

## 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;
```

The screenshot shows a SQL query editor with the following details:

- Toolbar: Run, Limit 100, Format SQL, View History...
- Query pane:

```
1 SELECT *
2 FROM demo.accounts
3 WHERE id > 1500 AND id < 1600
```
- Status bar: ✓ Succeeded in 2s
- Results pane:

	id	name	website	lat	long	primary poc	sales_rep_id
1	1601	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	Tinika Mistretta	321600
4	1631	Cisco Systems	www.cisco.com	41.20101093	-76.53824668	Deadra Waggener	321610
5	1641	Pfizer	www.pfizer.com	40.69325986	-75.79453197	Olevia Taubman	321620
6	1651	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 accidentally 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.

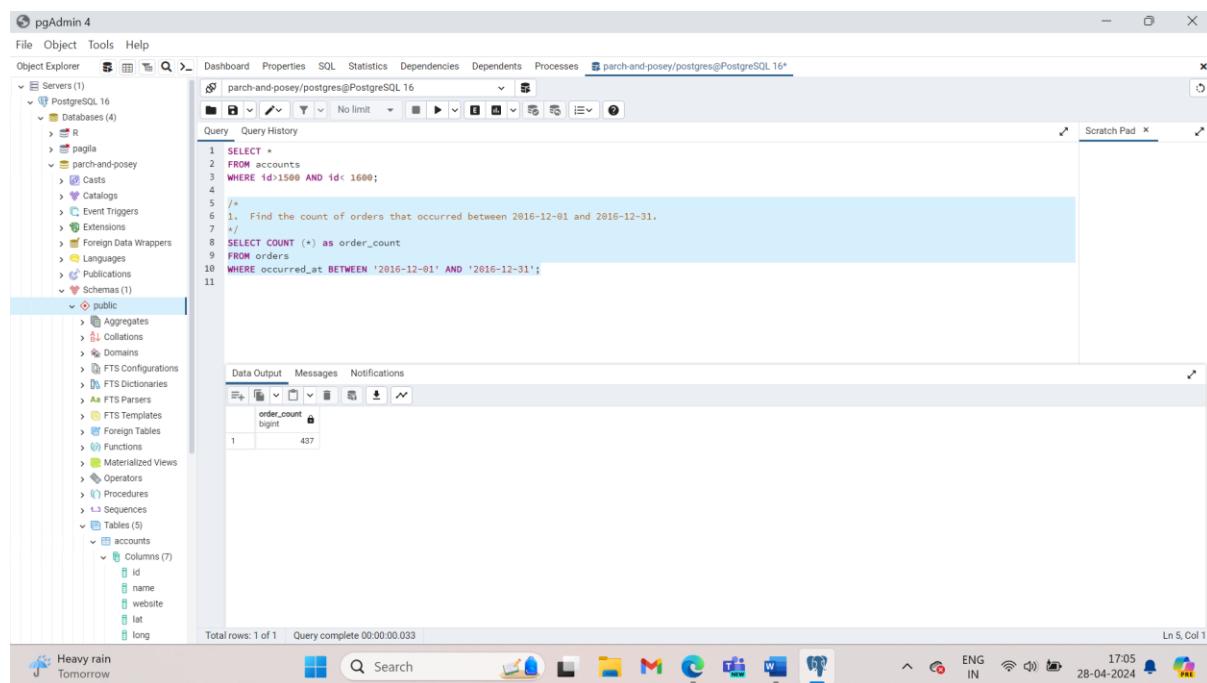
**Imagining yourself as a sales manager at Parch and Posy, you might want to know all the accounts for which the primary POC is null. If you don't have a point of contact, chances are you're not going to be able to keep that customer for much longer. To find all these accounts, we'll have to use some 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';
```



The screenshot shows the pgAdmin 4 interface. The left pane displays the Object Explorer with the 'Servers' node expanded, showing 'PostgreSQL 16' and its databases, including 'parch-and-posey'. The right pane contains a query editor window titled 'parch-and-posey/postgres@PostgreSQL 16'. The query is:

```
1 SELECT *
2 FROM accounts
3 WHERE id > 1500 AND id < 1600;
4
5 /*
6 1. Find the count of orders that occurred between 2016-12-01 and 2016-12-31.
7 */
8 SELECT COUNT (*) as order_count
9 FROM orders
10 WHERE occurred_at BETWEEN '2016-12-01' AND '2016-12-31';
11
```

The results pane shows a single row with the column 'order\_count' containing the value '437'. At the bottom of the interface, there is a taskbar with various icons and system status information.

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
```

The screenshot shows a SQL editor interface with a dark theme. At the top, there are tabs for 'Query 1', 'SQL' (which is selected), 'Display Table', and a '+' button. Below the tabs are buttons for 'Run', 'Limit 100', 'Format SQL', and 'View History...'. The main area contains the following SQL code:

```

1 SELECT *
2   FROM demo.accounts
3 WHERE primary_poc IS NULL;

```

A message at the bottom right indicates 'Succeeded in 931ms'. Below the code, there are buttons for 'Export', 'Copy', 'Chart', and 'Pivot'. The results section shows a table with the following data:

	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

9 rows returned.

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

- 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;

The screenshot shows the pgAdmin 4 interface. The left sidebar shows the Object Explorer with various database objects like servers, databases, and tables. The main window has a 'Query' tab with the following SQL code:

```

11 /**
12  * 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
13  * 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.
14
15
16 SELECT SUM(standard_qty) AS standard, SUM(gloss_qty) AS gloss, SUM (poster_qty) AS poster
17   FROM orders;
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34

```

The 'Data Output' tab shows the results of the query:

	standard	gloss	poster
1	1938346	1013773	723646

Total rows: 1 of 1 Query complete 00:00:00.101

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.

- Find the total amount of poster\_qty paper ordered in the orders table.

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

The screenshot shows the pgAdmin 4 interface with the following details:

- Object Explorer:** Shows the database structure under "Servers (1)" and "PostgreSQL 16".
- Query Editor:** Contains the following SQL code:
 

```
16 SELECT SUM(standard_qty) AS standard, SUM(gloss_qty) AS gloss, SUM (poster_qty) AS poster
17 FROM orders;
18 /*
19 3.Find the total amount of poster_qty paper ordered in the orders table.
20 */
21 /*
22 SELECT SUM(poster_qty) as poster_qty
23 FROM orders;
24
25
26
27
28
29
30
31
32
33
34 Data Output
35 poster_qty
36
37 1 723646
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
```
- Data Output:** A table showing the result of the query:
 

	poster_qty
1	723646
- System Status Bar:** Shows weather (65°F, Mostly cloudy), system icons, and the date/time (19:28, 28-04-2024).

4. Find the total amount of standard\_qty paper ordered in the orders table.

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

The screenshot shows the pgAdmin 4 interface with the following details:

- Object Explorer:** Shows the database structure under "Servers (1)" and "PostgreSQL 16".
- Query Editor:** Contains the following SQL code:
 

```
21 /*
22 SELECT SUM(poster_qty) as poster_qty
23 FROM orders;
24
25 /*
26 4. Find the total amount of standard_qty paper ordered in the orders table.
27 */
28 SELECT SUM(standard_qty) as standard_qty
29 FROM orders;
30
31
32
33
34
35
36
37
38 Data Output
39 standard_qty
40
41 1 1938346
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
```
- Data Output:** A table showing the result of the query:
 

	standard_qty
1	1938346
- System Status Bar:** Shows weather (65°F, Mostly cloudy), system icons, and the date/time (19:30, 28-04-2024).

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;
```

```

SELECT SUM(total_amt_usd) AS total_sales
FROM orders;

```

The screenshot shows the pgAdmin 4 interface. In the left sidebar, the 'Tables' section under the 'public' schema is expanded, showing five tables. The 'orders' table is selected. In the main pane, a query is being run:

```

SELECT SUM(total_amt_usd) AS total_sales
FROM orders;

```

The results are displayed in the 'Data Output' tab:

	total_sales
1	23141511.83

Total rows: 1 of 1 Query complete 00:00:00.102

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;`

```

SELECT standard_amt_usd + gloss_amt_usd AS total_standard_gloss
FROM orders;

```

The screenshot shows the pgAdmin 4 interface. In the left sidebar, the 'Tables' section under the 'public' schema is expanded, showing five tables. The 'orders' table is selected. In the main pane, a query is being run:

```

SELECT standard_amt_usd + gloss_amt_usd AS total_standard_gloss
FROM orders;

```

The results are displayed in the 'Data Output' tab:

	standard_amt_usd	gloss_amt_usd	total_standard_gloss
1	9672346.54	7993159.77	17665506.31

Total rows: 1 of 1 Query complete 00:00:00.103

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(std_qty) AS unit_price_standard  
FROM orders;`

The screenshot shows the pgAdmin 4 interface with a query window open. The query is:

```

38   6. Find the total amount spent on standard_amt_usd and gloss_amt_usd paper for each order in the orders table.
39   This should give a dollar amount for each order in the table.
40   /*
41   SELECT SUM(standard_amt_usd) AS standard_amt, SUM(gloss_amt_usd) AS gloss_amt
42   FROM orders;
43   */
44   /*
45   7. Find the standard_amt_usd per unit of standard_qty paper. Your solution should use both an aggregation
46   and a mathematical operator.
47   */
48   SELECT SUM(standard_amt_usd/(standard_qty+0.01)) AS unit_price_standard
49   FROM orders;

```

The results table shows one row with the value 30368.5511515993515022.

