SQL for Data Analysis – Tutorial – ep6 (advanced stuff)

Written by Tomi Mester on August 7, 2017

Last updated on January 11, 2021

Today I'll show you some more advanced SQL moves! Okay, I'd rather put them into the *intermediate* category. They are not beginner things — but it's also for sure that junior data scientists can't live without them.

In this article, I'll show you:

- subqueries,
- SQL `CASE` and
- SQL `HAVING`

These will definitely bring your SQL game to the next level and make you more efficient at your daily job. *(Or if you don't have a job yet, they will help impress the interviewer at your SQL screening interview.)*

Note: to get the most out of this article, I recommend not just reading it, but actually doing the coding part with me!

Before we start…

… I suggest going through these articles first – if you haven't done so yet:

1. Set up your own data server to practice: How to set up Python, SQL, R and Bash (for non-devs)
2. Install SQL Workbench to manage your SQL stuff better: How to install SQL Workbench for postgreSQL
3. SQL for Data Analysis ep1 (SQL basics)
4. SQL for Data Analysis ep2 (SQL WHERE clause)
5. SQL for Data Analysis ep3 (SQL functions and GROUP BY)
6. SQL for Data Analysis ep4 (SQL best practices)
7. SQL for Data Analysis ep5 (SQL JOIN)

SQL subqueries (query within a query)

There are some cases in which a simple SQL query isn't enough to answer your question. Open your SQL Workbench and let's start with an easy task (we will use our `flight_delays` data set again):

Select the average departure delay by tail numbers (`tailnum` column) from the table – and return the minimum and maximum values of these calculated averages.

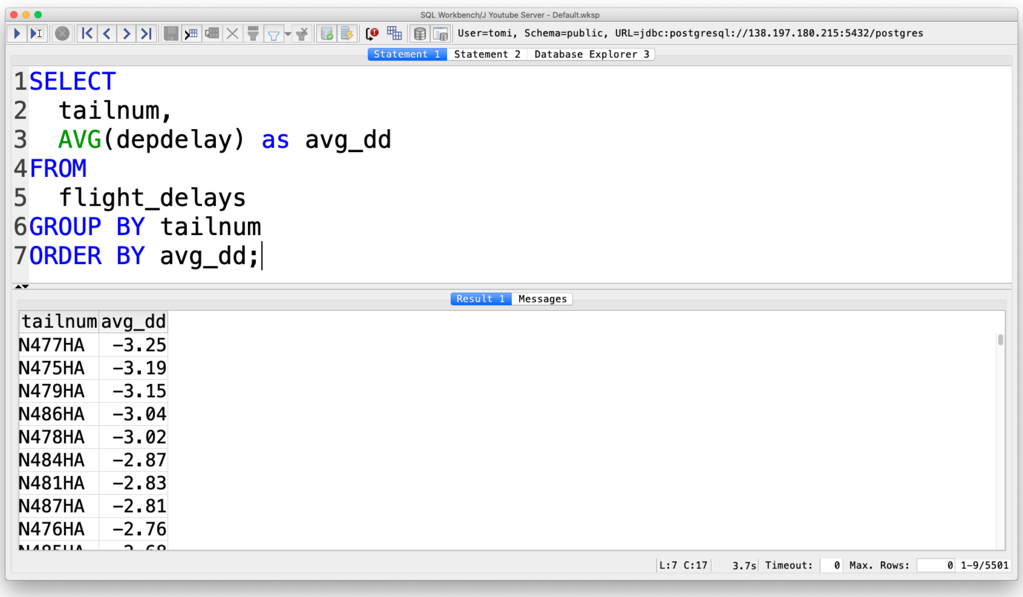
Note: Let's try to solve it with the tools we have learned so far!

This is the solution:

```
SELECT
  tailnum,
  AVG(depdelay) AS avg_dd
FROM
  flight_delays
GROUP BY tailnum
ORDER BY avg_dd;
```

Well, at least… as far as we can get with the tools we've learned so far.

The result is:


SQL Workbench/J Youtube Server - Default.wksp
User=tomi, Schema=public, URL=jdbc:postgresql://138.197.180.215:5432/postgres
Statement 1
Statement 2
Database Explorer 3
Result 1
Messages

