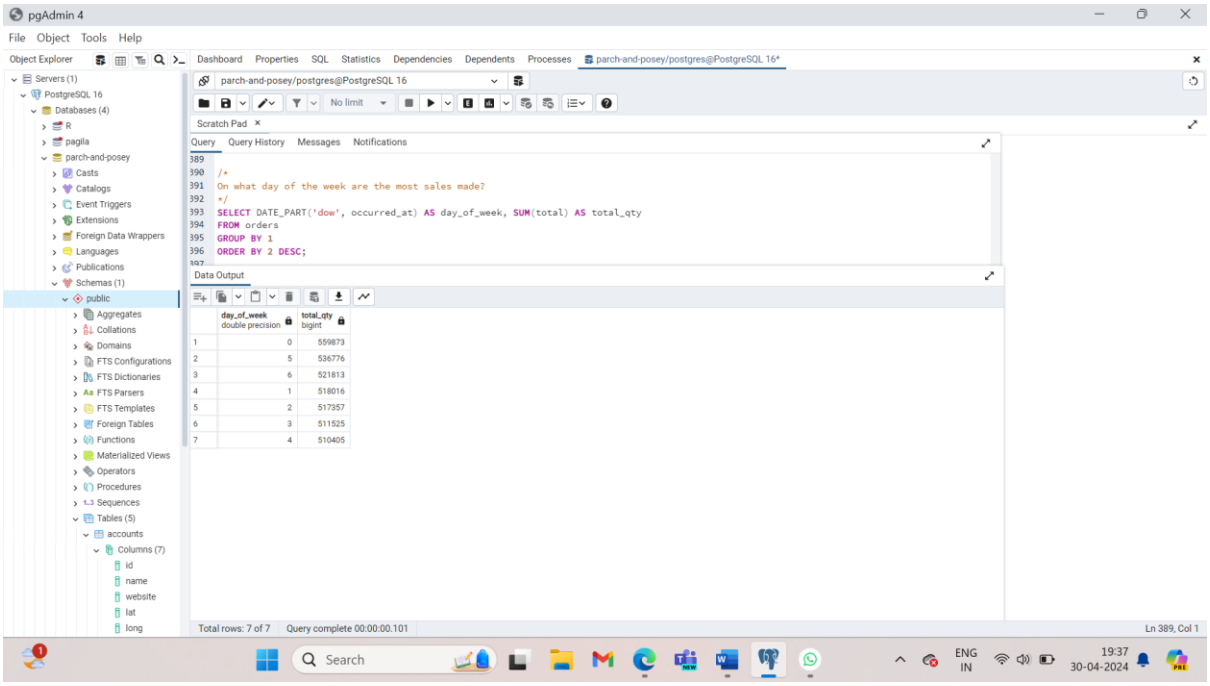
On what day of the week are the most sales made?

SELECT DATE_PART('dow', occurred_at) AS day_of_week, account_id, occurred_at, total
FROM orders;

'dow' is day of week which returns a value from 0 to 6, where 0 is Sunday and 6 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 with the sum in descending order where the day with the most sales will be at the top of the result set.

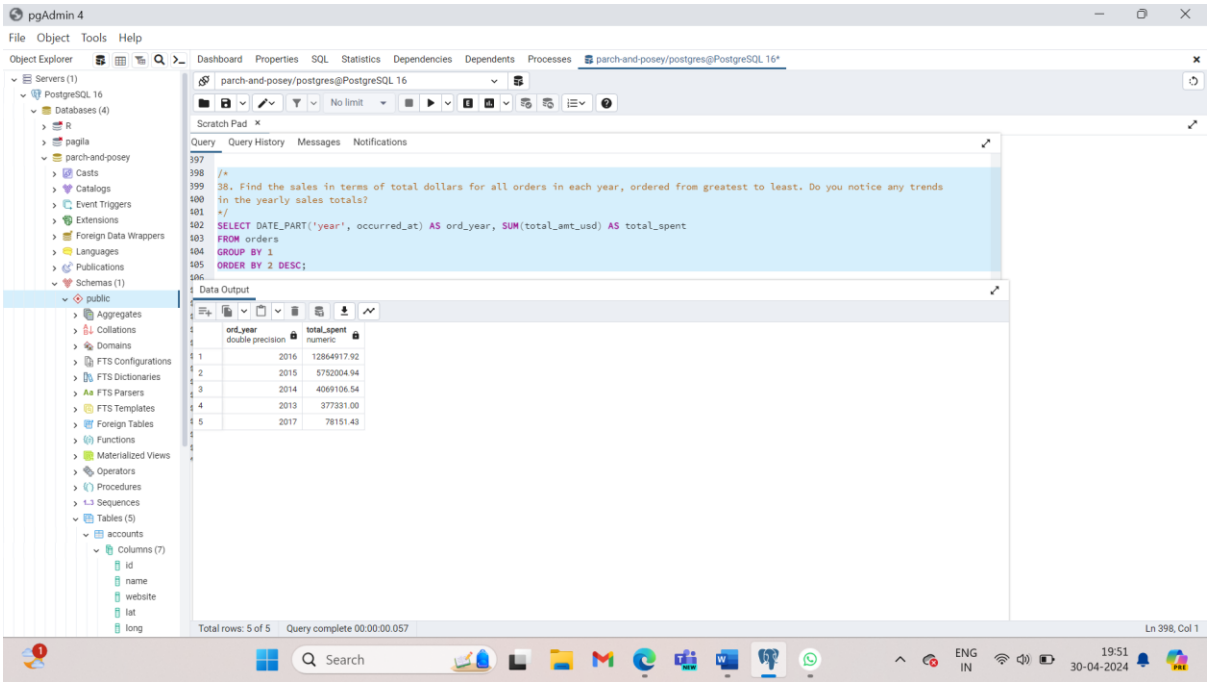
SELECT DATE_PART('dow', occurred_at) AS day_of_week, SUM(total) AS total_qty
FROM orders
GROUP BY 1
ORDER BY 2 DESC;



DATE_TRUNC allows you to truncate your date to a particular part of your date-time column. Common truncations are day, month, and year. **Here(opens in a new tab)** is a great blog post by Mode Analytics on the power of this function.
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.