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(std_qty) AS standard_min,
       MIN(gloss_qty) AS gloss_min,
       MIN(poster_qty) AS poster_min,
       MAX(std_qty) AS standard_max,
       MAX(gloss_qty) AS gloss_max,
       MAX(poster_qty) AS poster_max

```

FROM orders;

The screenshot shows the pgAdmin 4 interface with a query window open. The query is:

```

45   7. Find the standard_amt_usd per unit of standard_qty paper. Your solution should use both an aggregation
46   and a mathematical operator.
47   /*
48   SELECT SUM(standard_amt_usd/(standard_qty+0.01)) AS unit_price_standard
49   FROM orders;

```

The results table shows the following data:

	standard_min	gloss_min	poster_min	standard_max	gloss_max	poster_max
1	0	0	0	22591	14281	28262

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;
```

	standard_avg	gloss_avg	poster_avg
1	280.4320023148146148	146.6685474537037037	104.6941550923925926

#### 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;
```

```

/*
8. 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;

/*
9. When was the earliest order ever placed? You only need to return the date.
*/
SELECT MIN(occurred_at)
FROM orders;

```

Total rows: 1 of 1    Query complete 00:00:00.091

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

```

SELECT occurred_at
FROM orders
ORDER BY occurred_at
LIMIT 1;

```

```

/*
8. 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;

/*
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;

```

Total rows: 1 of 1    Query complete 00:00:00.088

11. When did the most recent (latest) web\_event occur?

```

SELECT MAX(occurred_at)
FROM web_events;

```

The screenshot shows the pgAdmin 4 interface. On the left is the Object Explorer pane, which lists various database objects like servers, databases, tables, and columns. The main area is a query editor window titled 'parch-and-posey/postgres@PostgreSQL 16'. It contains the following SQL code:

```

72 /*
73 10. Try performing the same query as above without using an aggregation function.
74 */
75
76 SELECT occurred_at
77 FROM orders
78 ORDER BY occurred_at
79 LIMIT 1;
80
81 /*
82 11. When did the most recent (latest) web_event occur?
83 */
84 SELECT MAX(occurred_at)
85 FROM web_events;
86
87
88
89 Data Output Messages Notifications
90
91 max timestamp without time zone
92
93
94 1 2017-01-01 23:51:09
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115

```

The results pane shows a single row of data: 'max timestamp without time zone' with value '2017-01-01 23:51:09'. At the bottom of the pgAdmin window, there's a status bar with system icons and the date/time '28-04-2024 21:37'.

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;

```

This screenshot is similar to the first one, showing the pgAdmin 4 interface. The query editor window now contains the modified SQL code from step 12:

```

79 LIMIT 1;
80
81 /*
82 11. When did the most recent (latest) web_event occur?
83 */
84 SELECT occurred_at
85 FROM web_events;
86
87 /*
88 12. Try to perform the result of the previous query without using an aggregation function.
89 */
90 SELECT occurred_at
91 FROM web_events
92 ORDER BY occurred_at DESC
93 LIMIT 1;
94
95
96 Data Output Messages Notifications
97
98 occurred_at timestamp without time zone
99
100
101 1 2017-01-01 23:51:09
102
103
104
105
106
107
108
109
110
111
112
113
114
115

```

The results pane shows a single row of data: 'occurred\_at timestamp without time zone' with value '2017-01-01 23:51:09'. The status bar at the bottom shows '28-04-2024 21:40'.

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(std_qty) AS std_qty,
       AVG(poster_qty) AS poster_qty,
       AVG(gloss_qty) AS gloss_qty,
       AVG(std_amt_usd) AS std_amt_usd,
       AVG(gloss_amt_usd) AS gloss_amt_usd;

```

```

    AVG(poster_amt_usd) AS poster_amt_avg
FROM orders;

```

The screenshot shows the pgAdmin 4 interface. On the left is the Object Explorer tree, which includes a 'Servers' node, a 'PostgreSQL 16' database node containing 'Databases', 'Extensions', 'Schemas', and 'Tables'. The 'Tables' node has a 'accounts' table expanded, showing columns: id, name, website, lat, and long. The main window contains a SQL query editor with the following code:

```

1 FROM web_events
2 ORDER BY occurred_at DESC
3 LIMIT 1;
4 /*
5 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.
6 Your final answer should have 6 values - one for each paper type for the average number of sales, as well as the average amount.
7 */
8
9 SELECT AVG(standard_qty) AS standard_avg,
10   AVG(poster_qty) AS poster_avg,
11   AVG(gloss_qty) AS gloss_avg,
12   AVG(standard_amt_usd) AS standard_amt_avg,
13   AVG(gloss_amt_usd) AS gloss_amt_avg,
14   AVG(poster_amt_usd) AS poster_amt_avg
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127

```

The Data Output tab shows the results of the query:

	standard_avg	poster_avg	gloss_avg	standard_amt_avg	gloss_amt_avg	poster_amt_avg
1	280.4320023148148	104.6941550925925926	146.6685474537037037	1399.3556915509259259	1098.5474204282407407	850.116539351851851851

Total rows: 1 of 1    Query complete 00:00:00.090

#### 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 screenshot shows the pgAdmin 4 interface. In the Object Explorer, the 'public' schema is selected. In the main window, a SQL query is run:

```

106 /*
107 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
108 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
109 the entire dataset.
110 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
111 GROUP BY clause.
112 */
113 SELECT account_id,
114     SUM (standard_qty) AS standard_sum,
115     SUM (gloss_qty) AS gloss_sum,
116     SUM (poster_qty) AS poster_sum
117 FROM orders
118 GROUP BY account_id
119 ORDER BY account_id;
--
```

The Data Output tab displays the results of the query:

account_id	standard_sum	gloss_sum	poster_sum
1	7896	7831	3197
2	527	14	0
3	3152	483	175
4	1148	0	215
5	836	770	646
6	3608	356	156
7	925	1556	911
8	803	484	291
9	8193	8744	5965
10	4603	1819	1854
11	1940	1096	917
12	712	1049	3366
13	7504	5949	4110

Total rows: 350 of 350    Query complete 00:00:00.230

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;

```

The screenshot shows the pgAdmin 4 interface with a query editor window titled 'parch-and-posey/postgres@PostgreSQL 16'. The query is:

```

122 /* 15. Which account (by name) placed the earliest order? Your solution should have the account name and the date of the order.
123 */
124 SELECT accounts.name AS account_name, web_events.occurred_at AS date
125 FROM accounts
126 JOIN web_events ON accounts.id = web_events.account_id
127 ORDER BY web_events.occurred_at
128 LIMIT 1;

```

The results table has columns 'account\_name' and 'date', with one row: DISH Network, 2013-12-04 04:18:29.

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;

```

The screenshot shows the pgAdmin 4 interface with a query editor window titled 'parch-and-posey/postgres@PostgreSQL 16'. The query is:

```

129 ORDER BY web_events.occurred_at
130 LIMIT 1;
131 /*
132 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
133 the company name.
134 */
135 SELECT accounts.name, SUM(total_amt_usd) AS total_sales
136 FROM orders
137 JOIN accounts ON accounts.id = orders.account_id
138 GROUP BY accounts.name;

```

The results table has columns 'name' and 'total\_sales', with 16 rows. The data includes Comcast, Microsoft, Monsanto, Dean Foods, KKR, Performance Food Group, Paccar, USA, CST Brands, Ally Financial, Argen, Broadcast, AIG, W.W. Grainger, Reynolds American, and Sears Holdings.

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;

The screenshot shows the pgAdmin 4 interface with a query editor and a results grid. The query is:

```
141 /*
142 17. Via what channel did the most recent (latest) web_event occur, which account was associated with this web_event? Your query should
143 return only three values - the date, channel, and account name.
144 */
145 SELECT web_events.occurred_at, web_events.channel, accounts.name
146 FROM web_events
147 JOIN accounts ON web_events.account_id = accounts.id
148 ORDER BY web_events.occurred_at DESC
149 LIMIT 1;
150
151
```

The results grid shows one row:

occurred_at	channel	name
2017-01-01 23:51:09	organic	Molina Healthcare

At the bottom of the pgAdmin window, there is a status bar with system icons and the text "Total rows: 1 of 1 Query complete 00:00:00.096".

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;
```

The screenshot shows the pgAdmin 4 interface with a query editor and a results grid. The query is:

```
150 /*
151 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
152 channel and the number of times the channel was used.
153 */
154 */
155 SELECT web_events.channel, COUNT(*)
156 FROM web_events
157 GROUP BY web_events.channel;
```

The results grid shows six rows:

channel	count
twitter	474
adwords	906
organic	952
banner	476
facebook	967
direct	5298

At the bottom of the pgAdmin window, there is a status bar with system icons and the text "Total rows: 6 of 6 Query complete 00:00:00.123".