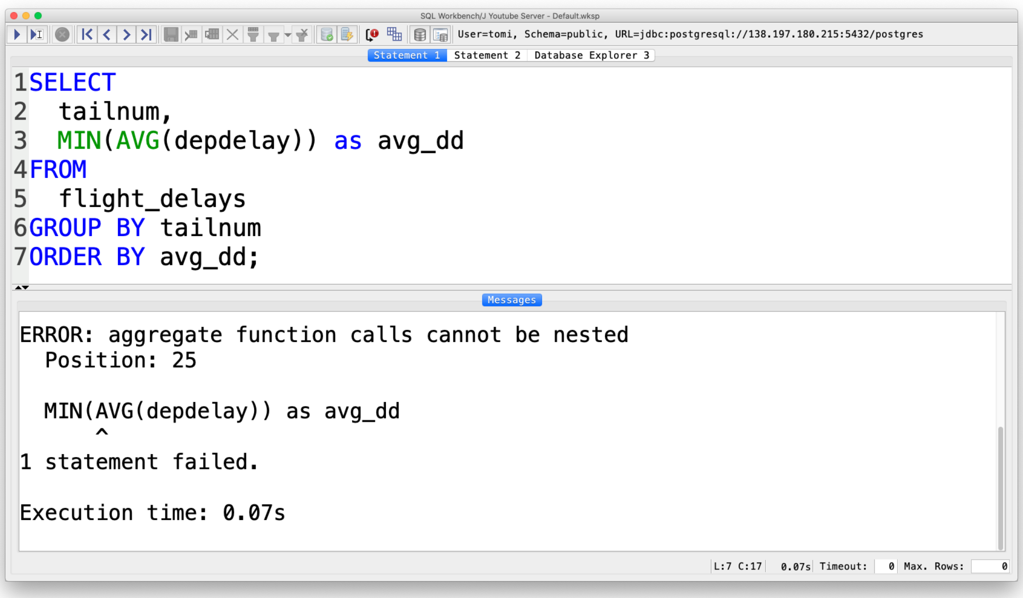
```
SELECT
  tailnum,
  AVG(depdelay) as avg_dd
FROM
  flight_delays
GROUP BY tailnum
ORDER BY avg_dd;
```

```
tailnum avg_dd
N477HA   -3.25
N475HA   -3.19
N479HA   -3.15
N486HA   -3.04
N478HA   -3.02
N484HA   -2.87
N481HA   -2.83
N487HA   -2.81
N476HA   -2.76
```

```
L:7 C:17   3.7s Timeout: 0 Max. Rows: 0 1-9/5501
```

Nice. Except that this is not the 100% correct answer for this question. Sure, you can scroll down to see the highest value in the `avg_dd` column and you can scroll back up for the lowest value. But I don't like that. It's not just inelegant, but in real life data science projects quite often it's not enough. If your automated script needs two specific values, you can't scroll through manually. You'll need to provide those two specific values. Period.

SQL-wise, the problem is that you have already run a function on your original table (`AVG(depdelay)`). And in SQL, you can't use *nested* functions, like `MIN(AVG(depdelay))` . You can even give it a try:


SQL Workbench/J Youtube Server - Default.wksp
User=tomi, Schema=public, URL=jdbc:postgresql://138.197.180.215:5432/postgres
Statement 1
Statement 2
Database Explorer 3
Messages
L:7 C:17
0.07s
Timeout:
0
Max. Rows:
0

```
SELECT
  tailnum,
  MIN(AVG(depdelay)) as avg_dd
FROM
  flight_delays
GROUP BY tailnum
ORDER BY avg_dd;
```

```
ERROR: aggregate function calls cannot be nested
  Position: 25

  MIN(AVG(depdelay)) as avg_dd
      ^

1 statement failed.

Execution time: 0.07s
```

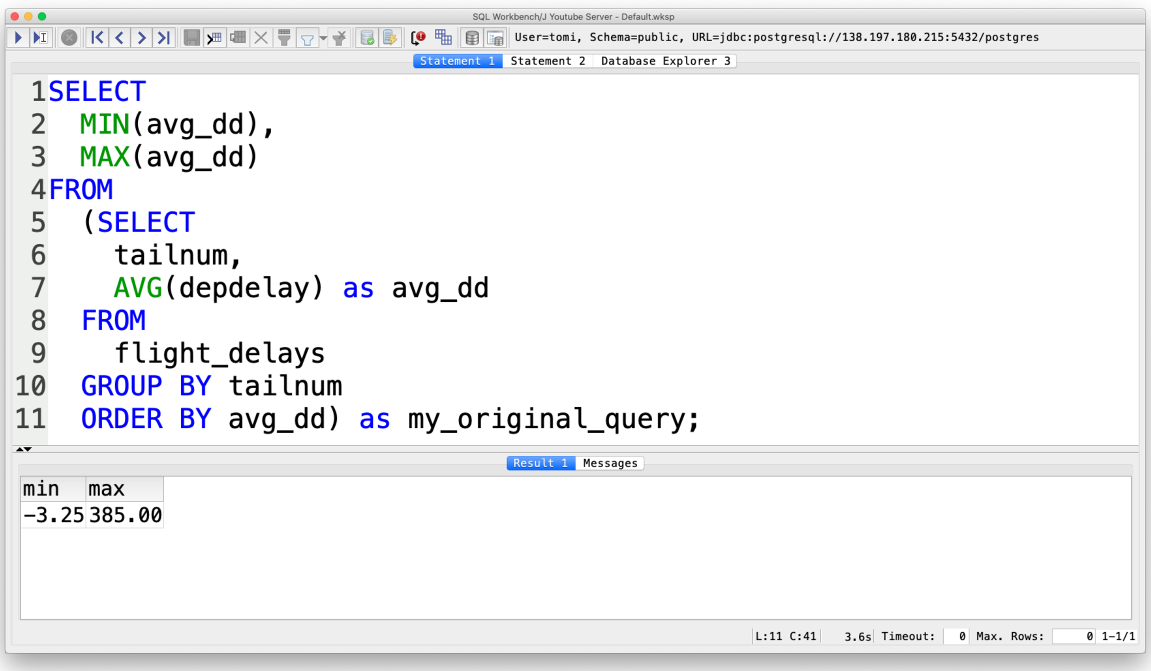
It'll just throw an error.

At this point you can't solve the problem with your original query, so you have two choices:

1. You can create a new table and put this result there. Then you can run your `MIN()` and `MAX()` functions on that freshly created table… This is not the preferred way, because it's a bit complicated – and it's not too flexible either.
2. A better solution is to *handle* the result of this query as if it *was* a new table – but in reality, put it as a subquery into another query.