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)*

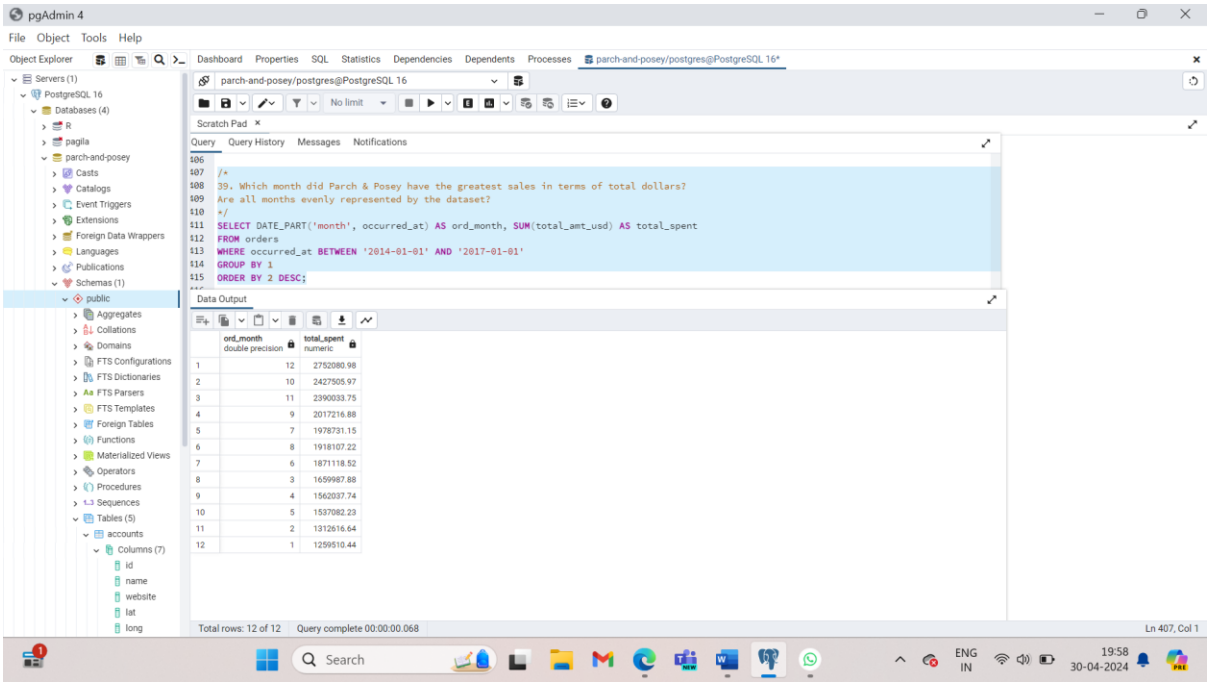
38. Find the sales in terms of total dollars for all orders in each year, ordered from greatest to least. Do you notice any trends in the yearly sales totals?
SELECT DATE_PART('year', occurred_at) AS ord_year, SUM(total_amt_usd) AS total_spent
FROM orders
GROUP BY 1
ORDER BY 2 DESC;



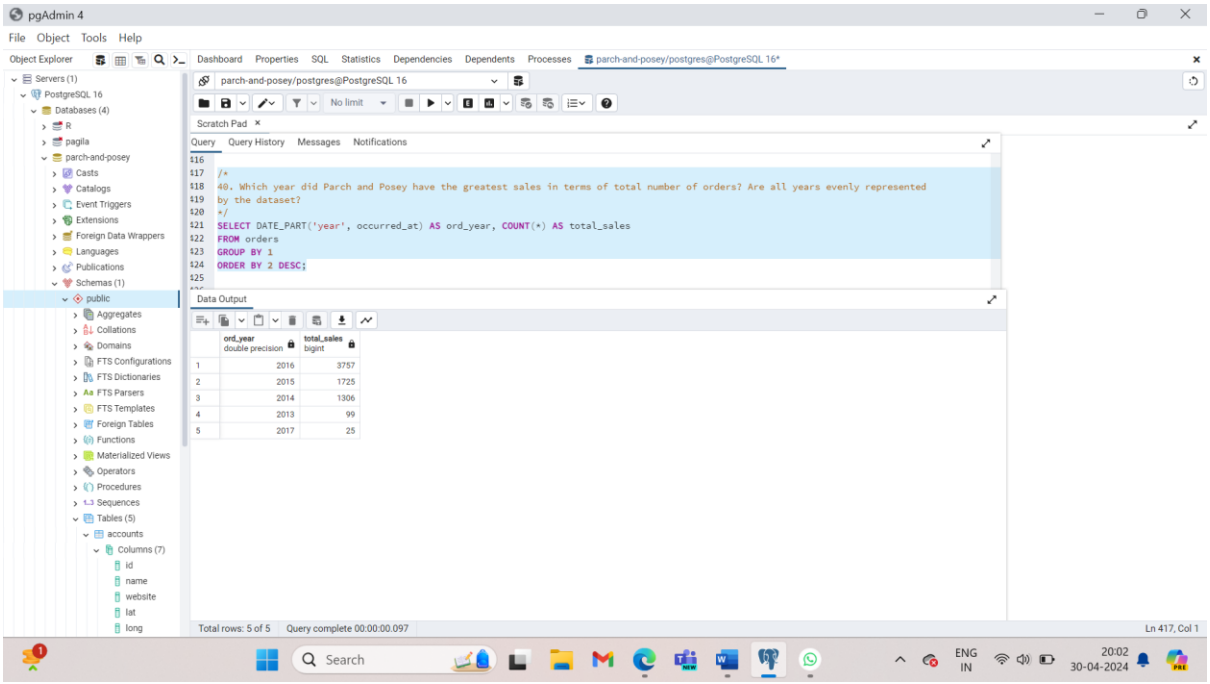
When we look at the yearly totals, you might notice that 2013 and 2017 have much smaller totals than all other years. If we look further at the monthly data, we see that for 2013 and 2017 there is only one month of sales for each of these years (12 for 2013 and 1 for 2017). Therefore, neither of these 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.