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;
```

The screenshot shows the pgAdmin 4 interface with a query editor containing the following SQL code:

```

159 /* 19. Who was the primary contact associated with the earliest web_event?
160 */
161 */
162 SELECT accounts.primary_poc
163 FROM accounts
164 JOIN web_events ON accounts.id = web_events.account_id
165 ORDER BY web_events.occurred_at DESC
166 LIMIT 1;
167
168

```

The results pane shows a single row:

primary_poc	character
Hilde Klopfer	

Below the results, it says "Total rows: 1 of 1 Query complete 00:00:00.109".

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;

```

The screenshot shows the pgAdmin 4 interface with a query editor containing the following SQL code:

```

168 /*
169 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.
170 Order from smallest dollar amounts to largest.
171 */
172 SELECT accounts.name, MIN(orders.total_amt_usd) AS smallest_order
173 FROM accounts
174 JOIN orders ON accounts.id = orders.account_id
175 GROUP BY accounts.name
176 ORDER BY smallest_order;
177
178

```

The results pane shows a table with 350 rows:

name	smallest_order
Disney	0.00
Chevron	0.00
Twenty-First Century Fox	0.00
Reynolds American	0.00
Navistar International	0.00
Boeing	0.00
Gilead Sciences	0.00
General Mills	0.00
United Continental Holdings	0.00
Citigroup	0.00
R.R. Donnelly & Sons	0.00
BlackRock	0.00
Lithia Motors	0.00
J.P. Morgan Chase	0.00
FedEx	0.00
Monsanto	0.00

Below the results, it says "Total rows: 350 of 350 Query complete 00:00:00.127".

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;

```

The screenshot shows the pgAdmin 4 interface. In the Object Explorer, under the 'Tables' section, there is a table named 'accounts'. In the Data Output tab, the results of the following SQL query are displayed:

```

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;

```

The results are:

name	num_reps
Midwest	9
Southeast	10
West	10
Northeast	21

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;

```

```

pgAdmin 4
File Object Tools Help
Object Explorer Dashboard Properties SQL Statistics Dependencies Dependents Processes parch-and-posey/postgres@PostgreSQL 16*
Servers (1) PostgreSQL 16
  Databases (4)
    R
    pagila
    parch-and-posey
      Casts
      Catalogs
      Event Triggers
      Extensions
      Foreign Data Wrappers
      Languages
      Publications
      Schemas (1)
        public
          Aggregates
          Collations
          Domains
          FTS Configurations
          FTS Dictionaries
          FTS Parsers
          FTS Templates
          Foreign Tables
          Functions
          Materialized Views
          Operators
          Procedures
          Sequences
          Tables (5)
            accounts
              Columns (7)
                id
                name
                website
                lat
                long
      scratchpad
      Query Pad
      Query History
      Messages
      Notifications
      189
      190   SELECT account_id, channel, COUNT(id) AS events
      191   FROM web_events
      192   GROUP BY account_id, channel
      193   ORDER BY account_id, channel;
      194
      195
      196
      197
      198
      199
      200
Data Output
account_id | channel | events | bigint
1           | adwords | 5       |
2           | banner  | 3       |
3           | direct   | 22      |
4           | facebook | 2       |
5           | organic  | 6       |
6           | twitter  | 1       |
7           | adwords  | 1       |
8           | direct   | 1       |
9           | facebook | 1       |
10          | adwords | 7       |
11          | banner  | 1       |
12          | direct   | 9       |
13          | facebook | 9       |
14          | organic  | 6       |
15          | twitter  | 2       |
Total rows: 1000 of 1509  Query complete 00:00:00.251

```

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;

```

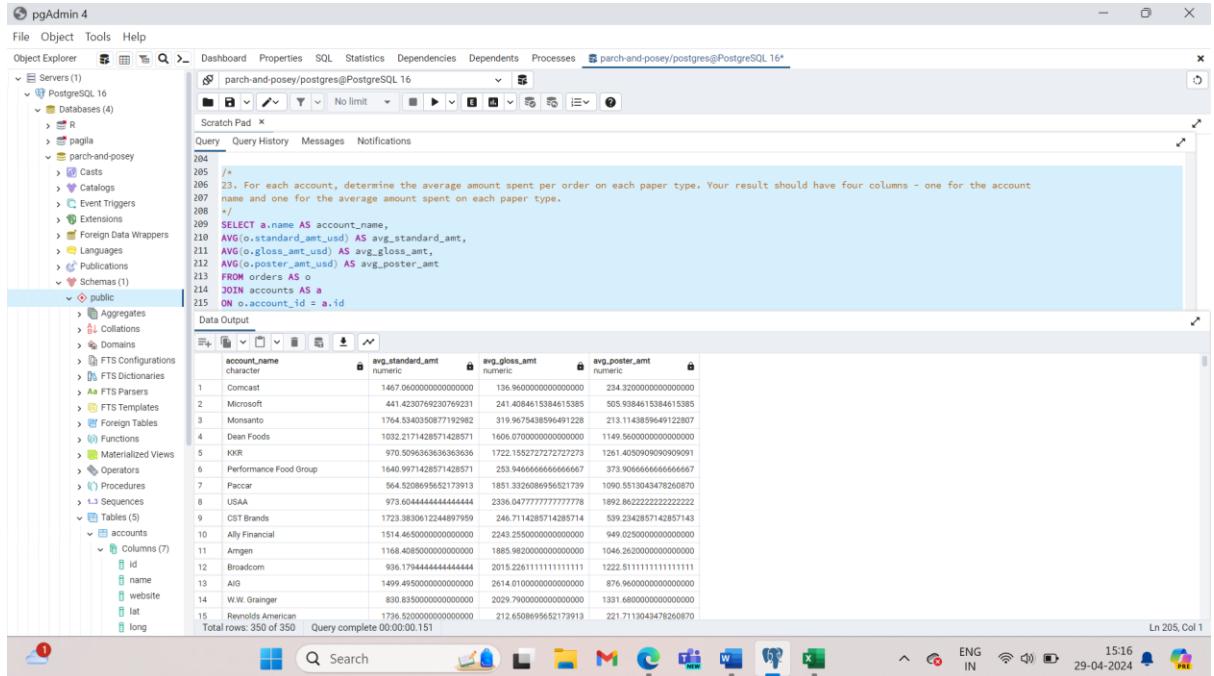
```

pgAdmin 4
File Object Tools Help
Object Explorer Dashboard Properties SQL Statistics Dependencies Dependents Processes parch-and-posey/postgres@PostgreSQL 16*
Servers (1) PostgreSQL 16
  Databases (4)
    R
    pagila
    parch-and-posey
      Casts
      Catalogs
      Event Triggers
      Extensions
      Foreign Data Wrappers
      Languages
      Publications
      Schemas (1)
        public
          Aggregates
          Collations
          Domains
          FTS Configurations
          FTS Dictionaries
          FTS Parsers
          FTS Templates
          Foreign Tables
          Functions
          Materialized Views
          Operators
          Procedures
          Sequences
          Tables (5)
            accounts
              Columns (7)
                id
                name
                website
                lat
                long
      scratchpad
      Query Pad
      Query History
      Messages
      Notifications
      195 /*
      196 22. For each account, determine the average amount of each type of paper they purchased across their orders. Your result should have four
      197 columns - one for the account name and one for the average quantity purchased for each of the paper types for each account.
      198 */
      199 SELECT a.name, AVG(o.standard_qty) AS avg_standard, AVG(o.poster_qty) AS avg_poster, AVG(o.gloss_qty) AS avg_gloss
      200 FROM accounts AS a
      201 JOIN orders AS o
      202 ON a.id= o.account_id
      203 GROUP BY a.name;
Data Output
name | avg_standard | avg_poster | avg_gloss
1   | 294.00000000000000 | 28.8571428571428571 | 18.2857142857142857
2   | 88.4615384615384615 | 62.307492307492307 | 32.2807492307492308
3   | 353.614353597192982 | 26.245614035077193 | 42.7192962456140351
4   | 206.8571428571428571 | 141.5714285714285714 | 214.285714285714286
5   | 194.4909090909090909 | 155.3454545454545455 | 229.9272727272727273
6   | 328.8571428571428571 | 46.0476190476190476 | 33.9047619047619048
7   | 113.1304347826086957 | 134.3043478260869565 | 247.1799130434782609
8   | 195.1111111111111111 | 233.1111111111111111 | 311.8888888888888889
9   | 345.967346987755102 | 66.4081625535061224 | 32.988755102040816
10  | 303.5000000000000000 | 116.8750000000000000 | 299.5000000000000000
11  | 234.1500000000000000 | 128.8500000000000000 | 251.8000000000000000
12  | 187.6111111111111111 | 150.5555555555555556 | 269.0555555555555556
13  | 300.5000000000000000 | 108.0000000000000000 | 349.0000000000000000
14  | 166.5000000000000000 | 164.0000000000000000 | 271.0000000000000000
15  | 348.0000000000000000 | 27.3043478260869565 | 28.391043478260869567
16  | 378.3333333333333333 | 152.366666666666666667 | 201.433333333333333333
Total rows: 350 of 350  Query complete 00:00:00.115

```

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
 WHERE 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;
```