Like this:

```
SELECT
  MIN(avg_dd),
  MAX(avg_dd)
FROM
  (SELECT
     tailnum,
     AVG(depdelay) AS avg_dd
   FROM
     flight_delays
   GROUP BY tailnum
   ORDER BY avg_dd) AS my_original_query;
```


SQL Workbench/J Youtube Server - Default.wksp
User=tomi, Schema=public, URL=jdbc:postgresql://138.197.180.215:5432/postgres
Statement 1
Statement 2
Database Explorer 3
Result 1
Messages
L:11 C:41
3.6s
Timeout:
0
Max. Rows:
0
1-1/1

```
SELECT
  MIN(avg_dd),
  MAX(avg_dd)
FROM
  (SELECT
    tailnum,
    AVG(depdelay) as avg_dd
  FROM
    flight_delays
  GROUP BY tailnum
  ORDER BY avg_dd) as my_original_query;
```

```
min   max
-3.25 385.00
```

If it looks complicated let me simplify it:

```
SELECT
  MIN(avg_dd),
  MAX(avg_dd)
FROM
  ([your_first_query]) AS [some_name];
```

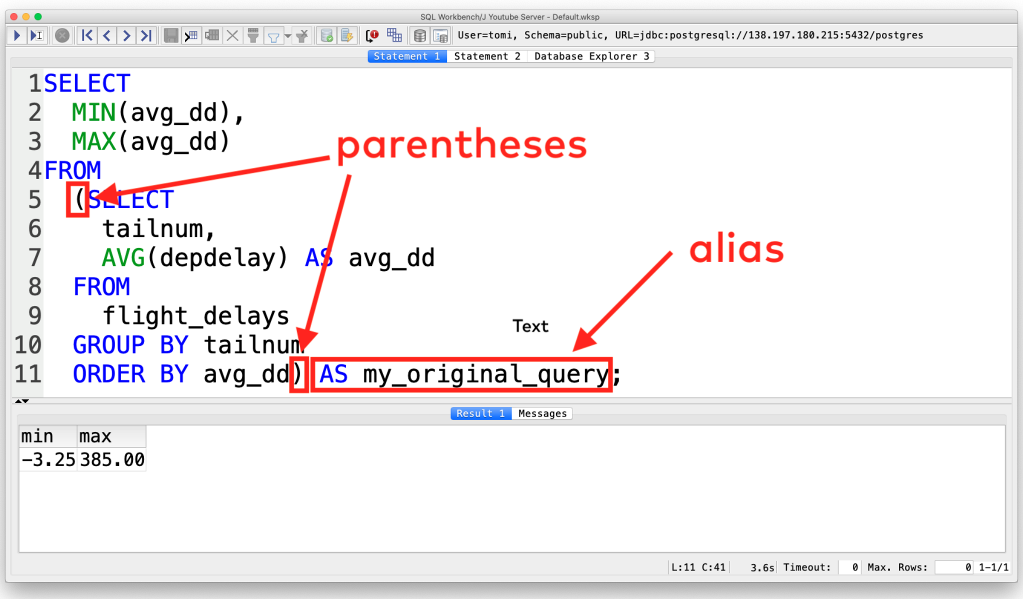
Pretty cool. We have just gotten the results and you have just learned that after the `FROM` keyword you can always use subqueries instead of tables. And just to let you know, it doesn't apply only to the `FROM` keyword (e.g. you can do it with `WHERE`, too.)

Note 1: While this solution is very handy, I have to admit that sometimes creating new tables is more efficient in terms of CPU-time. That would be a bit too advanced of a topic, so I won't go deeper into it in this article.

SQL subquery syntax requirements

When you use subqueries in SQL, you have to be very careful with the syntax. To be more specific, watch out for these two things:

- your inner query must go between parentheses (`(` and `)`)
- your inner query must get a new "name" with aliasing — so after the parentheses add this: `AS new_name` ... except that of course you should replace `new_name` with something more meaningful. 🙂


SQL Workbench/J Youtube Server - Default.wksp
User=tomi, Schema=public, URL=jdbc:postgresql://138.197.180.215:5432/postgres
Statement 1
Statement 2
Database Explorer 3
parentheses
alias
Text
Result 1
Messages
L:11 C:41
3.6s
Timeout: 0
Max. Rows: 0
1-1/1

```
SELECT
  MIN(avg_dd),
  MAX(avg_dd)
FROM
  (SELECT
    tailnum,
    AVG(depdelay) AS avg_dd
  FROM
    flight_delays
  GROUP BY tailnum
  ORDER BY avg_dd) AS my_original_query;
```

```
min   max
-3.25 385.00
```

SQL subquery syntax requirements

Test yourself #1 (with subqueries)

If you get the concept, here's another SQL exercise to test yourself with!

- **Calculate the total of the distances flown by each plane... *(Note: do the grouping based on the* `tailnum` *column — and* `SUM` *the values in the* `distance` *column)***
- **...then take only those planes for which this total distance is greater than 1,000,000...**
- **then calculate the average of the total distances flown by these planes!**

Hint 1: this might be a somewhat complex query. I recommend first trying to sketch and design on paper how it will work!

Hint 2: The result will be one number: `1311745.90`

3... 2... 1... Go!

.