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

SELECT DATE_PART('month', occurred_at) AS ord_month, SUM(total_amt_usd) AS total_spent
FROM orders
WHERE occurred_at BETWEEN '2014-01-01' AND '2017-01-01'
GROUP BY 1
ORDER BY 2 DESC;

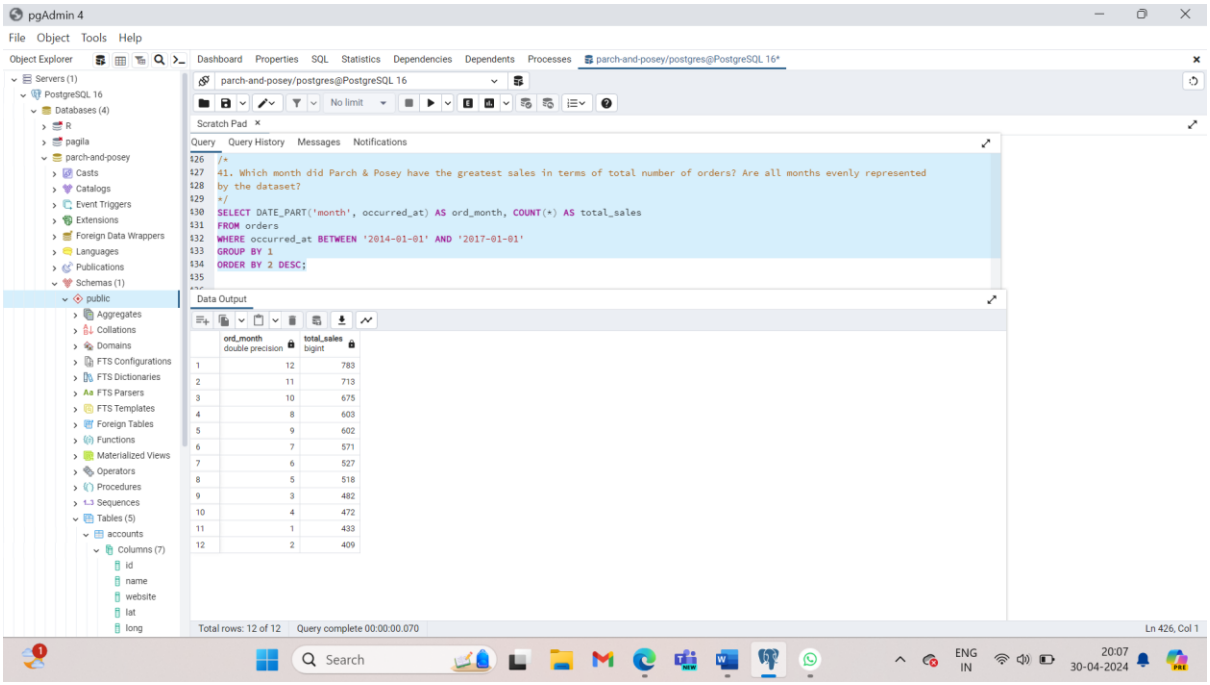


40. Which year did Parch and Posey have the greatest sales in terms of total number of orders? Are all years evenly represented by the dataset?
SELECT DAT_PART('year', occurred_at) AS ord_year, COUNT(*) AS total_sales
FROM orders
GROUP BY 1
ORDER BY 2 DESC;

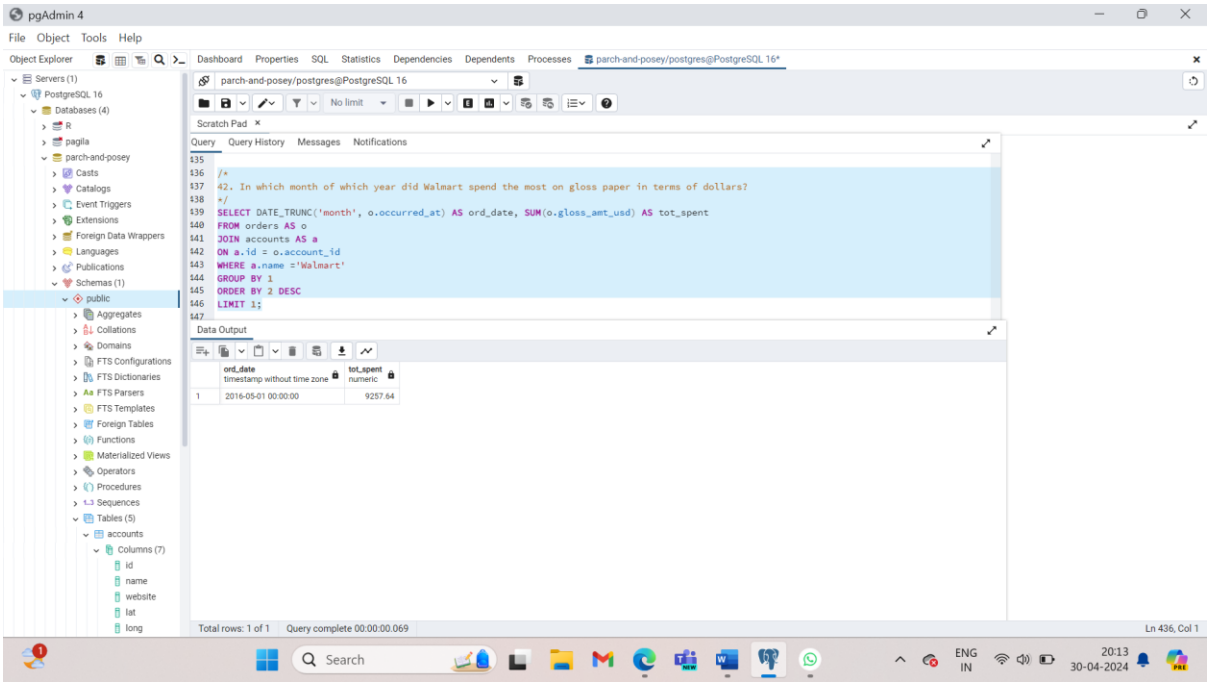


41. Which **month** did Parch & Posey have the greatest sales in terms of total number of orders? Are all months evenly represented by the dataset?
SELECT DATE_PART('month', occurred_at) AS ord_month, COUNT(*) AS total_sales
FROM orders
WHERE occurred_at BETWEEN '2014-01-01' AND '2017-01-01'

GROUP BY 1
ORDER BY 2 DESC;