The screenshot shows the pgAdmin 4 interface. In the Object Explorer, under the 'Tables' section, there is a table named 'accounts'. In the 'Data Output' tab, a SQL query is run:

```

218 /*
219 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.
220 */
222 SELECT s.name AS sales_reps_name, w.channel AS web_events_channel, COUNT(*) AS num_events
223 FROM web_events AS w
224 JOIN accounts AS a
225 ON w.account_id = a.id
226 JOIN sales_reps AS s
227 ON a.sales_rep_id = s.id
228 GROUP BY sales_reps_name, web_events_channel
229 ORDER BY num_events DESC;

```

The resulting data is as follows:

	sales_reps_name	web_events_channel	num_events
1	Earlie Schleusner	direct	234
2	Vernita Plump	direct	232
3	Moon Torian	direct	194
4	Georgiana Chisholm	direct	188
5	Tia Amato	direct	185
6	Maren Musto	direct	184
7	Nelle Meaux	direct	179
8	Maryanna Fiorentino	direct	168
9	Dorotha Seawell	direct	161
10	Charles Bidwell	direct	159
11	Sibyl Lauria	direct	147
12	Elna Condello	direct	146
13	Micha Woodford	direct	145
14	Michel Averette	direct	145
	Total rows: 295 of 295		***

Query complete 00:00:00.123

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;

```

```

231 /*
232 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
233 columns - the region name, the channel, and the number of occurrences. Order your table with the highest number of occurrences first.
234 */
235 SELECT r.name AS region_name, w.channel AS web_events_channel, COUNT(*) AS num_events
236 FROM accounts AS a
237 JOIN web_events AS w
238 ON a.id = w.account_id
239 JOIN sales_reps AS s
240 ON s.id = w.sales_rep_id
241 JOIN region AS r
242 ON r.id = s.region_id
243 GROUP BY region_name, web_events_channel
244 ORDER BY num_events DESC;

```

	region_name	web_events_channel	num_events
1	Northeast	direct	1800
2	Southeast	direct	1548
3	West	direct	1254
4	Midwest	direct	696
5	Northeast	facebook	335
6	Northeast	organic	317
7	Northeast	adwords	300
8	Southeast	facebook	278
9	Southeast	organic	275
10	Southeast	adwords	264
11	West	organic	243
12	West	adwords	241

Total rows: 24 of 24 Query complete 00:00:00.091 Ln 231, Col 1

## DISTINCT

You can think of distinct this way: if you want to group by some columns but you don't necessarily want to include any 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;

```

The screenshot shows the pgAdmin 4 interface. The left pane is the Object Explorer, displaying the database structure. The right pane is the Query tool, showing a query and its results.

```

245
246 /*
247 26. Use DISTINCT to test if there are any accounts associated with more than one region.
248 */
249
250 SELECT DISTINCT id, name
251 FROM accounts;
252

```

	<b>id</b>	<b>name</b>
1	1591	Lockheed Martin
2	2701	PNC Financial Services Group
3	2451	Starbucks
4	1681	Nationwide
5	4111	Republic Services
6	3881	Western Refining
7	4041	Celgene
8	3321	Computer Sciences
9	3811	Precision Castparts
10	3971	Ally Financial
11	1491	Prudential Financial
12	4031	Hormel Foods
13	2531	Lear
14	1831	CHS
15	3851	Baxter International
16	3501	WestRock
17	2121	EMC
18	2801	Whole Foods Market

Total rows: 351 of 351    Query complete 00:00:00.065

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

`SELECT DISTINCT id, name`

`FROM sales_reps;`

The screenshot shows the pgAdmin 4 interface. The left pane is the Object Explorer, displaying the database structure. The right pane is the Query tool, showing a query and its results.

```

253 /*
254 27. Have any sales reps worked on more than one account?
255 */
256 SELECT DISTINCT id, name
257 FROM sales_reps;
258

```

	<b>id</b>	<b>name</b>
1	321870	Derrick Boggs
2	321500	Samuel Racine
3	321580	Sibyl Lauria
4	321590	Lavera Oles
5	321860	Saran Ram
6	321820	Dorotha Seawell
7	321700	Debrah Wardle
8	321740	Charles Bidwell
9	321780	Julie Starr
10	321600	Ernestine Pickron
11	321510	Eugena Esser
12	321750	Cliff Mints
13	321760	Delilah Krum
14	321690	Gianna Dossey
15	321810	Moon Torian
16	321950	Elwood Shutt
17	321630	Julia Behrman
18	321940	Maryanna Florentino
19	321540	Cara Clarke
20	321900	Soraya Gutten

Total rows: 50 of 50    Query complete 00:00:00.064

## HAVING

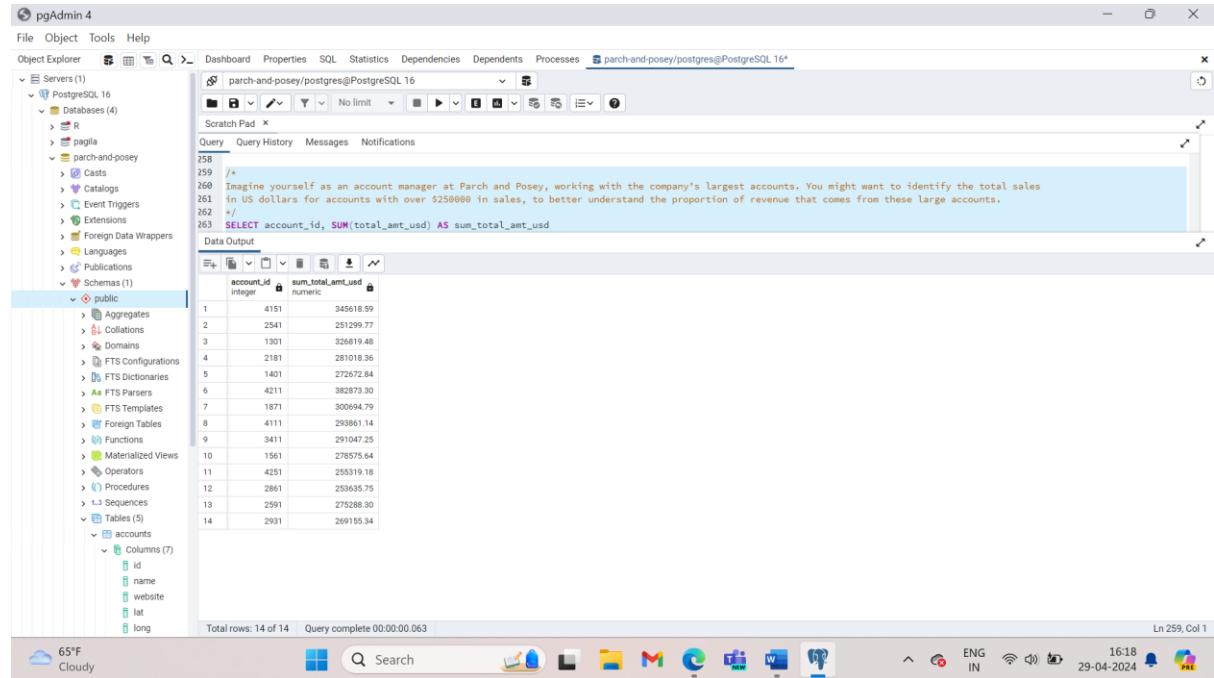
Imagine yourself as an account manager at Parch and Posey, working with the company's largest accounts. You might want to identify the total sales in US dollars for accounts with over \$250000 in sales, to better understand the proportion of revenue that comes from these large accounts.