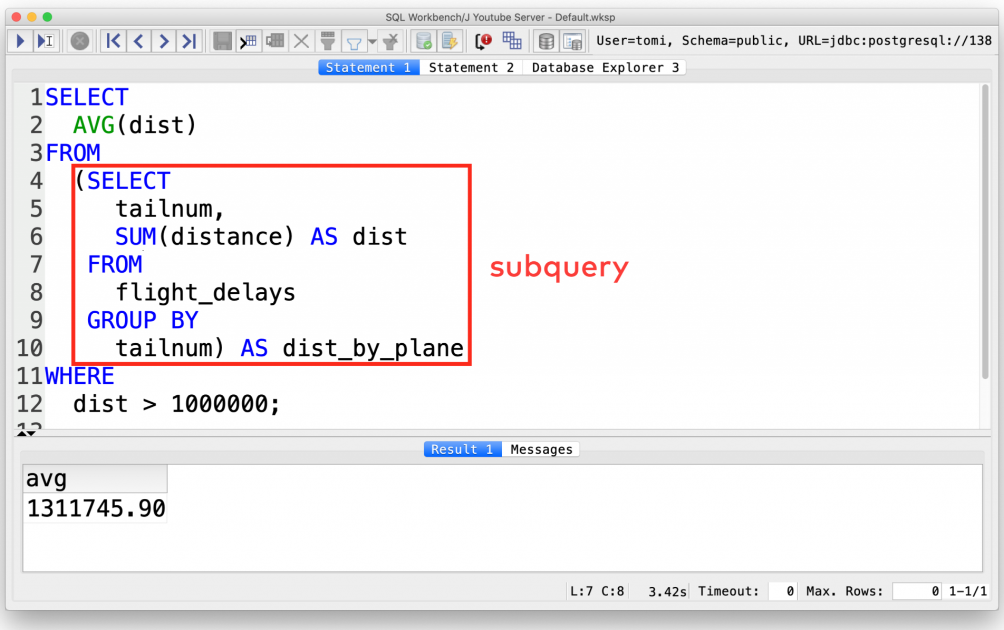
.

.

Here's my solution...

(… but I'm pretty sure that there are other SQL queries that would deliver the same result. And you are more than welcome to post them in the comment section below the article!)

```
SELECT
  AVG(dist)
FROM
  (SELECT
     tailnum,
     SUM(distance) AS dist
   FROM
     flight_delays
   GROUP BY
     tailnum) AS dist_by_plane
WHERE dist > 1000000;
```



```
SELECT
  AVG(dist)
FROM
  (SELECT
     tailnum,
     SUM(distance) AS dist
   FROM
     flight_delays
   GROUP BY
     tailnum) AS dist_by_plane
WHERE
  dist > 1000000;
```

```
avg
1311745.90
```

the solution (inner query highlighted)

See? Nothing new has happened here – only that we put a query into another query. That's what SQL subqueries are about.

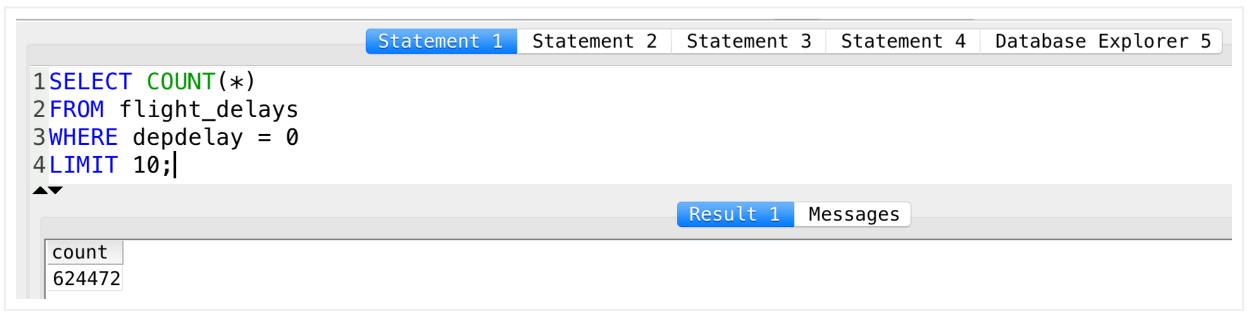
By the way, there is an alternative solution for this task using SQL `HAVING` … And I'll get back to that soon.

SQL CASE (the "if statement" of SQL)

Let's say we want to see how many planes departed early, late or on time. With our current toolset we can do this by running 3 queries – one by one.

E.g. the number of flights that departed right on time would look like this:

```
SELECT COUNT(*)
FROM flight_delays
WHERE depdelay = 0
LIMIT 10;
```



```
SELECT COUNT(*)
FROM flight_delays
WHERE depdelay = 0
LIMIT 10;
```

```
count
624472
```

To have all the variations you can change the `=` in the `WHERE` filter to `<` and to `>`.