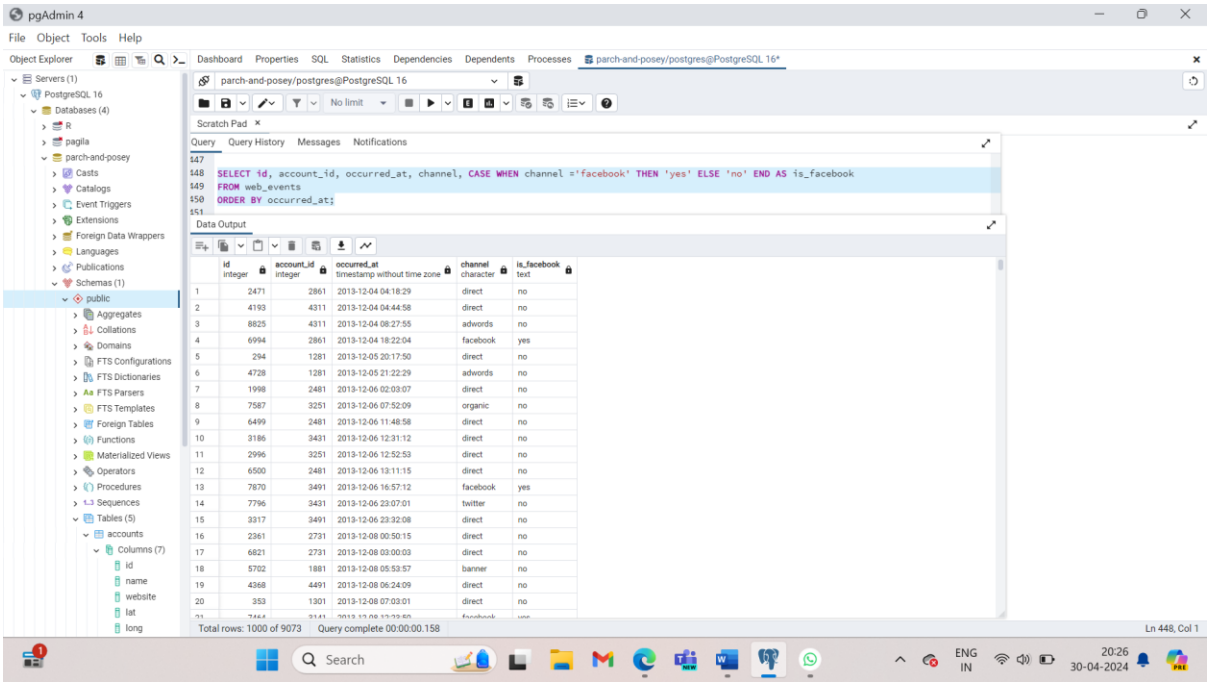


42. In which month of which year did Walmart spend the most on gloss paper in terms of dollars?
SELECT DATE_TRUNC('month', o.occurred_at) AS ord_date, SUM(o.gloss_amt_usd) AS tot_spent
FROM orders AS o
JOIN accounts AS o
ON a.id = o.account_id
WHERE a.name ='Walmart'
GROUP BY 1
ORDER BY 2 DESC
LIMIT 1;

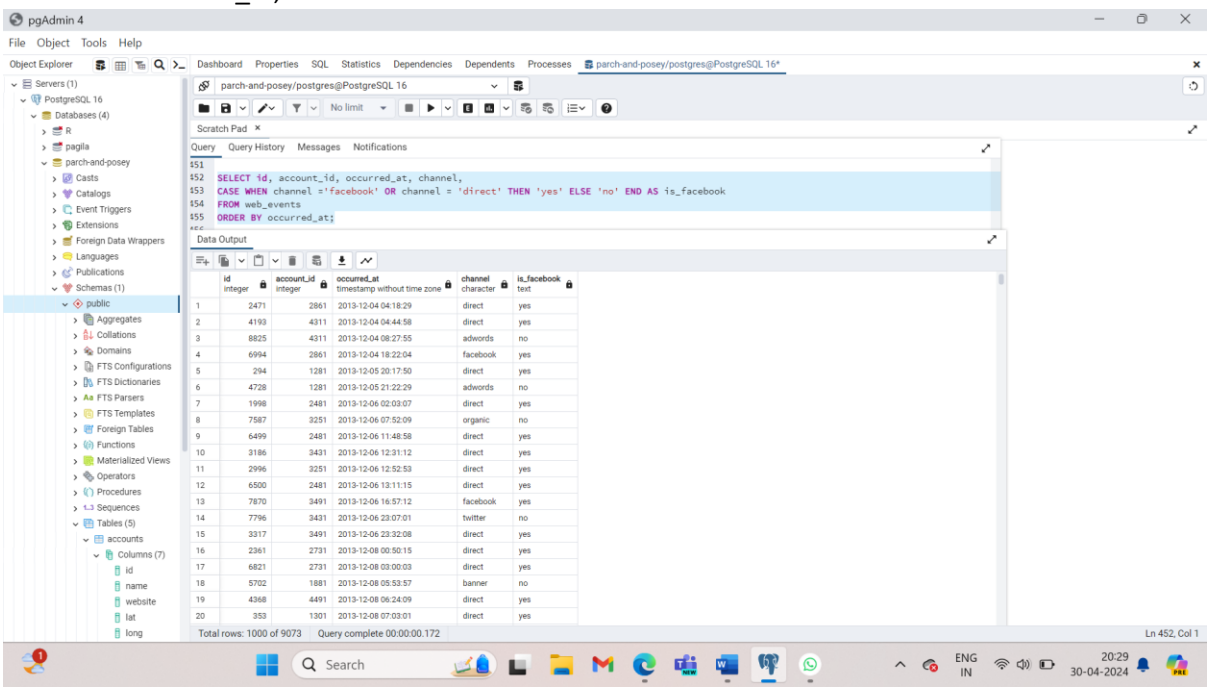


Imagine yourself as the marketing manager at Parch and Posey. You want to compare Facebook as a marketing channel against all other channels. You know facebook is good for your business but is it better than the rest combined? Let's find out. In order to do this, you'll need to create a derived column. That means you'll need to take data from existing columns and modify it. In previous lessons we did this using arithmentic. Here we do it using the 'CASE' statement which is sequel's way of handling 'IF''then' logic. The case statement is followed by atleast one pair of 'WHEN ' and 'Then' statements. It must finish with the word 'END'.