`SELECT account_id, SUM(total_amt_usd) AS sum_total_amt_usd`

FROM orders

GROUP BY 1

HAVING SUM(total\_amt\_usd) >= 250000;



The screenshot shows the pgAdmin 4 interface. In the Object Explorer, there's a tree view of databases, schemas, and tables. The 'Tables' section under the 'accounts' schema is expanded, showing columns: id, name, website, lat, and long. The main window contains a SQL query in the 'Query Pad' tab:

```
258 /*
259  * 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
260  * in US dollars for accounts with over $250000 in sales, to better understand the proportion of revenue that comes from these large accounts.
261 */
262
263 SELECT account_id, SUM(total_amt_usd) AS sum_total_amt_usd
```

The 'Data Output' tab shows the results of the query:

account_id	sum_total_amt_usd
1	345618.59
2	251299.77
3	326819.48
4	281018.36
5	272572.84
6	382873.30
7	300694.79
8	293861.14
9	291047.25
10	1561
11	255319.18
12	2861
13	275288.30
14	269155.34

Total rows: 14 of 14 Query complete 00:00:00.063

**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;
```

pgAdmin 4

File Object Tools Help

Object Explorer    Dashboard Properties SQL Statistics Dependencies Dependents Processes    parch-and-posey/postgres@PostgreSQL 16\*

Servers (1) PostgreSQL 16 Databases (4) R pagila parch-and-posey Casts Catalogs Event Triggers Extensions Foreign Data Wrappers Languages Publications Schemas (1) public Aggregates Collations Domains FTS Configurations FTS Dictionaries AA FTS Parsers FTS Templates Foreign Tables Functions Materialized Views Operators Procedures Sequences Tables (5) accounts Columns (7) id name website lat long

Scratch Pad    Query History Notifications

```

267
268 /*
269 28. How many of the sales reps have more than 5 accounts that they manage?
270 */
271 SELECT s.id, s.name, COUNT(*) num_accounts
272 FROM accounts AS a
273 JOIN sales_reps AS s
274 ON a.sales_rep_id = s.id
275 GROUP BY s.id, s.name
276 HAVING COUNT(*) > 5
277 ORDER BY num_accounts;

```

Data Output

	id	name	num_accounts
1	321500	Samuel Racine	6
2	321510	Eugena Esser	6
3	321580	Sibyl Lauria	6
4	321590	Nicole Victory	6
5	321600	Ella Felder	6
6	321700	Debrah Wardle	6
7	321870	Derrick Bogess	7
8	321740	Charles Bidwell	7
9	321750	Cliff Mente	7
10	321690	Gianna Dossey	7
11	321900	Soraya Fulton	7
12	321880	Babette Soukup	7
13	321890	Nelle Meaux	7
14	321680	Elna Condello	7
15	321520	Michel Averette	7

Total rows: 34 of 34    Query complete 00:00:00.067

Cloudy 65°F Search ENG IN 16:29 29-04-2024

## 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;

```

pgAdmin 4

File Object Tools Help

Object Explorer    Dashboard Properties SQL Statistics Dependencies Dependents Processes    parch-and-posey/postgres@PostgreSQL 16\*

Servers (1) PostgreSQL 16 Databases (4) R pagila parch-and-posey Casts Catalogs Event Triggers Extensions Foreign Data Wrappers Languages Publications Schemas (1) public Aggregates Collations Domains FTS Configurations FTS Dictionaries AA FTS Parsers FTS Templates Foreign Tables Functions Materialized Views Operators Procedures Sequences Tables (5) accounts Columns (7) id name website lat long

Scratch Pad    Query History Notifications

```

278
279 /*
280 29. How many accounts have more than 20 orders?
281 */
282 SELECT a.id, a.name, COUNT(*) num_orders
283 FROM accounts AS a
284 Join orders AS o
285 ON a.id = o.account_id
286 GROUP BY a.id , a.name
287 HAVING COUNT(*) >20
288 ORDER BY num_orders;

```

Data Output

	id	name	num_orders
1	2841	Performance Food Group	21
2	1321	Anthem	21
3	4171	Thrive Financial for Lutherans	21
4	2571	Jabil Circuit	22
5	2191	Raytheon	22
6	3651	Reynolds American	23
7	1841	American Express	23
8	1831	CHS	23
9	1941	Exelon	23
10	4281	SunTrust Banks	23
11	2701	PNC Financial Services Group	23
12	3181	Texas Instruments	23
13	1611	Coca-Cola	24
14	3321	Computer Sciences	24
15	1251	Bank of America Corp.	24

Total rows: 120 of 120    Query complete 00:00:00.066

USD/JPY -1.54% Search ENG IN 16:56 29-04-2024

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;
```

The screenshot shows the pgAdmin 4 interface. The left sidebar displays the Object Explorer with a tree view of servers, databases, tables, and columns. The main area shows a query editor window with the following content:

```
289 30. Which account has the most orders?
290 /*
291
292
293 SELECT a.id, a.name, COUNT(*) num_orders
294 FROM accounts AS a
295 JOIN orders AS o
296 ON a.id = o.account_id
297 GROUP BY a.id, a.name
298 ORDER BY num_orders DESC
299 LIMIT 1;
```

Below the query editor is a Data Output pane showing the results of the query:

	id	name	num_orders
1	3411	Leucadia National	71

Total rows: 1 of 1 Query complete 00:00:00.066

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;
```

The screenshot shows the pgAdmin 4 interface with a query editor and a data output window. The query is:

```

301 /*
302 31. How many accounts spent more than 30,000 usd total across all orders?
303 */
304 SELECT a.id, a.name, SUM(o.total_amt_usd) AS total_spent
305 FROM accounts a
306 JOIN orders o
307 ON a.id = o.account_id
308 GROUP BY a.id, a.name
309 HAVING SUM(o.total_amt_usd) > 30000
310 ORDER BY total_spent;
  
```

The data output table has columns: id, name, and total\_spent. The data is:

	id	name	total_spent
1	1661	American Airlines Group	30093.18
2	1431	PepsiCo	30095.72
3	3661	Group 1 Automotive	30708.92
4	1141	Costco	30741.01
5	1761	Oracle	31231.56
6	2961	Nordstrom	31273.02
7	2031	Enterprise Products Partners	31971.72
8	2641	American Electric Power	32044.55
9	2601	General Mills	32226.37
10	2161	Halliburton	32645.38
11	1511	Humana	32937.93
12	2331	Whirlpool	32965.13
13	2391	Tenet Healthcare	33028.08
14	3121	Viacom	33673.86
15	2991	Davita HealthCare Partner	33992.78

Total rows: 204 of 204 Query complete 00:00:00.066

### 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;
  
```

The screenshot shows the pgAdmin 4 interface with a query editor and a data output window. The query is:

```

307 ON a.id = o.account_id
308 GROUP BY a.id, a.name
309 HAVING SUM(o.total_amt_usd) > 30000
310 ORDER BY total_spent;
  
```

Below it, the next query is:

```

311 /*
312 32. How many accounts spent less than 1,000 usd total across all orders?
313 */
314 /*
315 SELECT a.id, a.name, SUM(o.total_amt_usd) AS total_spent
316 FROM accounts a
317 JOIN orders o
318 ON a.id = o.account_id
319 GROUP BY a.id, a.name
320 HAVING SUM(o.total_amt_usd) < 1000
321 ORDER BY total_spent;
  
```

The data output table has columns: id, name, and total\_spent. The data is:

	id	name	total_spent
1	1901	Nike	390.25
2	1671	Delta Air Lines	859.64
3	4321	Level 3 Communications	881.73

Total rows: 3 of 3 Query complete 00:00:00.052

### 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;

```

```

1. 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;

```

	<b>id</b>	<b>name</b>	<b>total_spent</b>
1	4211	EOG Resources	382873.30

Total rows: 1 of 1 Query complete 00:00:00.060 Ln 324, Col 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;

```

pgAdmin 4

File Object Tools Help

Object Explorer    Dashboard Properties SQL Statistics Dependencies Dependents Processes    parch-and-posey/postgres@PostgreSQL 16\*

Servers (1) PostgreSQL 16  
Databases (4)  
Schemas (1)  
Tables (5)

Scratch Pad

```

334
335 /*
336 34. Which account has spent the least with us?
337 */
338 SELECT a.id, a.name, SUM(o.total_amt_usd) AS total_spent
339 FROM accounts AS a
340 JOIN orders AS o
341 ON a.id = o.account_id
342 GROUP BY a.id, a.name
343 ORDER BY total_spent
344 LIMIT 1;
345

```

Data Output

	<b>id</b>	<b>name</b>	<b>total_spent</b>
1	1901	Nike	390.25

Total rows: 1 of 1    Query complete 00:00:00.055

Light rain At night

Search    Microsoft Edge    File Explorer    Mail    Task View    Start    17:29 29-04-2024 ENG IN

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;

```

pgAdmin 4

File Object Tools Help

Object Explorer    Dashboard Properties SQL Statistics Dependencies Dependents Processes    parch-and-posey/postgres@PostgreSQL 16\*

Schemas (1)

Tables (5)

Scratch Pad

```

345
346 /*
347 35. Which accounts used facebook as a channel to contact customers more than 6 times?
348 */
349 SELECT a.id, a.name, w.channel, COUNT(*) use_of_channel
350 FROM accounts AS a
351 JOIN web_events AS w
352 ON a.id = w.account_id
353 GROUP BY a.id, a.name, w.channel
354 Having COUNT(*) > 6 AND w.channel = 'facebook'
355 ORDER BY use_of_channel;
356

```

Data Output

	<b>id</b>	<b>name</b>	<b>channel</b>	<b>use_of_channel</b>
1	4291	Avis Budget Group	facebook	7
2	1701	Best Buy	facebook	7
3	3261	Farmers Insurance Exchange	facebook	7
4	4241	Laboratory Corp. of America	facebook	7
5	1741	Honeywell International	facebook	7
6	3411	Leucadia National	facebook	7
7	1261	Wells Fargo	facebook	7
8	3991	eBay	facebook	7
9	1221	J.P. Morgan Chase	facebook	7
10	3231	Parker-Hannifin	facebook	7
11	1271	Home Depot	facebook	7
12	2461	Paccar	facebook	8
13	2541	Fluor	facebook	8
14	4491	PPL	facebook	8
15	4161	CoreMark Holding	facebook	8