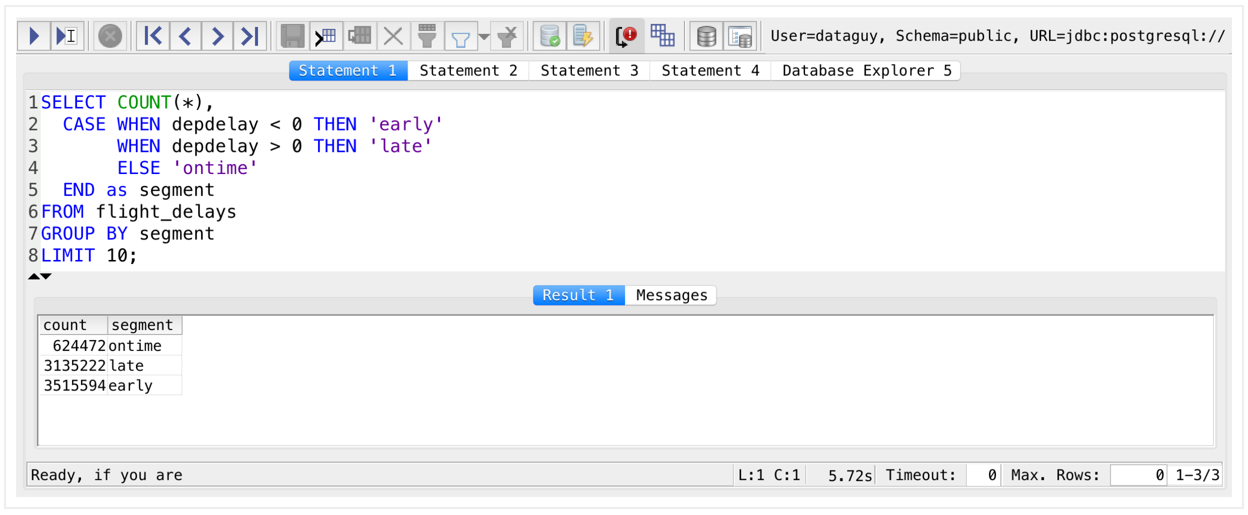
The results will be:

- Departed early: `3515594`

- Departed on time: `624472`
- Departed late: `3135222`

Good news! You can also do this using a single (slightly more advanced) SQL query if you use the `CASE` SQL keyword. Try this:

```
SELECT COUNT(*),
  CASE WHEN depdelay < 0 THEN 'early'
       WHEN depdelay > 0 THEN 'late'
       ELSE 'ontime'
  END as segment
FROM flight_delays
GROUP BY segment
LIMIT 10;
```



User=dataguy, Schema=public, URL=jdbc:postgresql://

```
SELECT COUNT(*),
  CASE WHEN depdelay < 0 THEN 'early'
       WHEN depdelay > 0 THEN 'late'
       ELSE 'ontime'
  END as segment
FROM flight_delays
GROUP BY segment
LIMIT 10;
```

```
count   segment
 624472 ontime
3135222 late
3515594 early
```

```
Ready, if you are    L:1 C:1    5.72s  Timeout:  0  Max. Rows:  0  1-3/3
```

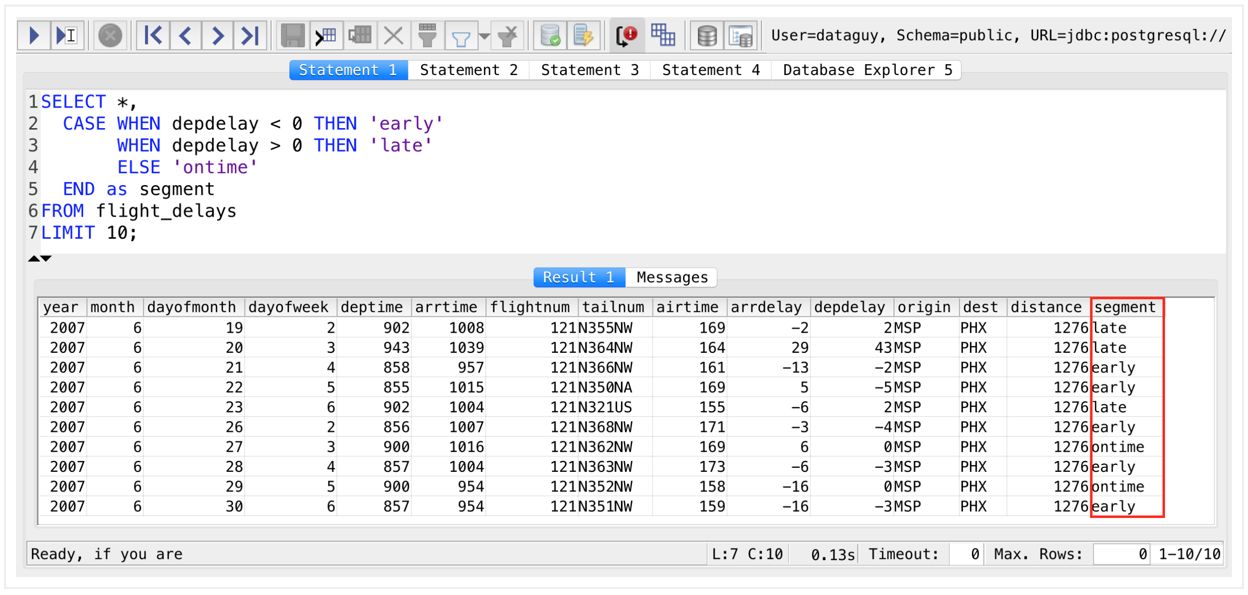
Nice, huh?
Let's take a look at the new part:

```
CASE WHEN depdelay < 0 THEN 'early'
     WHEN depdelay > 0 THEN 'late'
     ELSE 'ontime'
END as segment
```

If you are familiar with if statements in bash or in Python, it's not too hard to understand what's happening here: the `CASE WHEN` part of your query categorizes the flights, based on the `depdelay` column, into 3 categories.