SELECT id, account_id, occurred_at, channel, CASE WHEN channel ='facebook' THEN 'yes'
END AS is_facebook
FROM web_events
ORDER BY occurred_at;



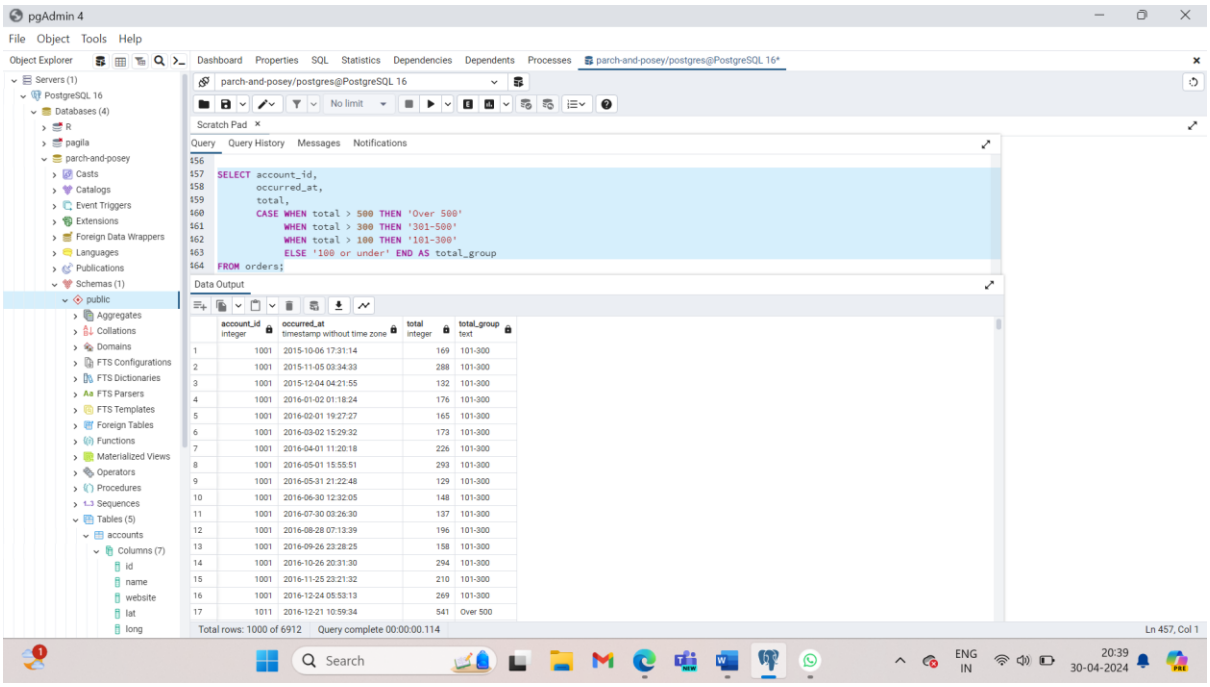
SELECT id, account_id, occurred_at, channel,
CASE WHEN channel =’facebook’ OR channel = ‘direct’ THEN ‘yes’ END AS is_facebook
FROM web_events
ORDER BY occurred_at;



Imagine yourself in operations at Parch and Posey. You’d like to classify orders into general groups based on order size to get more granular about inventory planning. You can use a case statement to define a number of outcomes by including as many ‘WHEN’ ‘THEN’ statements as you’d like.

SELECT account_id, occurred_at, total,
CASE WHEN total > 500 THEN ‘Over 500’
WHEN total >300 THEN ‘301-500’

WHEN total > 100 THEN ‘101-300’
ELSE ‘100 or under’ END AS total_group
FROM orders;



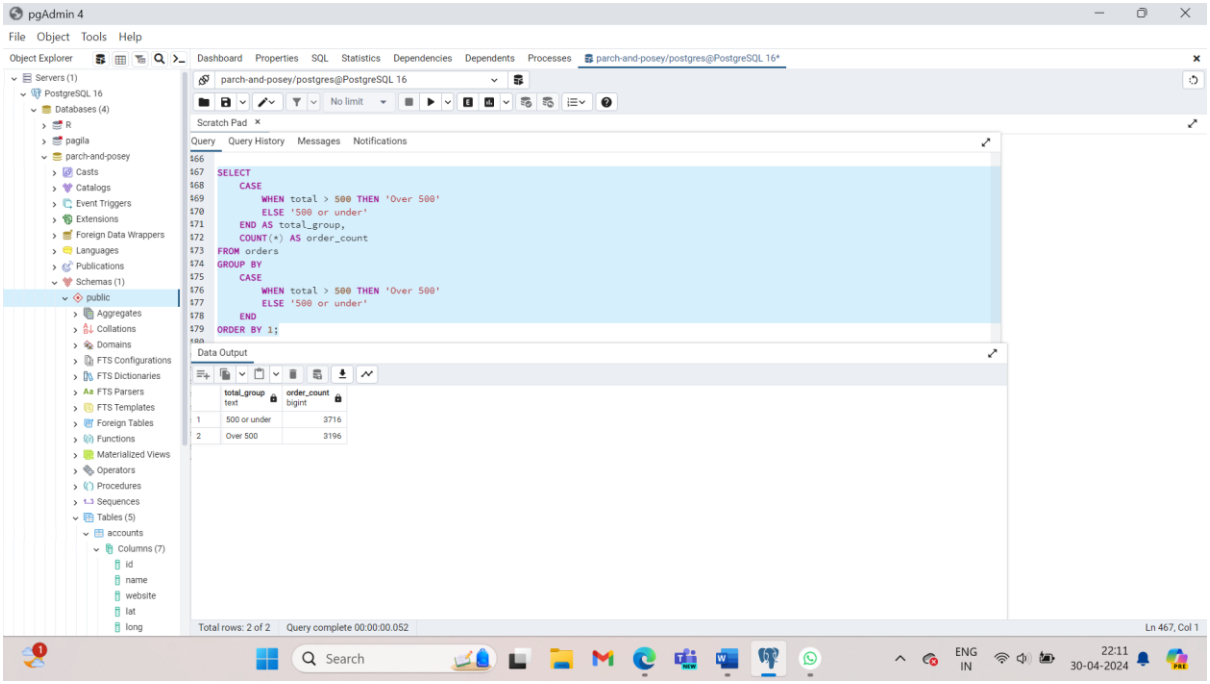
A better way to write this query is to remove overlapping parts:
SELECT account_id, occurred_at, total,
CASE WHEN total > 500 THEN ‘Over 500’
WHEN total >300 AND total <=500 THEN ‘301-500’
WHEN total > 100 AND total <=300 THEN ‘101-300’
ELSE ‘100 or under’ END AS total_group
FROM orders;

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.