Total rows: 46 of 46    Query complete 00:00:00.064

65°F Cloudy

Search    Microsoft Edge    File Explorer    Mail    Task View    Start    17:42 29-04-2024 ENG IN

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;

```

The screenshot shows the pgAdmin 4 interface. The left sidebar displays the Object Explorer with various database objects like Schemas, Tables, and Columns. The main area contains a query editor with the following SQL code:

```

357 /*
358 36. Which account used facebook more as a channel?
359 */
360 SELECT a.id, a.name, w.channel, COUNT(*) use_of_channel
361 FROM accounts AS a
362 JOIN web_events AS w
363 ON a.id = w.account_id
364 WHERE w.channel = 'facebook'
365 GROUP BY a.id, a.name, w.channel
366 ORDER BY use_of_channel DESC
367 LIMIT 1;
368

```

The Data Output pane shows the result of the query:

	id	name	channel	use_of_channel
1	1851	Gilead Sciences	facebook	16

Total rows: 1 of 1 Query complete 00:00:00.043 Ln 357, Col 3

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;

```

pgAdmin 4

File Object Tools Help

Object Explorer Dashboard Properties SQL Statistics Dependencies Dependents Processes parch-and-posey/postgres@PostgreSQL 16\*

Scratch Pad

```

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;

```

Data Output

	<b>id</b>	<b>name</b>	<b>channel</b>	<b>use_of_channel</b>
1	3411	Leucadia National	direct	52
2	2731	Colgate-Palmolive	direct	51
3	1601	New York Life Insurance	direct	51
4	2051	Philip Morris International	direct	49
5	3911	Charter Communications	direct	48
6	3491	BlackRock	direct	48
7	2871	FifthEnergy	direct	48
8	3471	ADP	direct	48
9	2351	AutoNation	direct	48
10	2481	Altria Group	direct	47

Total rows: 10 of 10 Query complete 00:00:00.062

USD/JPY -1.26% 17:51 29-04-2024 Ln 369, Col 1

## 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.

pgAdmin 4

File Object Tools Help

Object Explorer Dashboard Properties SQL Statistics Dependencies Dependents Processes parch-and-posey/postgres@PostgreSQL 16\*

Scratch Pad

```

SELECT occurred_at, SUM(standard_qty) AS standard_qty_sum
FROM orders
GROUP BY occurred_at
ORDER BY occurred_at;

```

Data Output

	<b>occurred_at</b>	<b>standard_qty_sum</b>
1	2013-12-04 04:22:44	0
2	2013-12-04 04:45:54	490
3	2013-12-04 04:53:25	528
4	2013-12-05 02:29:16	0
5	2013-12-05 03:35:56	492
6	2013-12-06 02:19:20	502
7	2013-12-06 12:55:22	53
8	2013-12-06 12:57:41	308
9	2013-12-06 13:14:47	75
10	2013-12-06 13:17:25	281
11	2013-12-06 23:45:16	12
12	2013-12-06 23:47:45	461
13	2013-12-08 00:54:42	490
14	2013-12-08 00:54:42	43
15	2013-12-08 07:05:07	85
16	2013-12-08 07:11:38	477
17	2013-12-08 11:24:52	1780
18	2013-12-08 11:32:52	501
19	2013-12-08 20:13:49	0
20	2013-12-08 20:37:53	501

Total rows: 1000 of 6908 Query complete 00:00:00.268

68°F Mostly sunny 17:48 30-04-2024 Ln 385, Col 1

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 1<sup>st</sup>, 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 1<sup>st</sup> 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 1<sup>st</sup>. 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);
```

```

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);

```

day	standard_qty.sum
2013-12-04 00:00:00	1018
2013-12-05 00:00:00	492
2013-12-06 00:00:00	1692
2013-12-08 00:00:00	3877
2013-12-09 00:00:00	972
2013-12-10 00:00:00	1140
2013-12-11 00:00:00	1550
2013-12-12 00:00:00	1244
2013-12-13 00:00:00	315
2013-12-14 00:00:00	2120
2013-12-15 00:00:00	489
2013-12-16 00:00:00	997
2013-12-17 00:00:00	1275
2013-12-18 00:00:00	519
2013-12-19 00:00:00	928
2013-12-20 00:00:00	863
2013-12-22 00:00:00	1158
2013-12-23 00:00:00	487
2013-12-24 00:00:00	308
2013-12-25 00:00:00	504

Total rows: 1060 Query complete 00:00:00.097 Ln 384, Col 1

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;
```

The screenshot shows the pgAdmin 4 interface. On the left is the Object Explorer tree, which includes a 'Servers' node, a 'PostgreSQL 16' database node containing 'Databases' (parch-and-posey, R, pagila), 'Extensions', 'Foreign Data Wrappers', 'Languages', 'Publications', and 'Schemas' (public). The 'public' schema is expanded to show 'Aggregates', 'Collations', 'Domains', 'FTS Configurations', 'FTS Dictionaries', 'FTS Parsers', 'FTS Templates', 'Foreign Tables', 'Functions', 'Materialized Views', 'Operators', 'Procedures', 'Sequences', and 'Tables' (5). One of the tables, 'accounts', is selected and expanded to show columns: id, name, website, lat, and long.

In the center, the 'Query Pad' tab is active, displaying a SQL query:

```

389 /*
390  On what day of the week are the most sales made?
391 */
392
393 SELECT DATE_PART('dow', occurred_at) AS day_of_week, SUM(total) AS total_qty
394 FROM orders
395 GROUP BY 1
396 ORDER BY 2 DESC;
397

```

The 'Data Output' tab shows the results of the query:

day_of_week	total_qty
0	559873
5	536776
6	521813
1	518016
2	517857
3	511525
4	510405

Total rows: 7 of 7    Query complete 00:00:00.101

**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;
```

The screenshot shows the pgAdmin 4 interface. In the Object Explorer, under the 'Servers' node, there is a 'PostgreSQL 16' server with a 'Databases' node containing 'parch-and-posey'. The 'Tables' node under 'parch-and-posey' contains a 'orders' table. A query is running in the 'Query Pad' tab:

```

397 /*
398 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
399   in the yearly sales totals?
400 */
401
402 SELECT DATE_PART('year', occurred_at) AS ord_year, SUM(total_amt_usd) AS total_spent
403 FROM orders
404 GROUP BY 1
405 ORDER BY 2 DESC;
406

```

The 'Data Output' tab shows the results of the query:

ord_year	total_spent
2016	12864917.92
2015	5762004.94
2014	4069106.54
2013	377331.00
2017	78151.43

Total rows: 5 of 5    Query complete 00:00:00.057

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;

```

pgAdmin 4

File Object Tools Help

Object Explorer    Dashboard Properties SQL Statistics Dependencies Dependents Processes    parch-and-posey/postgres@PostgreSQL 16\*

Servers (1)    PostgreSQL 16    Databases (4)    public    Schema (1)

Scratch Pad    Query History Notifications

```

106 /*
107 39. Which month did Parch & Posey have the greatest sales in terms of total dollars?
108 Are all months evenly represented by the dataset?
109 */
110
111 SELECT DATE_PART('month', occurred_at) AS ord_month, SUM(total_amt_usd) AS total_spent
112 FROM orders
113 WHERE occurred_at BETWEEN '2014-01-01' AND '2017-01-01'
114 GROUP BY 1
115 ORDER BY 2 DESC;

```

Data Output

	ord_month	total_spent
1	12	2782080.98
2	10	2427505.97
3	11	2390033.75
4	9	2017216.88
5	7	1978731.15
6	8	1918107.22
7	6	1871118.52
8	3	1659887.88
9	4	1562037.74
10	5	1537082.23
11	2	1312616.64
12	1	1259951.44

Total rows: 12 of 12    Query complete 00:00:00.068

Ln 407, Col 1

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;

```

pgAdmin 4

File Object Tools Help

Object Explorer    Dashboard Properties SQL Statistics Dependencies Dependents Processes    parch-and-posey/postgres@PostgreSQL 16\*

Servers (1)    PostgreSQL 16    Databases (4)    public    Schema (1)

Scratch Pad    Query History Notifications