Note: if it's not clear to you at first sight, I recommend running the query without the `GROUP BY`*, like this:*

```
SELECT *,
  CASE WHEN depdelay < 0 THEN 'early'
       WHEN depdelay > 0 THEN 'late'
       ELSE 'ontime'
  END as segment
FROM flight_delays
LIMIT 10;
```



User=dataguy, Schema=public, URL=jdbc:postgresql://

```
SELECT *,
  CASE WHEN depdelay < 0 THEN 'early'
       WHEN depdelay > 0 THEN 'late'
       ELSE 'ontime'
  END as segment
FROM flight_delays
LIMIT 10;
```

```
year month dayofmonth dayofweek deptime arrtime flightnum tailnum airtime arrdelay depdelay origin dest distance segment
2007     6         19         2     902    1008       121 N355NW      169       -2        2 MSP    PHX      1276 late
2007     6         20         3     943    1039       121 N364NW      164       29       43 MSP    PHX      1276 late
2007     6         21         4     858     957       121 N366NW      161      -13       -2 MSP    PHX      1276 early
2007     6         22         5     855    1015       121 N350NA      169        5       -5 MSP    PHX      1276 early
2007     6         23         6     902    1004       121 N321US      155       -6        2 MSP    PHX      1276 late
2007     6         26         2     856    1007       121 N368NW      171       -3       -4 MSP    PHX      1276 early
2007     6         27         3     900    1016       121 N362NW      169        6        0 MSP    PHX      1276 ontime
2007     6         28         4     857    1004       121 N363NW      173       -6       -3 MSP    PHX      1276 early
2007     6         29         5     900     954       121 N352NW      158      -16        0 MSP    PHX      1276 ontime
2007     6         30         6     857     954       121 N351NW      159      -16       -3 MSP    PHX      1276 early
```

```
Ready, if you are    L:7 C:10    0.13s  Timeout:  0  Max. Rows:  0  1-10/10
```

See? The trick is that the `CASE WHEN` *statement is creating a new column at the end of the table.*

To be honest I don't use the SQL `CASE` statement too often during my daily job… but sometimes it's very handy when I have to do some quick and dirty data cleaning/transformation, turn a continuous value into a categorical value (like we did in the above example), and so on…

SQL HAVING

You will need SQL HAVING for complex queries.

`HAVING` is for filtering on the results of SQL functions like `COUNT()`, `SUM()`, `AVG()`, etc.

But wait? Isn't the WHERE keyword for filtering in SQL?

Yes but here's the thing: in SQL, you can't use `WHERE` with the results of aggregate functions. Why is that? Here's a short explanation!

Let's say we want to query something simple from our table: how many times an airport shows up in the table. You should go with this query:

```
SELECT COUNT(*) as nmbr,
       dest
FROM flight_delays
GROUP BY dest;
```

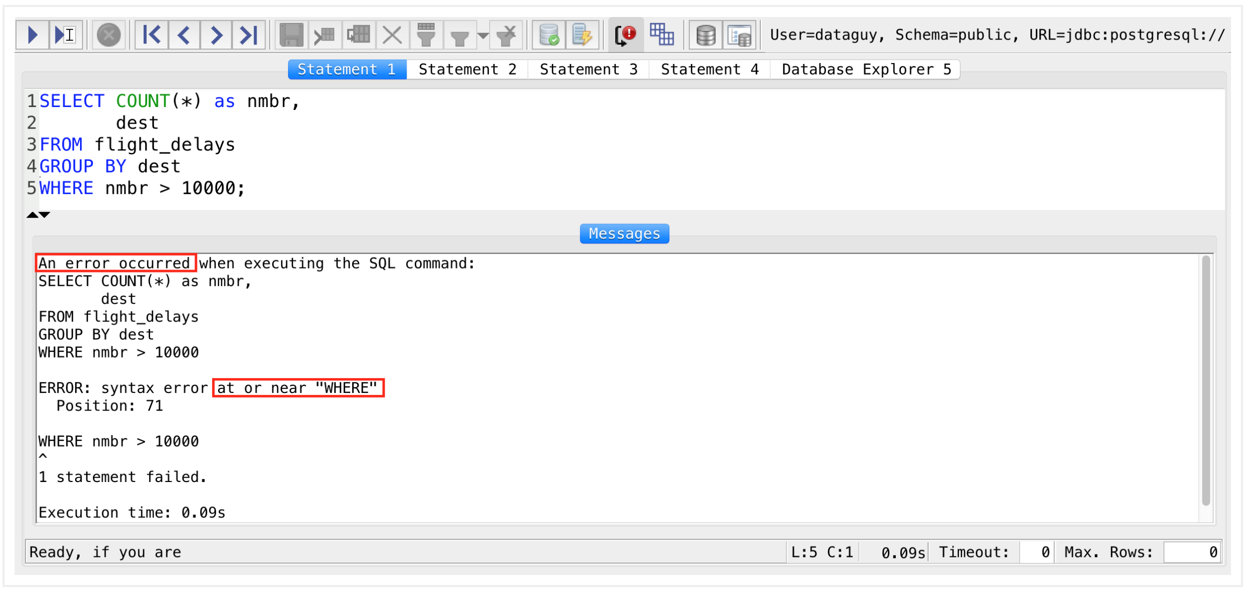
Good.

Now we want to see only those airports that show up more than 10,000 times (means that the result of `COUNT(*)` is greater than `10000`).