SELECT account_id, CASE WHEN standard_qty =0 OR standard_qty IS NULL THEN 0
ELSE standard_amt_usd/ standard_qty END AS unit_price
FROM orders
LIMIT 10;

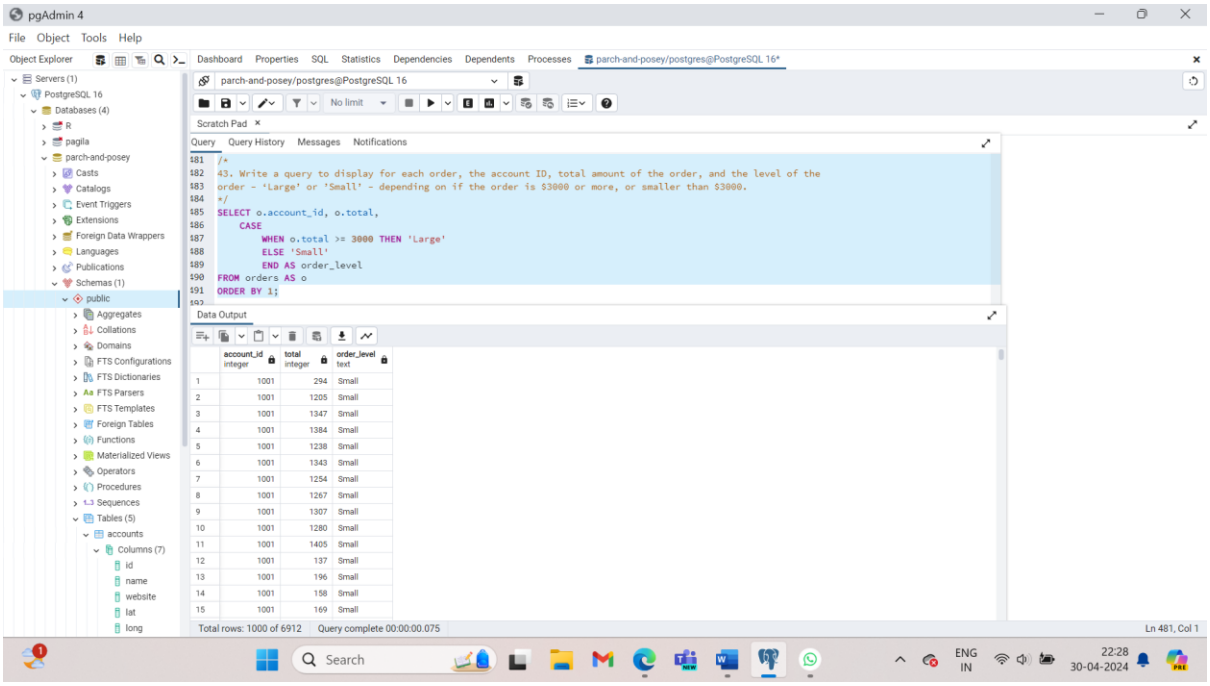
As a sales manager at Parch and Posey, classifying orders into general groups is a helpful exercise. Butit’s much more useful if you can let it count up all the orders in each group. Aggregating based on these new categories will make it easier to report back to company leaders and take action. The easiest way to count all the members of a group is to create a column that groups the way you want it to, then create another column to count by that group. Here we are using CASE to group orders into those with total quantity sold over 500, and those with 500 or less.

```
SELECT
   CASE
      WHEN total > 500 THEN 'Over 500'
      ELSE '500 or under'
   END AS total_group,
   COUNT(*) AS order_count
FROM orders
GROUP BY
   CASE
      WHEN total > 500 THEN 'Over 500'
      ELSE '500 or under'
   END
ORDER BY 1;
```



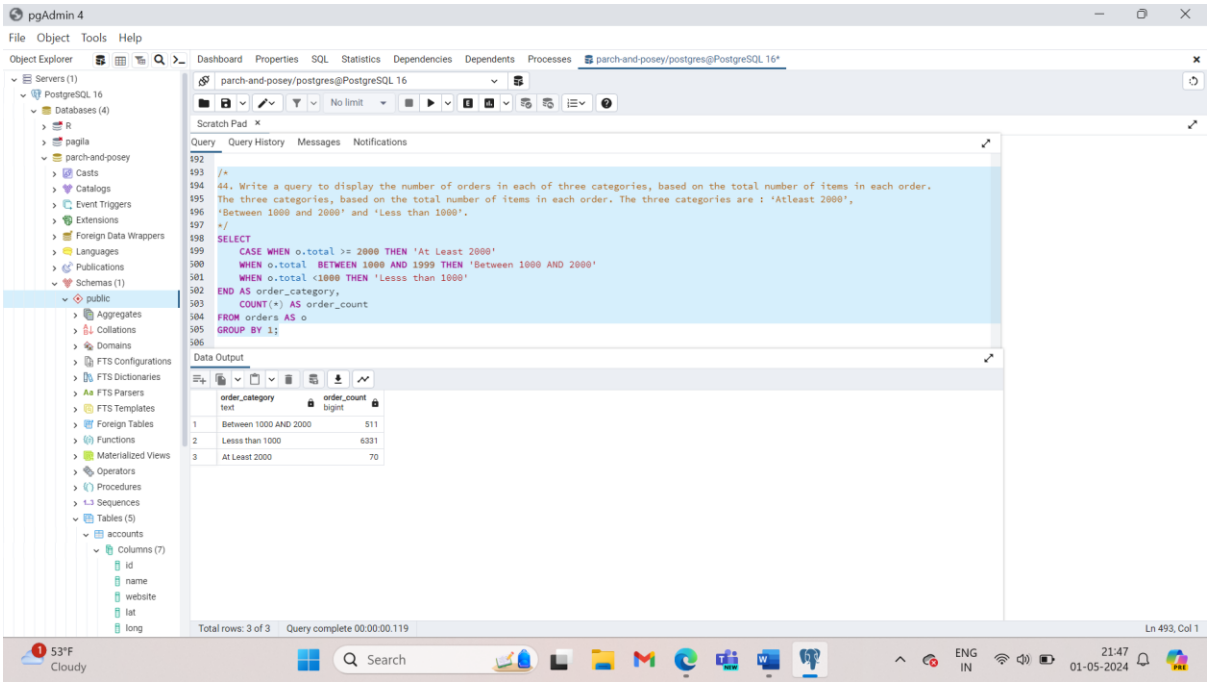
43. Write a query to display for each order, the account ID, total amount of the order, and the level of the order - ‘Large’ or ’Small’ - depending on if the order is $3000 or more, or smaller than $3000.

```
SELECT o.account_id, o.total,
            CASE WHEN o.total >= 3000 THEN ‘Large’
            ELSE ‘Small’
            END AS order_level
FROM orders AS o
ORDER BY 1;
```