```

116 /*
117 40. Which year did Parch and Posey have the greatest sales in terms of total number of orders? Are all years evenly represented
118 by the dataset?
119 */
120
121 SELECT DATE_PART('year', occurred_at) AS ord_year, COUNT(*) AS total_sales
122 FROM orders
123 GROUP BY 1
124 ORDER BY 2 DESC;
125

```

Data Output

	ord_year	total_sales
1	2016	3757
2	2015	1725
3	2014	1306
4	2013	99
5	2017	25

Total rows: 5 of 5    Query complete 00:00:00.097

Ln 417, Col 1

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;

The screenshot shows the pgAdmin 4 interface. In the Object Explorer, there is one server named 'PostgreSQL 16' which contains a database named 'parch-and-posey'. This database has several objects like 'Casts', 'Catalogs', 'Event Triggers', 'Extensions', 'Foreign Data Wrappers', 'Languages', 'Publications', and 'Schemas'. One schema is named 'public' which contains tables like 'Aggregates', 'Collations', 'Domains', 'FTS Configurations', 'FTS Dictionaries', 'FTS Parsers', 'FTS Templates', 'Foreign Tables', 'Functions', 'Materialized Views', 'Operators', 'Procedures', 'Sequences', and 'Tables'. A specific table named 'accounts' is selected, showing columns: id, name, website, lat, and long.

In the main window, a query is being run:

```

126 /*
127 41. Which month did Parch & Posey have the greatest sales in terms of total number of orders? Are all months evenly represented
128 in the dataset?
129 */
130 SELECT DATE_PART('month', occurred_at) AS ord_month, COUNT(*) AS total_sales
131 FROM orders
132 WHERE occurred_at BETWEEN '2014-01-01' AND '2017-01-01'
133 GROUP BY 1
134 ORDER BY 2 DESC;
135

```

The Data Output tab displays the results of the query:

ord_month	total_sales
1	12
2	11
3	10
4	8
5	9
6	7
7	6
8	5
9	3
10	4
11	1
12	2

Total rows: 12 of 12    Query complete 00:00:00.070

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;

```

The screenshot shows the pgAdmin 4 interface. In the Object Explorer, under the 'Servers' section, there is a connection to 'PostgreSQL 16' which contains four databases: 'R', 'pagila', 'parch-and-posey', and 'public'. The 'parch-and-posey' database is selected. In the main area, a 'Scratch Pad' tab is open with a SQL query. The query is as follows:

```

335 /*
336 42. In which month of which year did Walmart spend the most on gloss paper in terms of dollars?
337 */
338
339 SELECT DATE_TRUNC('month', o.occurred_at) AS ord_date, SUM(o.gloss_amt_usd) AS tot_spent
340 FROM orders AS o
341 JOIN accounts AS a
342 ON a.id = o.account_id
343 WHERE a.name = 'Walmart'
344 GROUP BY 1
345 ORDER BY 2 DESC
346
347

```

The 'Data Output' tab shows the result of the query:

	ord_date	tot_spent
1	2016-05-01 00:00:00	9257.64

Total rows: 1 of 1 Query complete 00:00:00.069

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 arithmetic. 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;

```

pgAdmin 4

File Object Tools Help

Object Explorer    Dashboard Properties SQL Statistics Dependencies Dependents Processes parch-and-posey/postgres@PostgreSQL 16\*

Servers (1) PostgreSQL 16 Databases (4) R pagila parch-and-posey Casts Catalogs Event Triggers Extensions Foreign Data Wrappers Languages Publications Schemas (1) public Aggregates Collations Domains FTS Configurations FTS Dictionaries AA FTS Parsers FTS Templates Foreign Tables Functions Materialized Views Operators Procedures Sequences Tables (5) accounts Columns (7) id name website lat long

Query Pad

Query History Notifications

```
147
148 SELECT id, account_id, occurred_at, channel, CASE WHEN channel = 'facebook' THEN 'yes' ELSE 'no' END AS is_facebook
149 FROM web_events
150 ORDER BY occurred_at;
```

Data Output

	<b>id</b>	<b>account_id</b>	<b>occurred_at</b>	<b>channel</b>	<b>is_facebook</b>
1	2471	2861	2013-12-04 04:18:29	direct	no
2	4193	4311	2013-12-04 04:44:58	direct	no
3	8825	4311	2013-12-04 08:27:55	adwords	no
4	6994	2861	2013-12-04 18:22:04	facebook	yes
5	294	1281	2013-12-05 20:17:50	direct	no
6	4728	1281	2013-12-05 21:22:29	adwords	no
7	1998	2481	2013-12-06 02:00:07	direct	no
8	7587	3251	2013-12-06 07:52:09	organic	no
9	6499	2481	2013-12-06 11:48:58	direct	no
10	3186	3491	2013-12-06 12:31:12	direct	no
11	2996	3251	2013-12-06 12:52:53	direct	no
12	6500	2481	2013-12-06 13:11:15	direct	no
13	7870	3491	2013-12-06 16:57:12	facebook	yes
14	7796	3491	2013-12-06 23:07:01	twitter	no
15	3317	3491	2013-12-06 23:22:08	direct	no
16	2361	2731	2013-12-08 05:00:15	direct	no
17	6821	2731	2013-12-08 03:00:03	direct	no
18	5702	1881	2013-12-08 05:53:57	banner	no
19	4368	4491	2013-12-08 06:24:09	direct	no
20	353	1301	2013-12-08 07:03:01	direct	no
21	7444	3141	2013-12-08 15:59:45	adwords	no

Total rows: 1000 of 9073 Query complete 00:00:00.158 Ln 448, Col 1

```
SELECT id, account_id, occurred_at, channel,
CASE WHEN channel = 'facebook' OR channel = 'direct' THEN 'yes' ELSE 'no' END AS is_facebook
FROM web_events
ORDER BY occurred_at;
```

pgAdmin 4

File Object Tools Help

Object Explorer    Dashboard Properties SQL Statistics Dependencies Dependents Processes parch-and-posey/postgres@PostgreSQL 16\*

Servers (1) PostgreSQL 16 Databases (4) R pagila parch-and-posey Casts Catalogs Event Triggers Extensions Foreign Data Wrappers Languages Publications Schemas (1) public Aggregates Collations Domains FTS Configurations FTS Dictionaries AA FTS Parsers FTS Templates Foreign Tables Functions Materialized Views Operators Procedures Sequences Tables (5) accounts Columns (7) id name website lat long

Query Pad

Query History Notifications

```
151
152 SELECT id, account_id, occurred_at, channel,
153 CASE WHEN channel = 'facebook' OR channel = 'direct' THEN 'yes' ELSE 'no' END AS is_facebook
154 FROM web_events
155 ORDER BY occurred_at;
```

Data Output

	<b>id</b>	<b>account_id</b>	<b>occurred_at</b>	<b>channel</b>	<b>is_facebook</b>
1	2471	2861	2013-12-04 04:18:29	direct	yes
2	4193	4311	2013-12-04 04:44:58	direct	yes
3	8825	4311	2013-12-04 08:27:55	adwords	no
4	6994	2861	2013-12-04 18:22:04	facebook	yes
5	294	1281	2013-12-05 20:17:50	direct	yes
6	4728	1281	2013-12-05 21:22:29	adwords	no
7	1998	2481	2013-12-06 02:00:07	direct	yes
8	7587	3251	2013-12-06 07:52:09	organic	no
9	6499	2481	2013-12-06 11:48:58	direct	yes
10	3186	3491	2013-12-06 12:31:12	direct	yes
11	2996	3251	2013-12-06 12:52:53	direct	yes
12	6500	2481	2013-12-06 13:11:15	direct	yes
13	7870	3491	2013-12-06 16:57:12	facebook	yes
14	7796	3491	2013-12-06 23:07:01	twitter	no
15	3317	3491	2013-12-06 23:22:08	direct	yes
16	2361	2731	2013-12-08 05:00:15	direct	yes
17	6821	2731	2013-12-08 03:00:03	direct	yes
18	5702	1881	2013-12-08 05:53:57	banner	no
19	4368	4491	2013-12-08 06:24:09	direct	yes
20	353	1301	2013-12-08 07:03:01	direct	yes

Total rows: 1000 of 9073 Query complete 00:00:00.172 Ln 452, Col 1

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;

```

The screenshot shows the pgAdmin 4 interface. In the Object Explorer, the 'Tables (5)' node is expanded, showing the 'accounts' table with columns: id, name, website, lat, and long. In the main window, a query is written in the SQL tab:

```

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;

```

The Data Output tab displays the results of the query:

account_id	occurred_at	total	total_group
1	2015-10-06 17:31:14	169	101-300
2	2015-11-05 09:34:33	288	101-300
3	2015-12-04 04:21:55	132	101-300
4	2016-01-02 01:18:24	176	101-300
5	2016-02-01 19:27:27	165	101-300
6	2016-03-02 15:29:22	173	101-300
7	2016-04-01 11:20:18	226	101-300
8	2016-05-01 15:55:51	293	101-300
9	2016-05-31 21:22:48	129	101-300
10	2016-06-30 12:32:05	148	101-300
11	2016-07-30 02:26:30	137	101-300
12	2016-08-28 07:13:39	196	101-300
13	2016-09-26 23:28:25	158	101-300
14	2016-10-26 20:31:30	294	101-300
15	2016-11-25 23:27:32	210	101-300
16	2016-12-24 05:53:13	269	101-300
17	2016-12-21 10:59:34	541	Over 500