Someone new to SQL would try this *(and I have to admit it might seem to be highly logical at first…)*

BUT THIS QUERY WON'T(!) WORK:

```
SELECT COUNT(*) as nmbr,
       dest
FROM flight_delays
GROUP BY dest
WHERE nmbr > 10000;
```



```
SELECT COUNT(*) as nmbr,
       dest
FROM flight_delays
GROUP BY dest
WHERE nmbr > 10000;
```

```
An error occurred when executing the SQL command:
SELECT COUNT(*) as nmbr,
       dest
FROM flight_delays
GROUP BY dest
WHERE nmbr > 10000

ERROR: syntax error at or near "WHERE"
  Position: 71

WHERE nmbr > 10000
^
1 statement failed.

Execution time: 0.09s
```

```
Ready, if you are                    L:5 C:1   0.09s Timeout:   0 Max. Rows:     0
```

Yes, it looks logical, but again: this query won't work. Why? The answer is in my previous article (SQL best practices)… But I am going to highlight it here as well. This is the order of the SQL keywords your computer sees when processing your query:

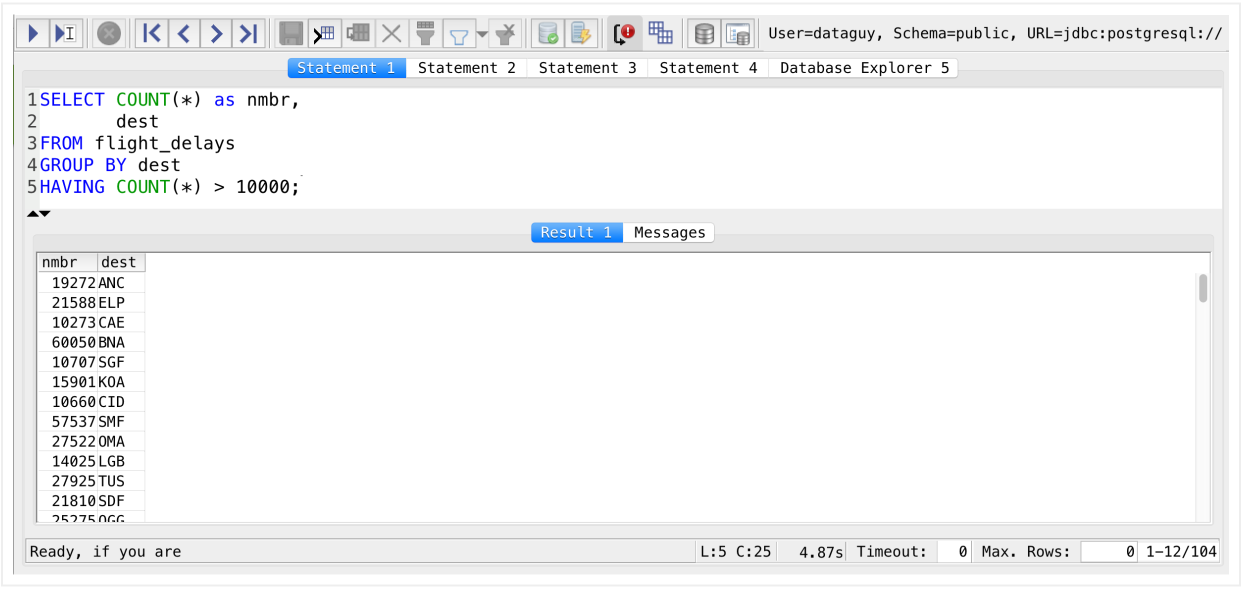
1. `FROM`
2. `ON`

3. `JOIN`
4. `WHERE`
5. `GROUP BY`
6. `SELECT`
7. `DISTINCT`
8. `ORDER BY`
9. `LIMIT`

As you can see, `WHERE` comes before `GROUP BY` and `SELECT`. This means that any aggregation column created by your `GROUP BY` clause can't be part of your `WHERE` filter. And that's where the SQL HAVING keyword comes into the picture. `HAVING` is the new `WHERE` – at least for the aggregated values in your SQL query. (Okay, this sounds very nerdy.)

You have to change your previous *wrong* query to this:

```
SELECT COUNT(*) as nmbr,
       dest
FROM flight_delays
GROUP BY dest
HAVING COUNT(*) > 10000;
```



```
SELECT COUNT(*) as nmbr,
       dest
FROM flight_delays
GROUP BY dest
HAVING COUNT(*) > 10000;
```

```
nmbr  dest
19272 ANC
21588 ELP
10273 CAE
60050 BNA
10707 SGF
15901 KOA
10660 CID
57537 SMF
27522 OMA
14025 LGB
27925 TUS
21810 SDF
25275 OGG
```

```
Ready, if you are        L:5 C:25   4.87s  Timeout:   0  Max. Rows:   0  1-12/104
```

And it'll just work!

The only unfortunate thing is that you can't use your aliases with your `HAVING`, so you need to type the whole `COUNT(*)` function again... But we can deal with that!

Test yourself #2 (alternative solution with HAVING)

Let's try to solve the same task that we had in ***TEST YOURSELF #1***... Only this time, try to take advantage of `HAVING`, as well! So:

- **Calculate the total of the distances flown by each plane...** ***(Note: do the grouping based on the*** `tailnum` ***column — and*** `SUM` ***the values in the*** `distance` ***column)***
- **...then take only those planes for which this total distance is greater than 1,000,000...**
- **then calculate the average of the total distances flown by these planes!**

Hint 1: you need to tweak your previous query just a little bit!

Hint 2: The expected result is still: `1311745.90`.