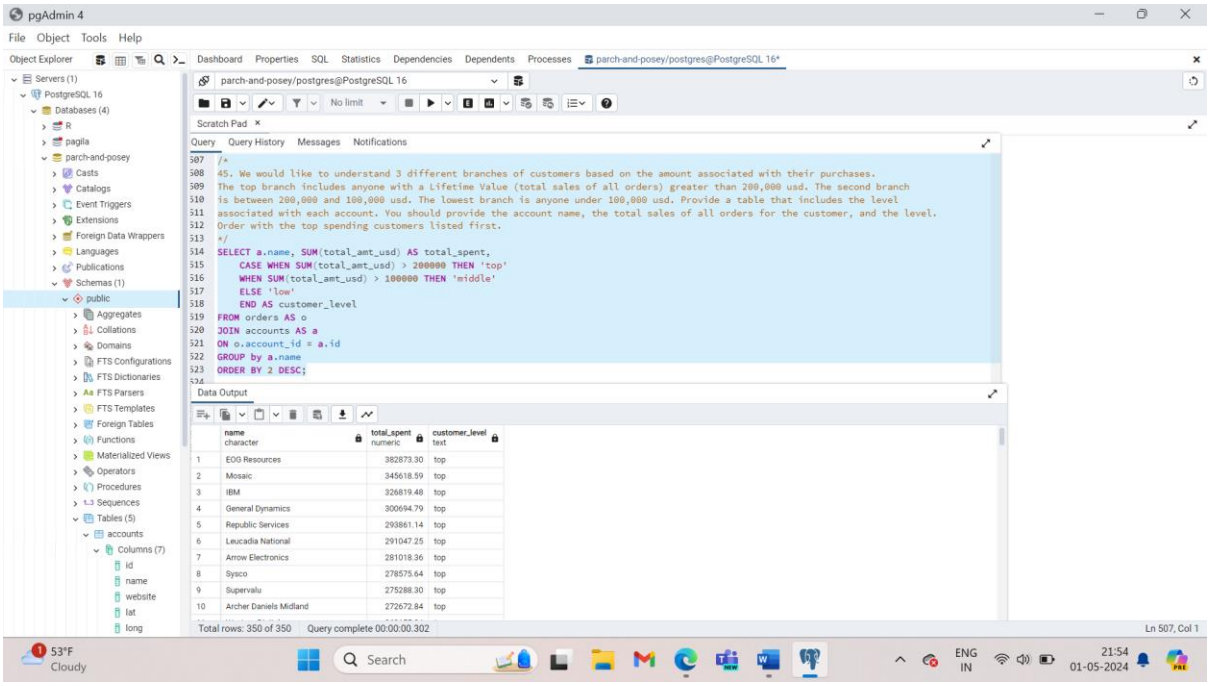
44. Write a query to display the number of orders in each of three categories, based on the total number of items in each order. The three categories, based on the total number of items in each order. The three categories are : 'Atleast 2000', 'Between 1000 and 2000' and 'Less than 1000'.

```
SELECT
        CASE WHEN o.total >= 2000 THEN 'At Least 2000'
        WHEN o.total  BETWEEN 1000 AND 1999 THEN 'Between 1000 AND 2000'
        WHEN o.total <1000 THEN 'Lesss than 1000'
        END AS order_category,

COUNT(*) AS order_count

FROM orders AS o

GROUP BY 1;
```