Total rows: 1000 of 6912 Query complete 00:00:00.114

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. But it'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;

```

The screenshot shows the pgAdmin 4 interface. In the Object Explorer, there is a connection to 'parch-and-posey/postgres@PostgreSQL 16'. The 'Tables' node under 'public' contains a single table named 'orders'. In the main window, a query is being run:

```

166
167 SELECT
168     CASE
169         WHEN total > 500 THEN 'Over 500'
170         ELSE '500 or under'
171     END AS total_group,
172     COUNT(*) AS order_count
173     FROM orders
174     GROUP BY
175         CASE
176             WHEN total > 500 THEN 'Over 500'
177             ELSE '500 or under'
178         END
179     ORDER BY 1;

```

The 'Data Output' tab shows the results of the query:

total_group	order_count
500 or under	3716
Over 500	3196

Total rows: 2 of 2 Query complete 00:00:00.052

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;

```

The screenshot shows the pgAdmin 4 interface. The left sidebar displays the Object Explorer with a tree view of databases, tables, and columns. The main area contains a query editor window with the following SQL code:

```

181 /*
182 43. Write a query to display for each order, the account ID, total amount of the order, and the level of the
183 order - 'Large' or 'Small' - depending on if the order is $3000 or more, or smaller than $3000.
184 */
185 SELECT o.account_id, o.total,
186 CASE
187 WHEN o.total >= 3000 THEN 'Large'
188 ELSE 'Small'
189 END AS order_level
190 FROM orders AS o
191 ORDER BY 1;
192

```

Below the code, the Data Output tab is selected, showing a table with the following data:

account_id	total	order_level
1	1001	294
2	1001	1205
3	1001	1347
4	1001	1384
5	1001	1238
6	1001	1343
7	1001	1254
8	1001	1267
9	1001	1307
10	1001	1280
11	1001	1405
12	1001	137
13	1001	196
14	1001	158
15	1001	169

Total rows: 1000 of 6912 Query complete 00:00:00.075

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;

The screenshot shows the pgAdmin 4 interface. The left sidebar displays the Object Explorer with various database objects like servers, databases, tables, and functions. The main area is a query editor titled 'parch-and-posey/postgres@PostgreSQL 16'. It contains a SQL query and its execution results. The query is:

```

192 /*
193 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.
194 The three categories, based on the total number of items in each order. The three categories are : 'Atleast 2000',
195 'Between 1000 and 2000' and 'Less than 1000'.
196 */
197
198 SELECT
199     CASE WHEN o.total >= 2000 THEN 'At Least 2000'
200         WHEN o.total BETWEEN 1000 AND 1999 THEN 'Between 1000 AND 2000'
201         WHEN o.total <1000 THEN 'Less than 1000'
202     END AS order_category,
203     COUNT(*) AS order_count
204 FROM orders AS o
205 GROUP BY 1;
206

```

The 'Data Output' tab shows the results of the query:

order_category	order_count
Between 1000 AND 2000	511
Less than 1000	6331
At Least 2000	70

Total rows: 3 of 3 Query complete 00:00:00.119

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;

```

pgAdmin 4

File Object Tools Help

Object Explorer    Dashboard Properties SQL Statistics Dependencies Dependents Processes    parch-and-posey/postgres@PostgreSQL 16\*

Server (1) PostgreSQL 16 Databases (4) R pagila parch-and-posey Casts Catalogs Event Triggers Extensions Foreign Data Wrappers Languages Publications Schemas (1) public Aggregates Collations Domains FTS Configurations FTS Dictionaries FTS Parsers FTS Templates Foreign Tables Functions Materialized Views Operators Procedures Sequences Tables (5) accounts Columns (7) id name website lat long

Query Pad    Scratch Pad    Query History Messages Notifications

```

508 45. We would like to understand 3 different branches of customers based on the amount associated with their purchases.
509 The top branch includes anyone with a Lifetime Value (total sales of all orders) greater than 200,000 usd. The second branch
510 is between 200,000 and 100,000 usd. The lowest branch is anyone under 100,000 usd. Provide a table that includes the level
511 associated with each account. You should provide the account name, the total sales of all orders for the customer, and the level
512 Order with the top spending customers listed first.
513 /
514 SELECT a.name, SUM(total_amt_usd) AS total_spent,
515 CASE WHEN SUM(total_amt_usd) > 200000 THEN 'top'
516 WHEN SUM(total_amt_usd) > 100000 THEN 'middle'
517 ELSE 'low' END AS customer_level
518 FROM orders AS o
519 JOIN accounts AS a
520 ON o.account_id = a.id
521 GROUP by a.name
522 ORDER BY 2 DESC;
523

```

Data Output

	name	total_spent	customer_level
1	EOG Resources	382873.30	top
2	Mosaic	345618.59	top
3	IBM	326819.48	top
4	General Dynamics	300694.79	top
5	Republic Services	293861.14	top
6	Leucadia National	291047.25	top
7	Arrow Electronics	281018.36	top
8	Sysco	278575.64	top
9	Supervalu	275288.30	top
10	Archer Daniels Midland	272672.84	top

Total rows: 350 of 350    Query complete 00:00:00.302

Ln 507, Col 1

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;

```

pgAdmin 4

File Object Tools Help

Object Explorer    Dashboard Properties SQL Statistics Dependencies Dependents Processes    parch-and-posey/postgres@PostgreSQL 16\*

Server (1) PostgreSQL 16 Databases (4) R pagila parch-and-posey Casts Catalogs Event Triggers Extensions Foreign Data Wrappers Languages Publications Schemas (1) public Aggregates Collations Domains FTS Configurations FTS Dictionaries FTS Parsers FTS Templates Foreign Tables Functions Materialized Views Operators Procedures Sequences Tables (5) accounts Columns (7) id name website lat long

Query Pad    Scratch Pad    Query History Messages Notifications

```

524 /*
525 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
526 in 2016 and 2017. Keep the same levels as in the previous question. Order with the top spending customers listed first.
527 */
528 /
529 SELECT a.name, SUM(total_amt_usd) AS total_spent,
530 CASE WHEN SUM(TOTAL_AMT_USD) > 200000 THEN 'top'
531 WHEN SUM(total_amt_usd) > 100000 THEN 'middle'
532 ELSE 'low' END AS customer_level
533 FROM orders AS o
534 JOIN accounts AS a
535 ON o.account_id = a.id
536 WHERE occurred_at > '2015-12-31'
537 GROUP BY 1
538 ORDER BY 2 DESC;
539

```

Data Output

	name	total_spent	customer_level
1	Pacific Life	255319.18	top
2	Mosaic	172180.04	middle
3	OHS	160471.78	middle
4	Core-Mark Holding	148105.93	middle
5	Disney	129157.38	middle
6	National Oilwell Varco	121873.16	middle
7	Sears Holdings	114003.21	middle
8	State Farm Insurance Cos.	111810.55	middle
9	Fidelity National Financial	110027.29	middle
10	BB&T Corp.	107900.05	middle
11	Arrow Electronics	105426.87	middle
12	United States Steel	104452.39	middle

Total rows: 322 of 322    Query complete 00:00:00.104

Ln 525, Col 1

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;

The screenshot shows the pgAdmin 4 interface. The left sidebar displays the Object Explorer with a tree view of databases, tables, and columns. The main area is a query editor containing the SQL code for question 47. Below the query editor is a Data Output window showing the results of the query. The results are a table with three columns: name, num\_orders, and sales\_rep\_level. The data is as follows:

	name	num_orders	sales_rep_level
1	Earlie Schleusner	335	top
2	Vernita Plump	299	top
3	Tia Amato	267	top
4	Georgianne Chisholm	256	top
5	Moon Torian	250	top
6	Nelle Meaux	241	top
7	Maren Musto	224	top
8	Dorotha Seawell	208	top
9	Charles Bidwell	205	top
10	Maryanna Florentino	204	top
11	Calvin Ollison	199	not

Total rows: 50 of 50 Query complete 00:00:00.103 Ln 540, Col 1

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) > 500000 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;

```

The screenshot shows the pgAdmin 4 interface. The left sidebar displays the Object Explorer with various database objects like servers, databases, tables, and functions. The main area is the Query Editor containing a SQL query. The Data Output tab below it shows the results of the query.

```

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) > 75000 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;

```

	name	character	count	begin	total_spent	sales_rep_level
1	Earlie Schleusner	character	335	1098137.72	top	
2	Tia Amato	character	257	1010690.60	top	
3	Vernita Plump	character	299	934212.93	top	
4	Georgianna Chisholm	character	256	886244.12	top	
5	Arica Stoltzfus	character	186	810353.34	top	
6	Dorotha Seawell	character	208	766935.04	top	
7	Nelle Meaux	character	241	749076.16	top	
8	Sibyl Lauria	character	193	722084.27	middle	
9	Adrienne Sikkens	character	174	707647.70	middle	
10	Leanne Ladd	character	174	707647.70	middle	

Total rows: 50 of 50    Query complete 00:00:00.129    Ln 557, Col 1