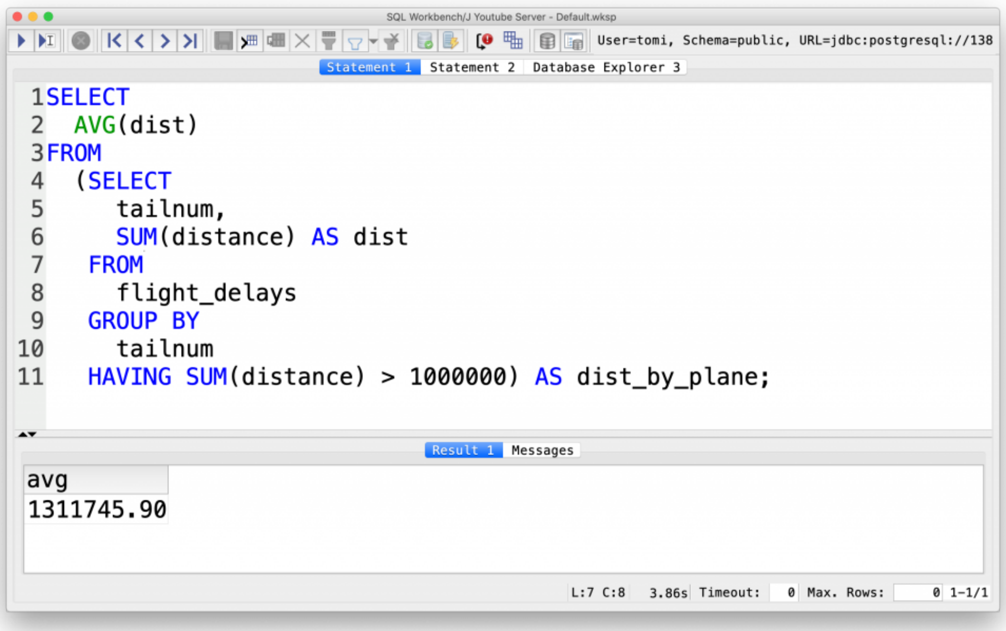
.

.

.

The solution is:

```
SELECT
  AVG(dist)
FROM
  (SELECT
     tailnum,
     SUM(distance) AS dist
   FROM
     flight_delays
   GROUP BY
     tailnum
   HAVING SUM(distance) > 1000000) AS dist_by_plane;
```


SQL Workbench/J Youtube Server - Default.wksp
User=tomi, Schema=public, URL=jdbc:postgresql://138
Statement 1
Statement 2
Database Explorer 3
Result 1
Messages

```
SELECT
  AVG(dist)
FROM
  (SELECT
     tailnum,
     SUM(distance) AS dist
   FROM
     flight_delays
   GROUP BY
     tailnum
   HAVING SUM(distance) > 1000000) AS dist_by_plane;
```

```
avg
1311745.90
```

```
L:7 C:8    3.86s  Timeout:   0  Max. Rows:      0  1-1/1
```

In this version, I've changed only one thing — instead of the `WHERE` in the outer query, I put a `HAVING` in the inner query. The result is the same. There might be slight differences in the computing time of the two solutions… But in this case, it won't really matter.

Challenge: Is there a possible better solution by using `CASE` *?* 😉 *Let me know in the comments!*

Conclusion

Nice! These were some more advanced SQL concepts! And you have gone through them! Even better: you are done with the querying part of the SQL for data analysis tutorial series! Congrats!

If you have read and worked through all 6 episodes, you can proudly say that you know the basics of SQL. (You might also want to check out how to CREATE SQL tables.)

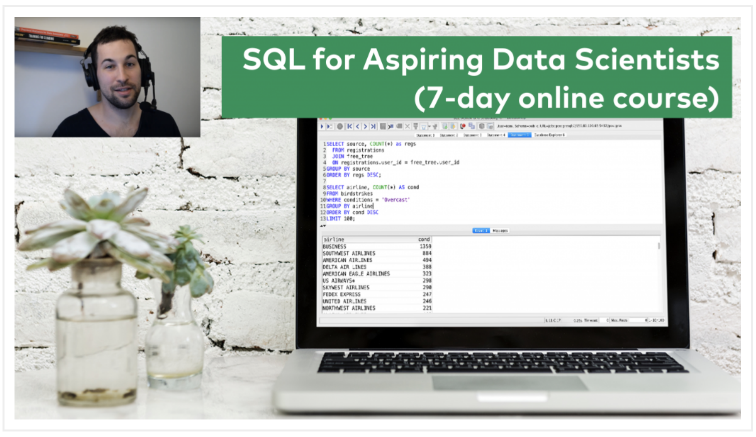
If you want to be better and more confident, let's do these 3 things now:

- practice,
- **practice** and
- **PRACTICE**!

I've created a 7-day SQL course for that, by the way. It doesn't just make you practice everything we learned so far but it also contains 20 job-interview-like SQL questions (and the solutions with explanations). Find more info here:

SQL for Aspiring Data Scientists (7-day online course)

I've created an online course that will take you from zero to intermediate level with SQL in 7 days. Go ahead and check it out here:



More info...

- If you want to learn more about how to become a data scientist, take my 50-minute video course: How to Become a Data Scientist. (It's free!)
- Also check out my 6-week online course: The Junior Data Scientist's First Month video course.

Cheers,
Tomi Mester

August 7, 2017 In Coding In Data Science and Analytics
#analytics #data coding #learn data science #learn to code #small data #sql #sql for data analysis #tomi mester

← PREVIOUS POST NEXT POST →