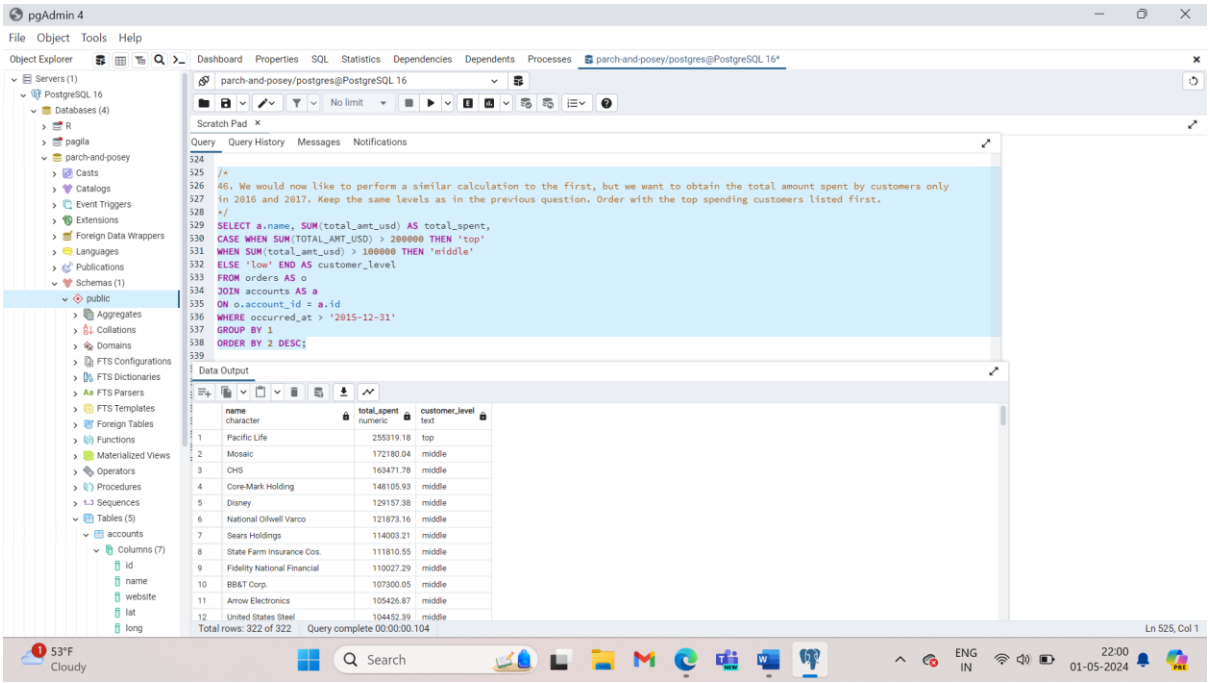


45. We would like to understand 3 different branches of customers based on the amount associated with their purchases. The top branch includes anyone with a Lifetime Value (total sales of all orders) greater than 200,000 usd. The second branch is between 200,000 and 100,000 usd. The lowest branch is 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.

```
SELECT a.name, SUM(total_amt_usd) AS total_spent,
        CASE WHEN SUM(total_amt_usd) > 200000 THEN 'top'
        WHEN SUM(total_amt_usd) > 100000 THEN 'middle'
        ELSE 'low'
        END AS customer_level
FROM order AS o
JOIN accounts AS a
ON o.account_id = a.id
GROUP by a.name
ORDER BY 2 DESC;
```

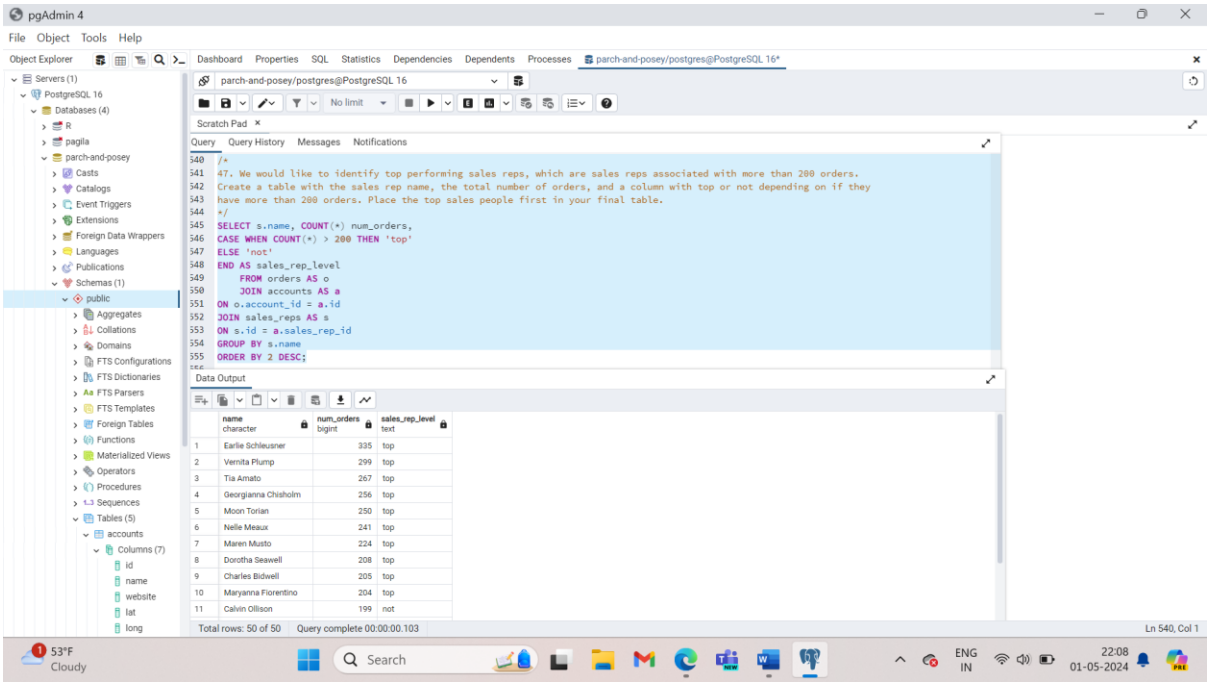


46. We would now like to perform a similar calculation to the first, but we want to obtain the total amount spent by customers only in 2016 and 2017. Keep the same levels as in the previous question. Order with the top spending customers listed first.
SELECT a.name, SUM(total_amt_usd) AS total_spent,
CASE WHEN SUM(TOTAL_AMT_USD) > 200000 THEN ‘top’
WHEN SUM(total_amt_usd) > 100000 THEN ‘middle’
ELSE ‘low’ END AS customer_level
FROM orders AS o
JOIN accounts AS a
ON o.account_id = a.id
WHERE occurred_at > ‘2015-12-31’
GROUP BY 1
ORDER BY 2 DESC;



47. 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.

```
SELECT s.name, COUNT(*) num_orders,
        CASE WHEN COUNT(*) > 200 THEN 'top'
        ELSE 'not'
        END AS sales_rep_level

FROM orders AS o

JOIN accounts AS a

ON o.account_id = a.id

JOIN sales_reps AS s

ON s.id = a.sales_rep_id

GROUP BY s.name

ORDER BY 2 DESC;
```



48. 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.

```
SELECT s.name, COUNT(*), SUM(o.total_amt_usd) total_spent,
        CASE WHEN COUNT(*) > 200 OR SUM(o.total_amt_usd) > 750000 THEN 'top'
        WHEN COUNT(*) > 150 OR SUM(o.total_amt_usd) > 750000 THEN 'middle'
```

```
        ELSE 'low' END AS sales_rep_level
FROM orders o
JOIN accounts a
ON o.account_id = a.id
Join sales_reps s
ON s.id = a.sales_rep_id
GROUP BY s.name
ORDER BY 3 DESC;
```