6. Presenting and Aggregating Your Result

Contents

[1. Overview 1](#_Toc182945537)

[2. Sort Order 1](#_Toc182945538)

[3. Introduction to Aggregate Functions 2](#_Toc182945539)

[4. Filtering Aggregate Results 5](#_Toc182945540)

[5. Understanding Subqueries in PostgreSQL 6](#_Toc182945541)

[6. Database Views in PostgreSQL 8](#_Toc182945542)

[Execute the View 8](#_Toc182945543)

[Dropping the View 9](#_Toc182945544)

[View the Definition of the View 9](#_Toc182945545)

[7. Writing High-Performing SQL Queries 9](#_Toc182945546)

[Optimized Query Example 12](#_Toc182945547)

[Unoptimized Query Example 12](#_Toc182945548)

[8. Query Plans Overview 12](#_Toc182945549)

[9. Identifying Performance Issues in SQL Queries 12](#_Toc182945550)

[10. Common Performance Issues 12](#_Toc182945551)

[11. Summary 12](#_Toc182945552)

# 1. Overview

=>slides: Pg. 1

In this module, we'll build upon what we've learned in earlier modules.

=>slides: Pg. 2

We'll sort the output from queries to make them easier to interpret and more informative, and we'll also apply aggregate calculations to our data. Let's get started.

# Sort Order

=>slides: Pg. 3

When we execute a query against a Postgres database, the database returns the results in a seemingly random order. In fact, the data is returned in the order it is stored in the database. We can, however, sort these results using the ORDER BY keyword. This simple query retrieves data from a table of customers and states of residence. As we can see, these results are not well organized.

=>slides: Pg. 4

We can simply add the ORDER BY keyword and tell Postgres what field want to order by. In this case, customer first name. Notice that the result set is now sorted in ascending order by the first name. If we do not list a sorting direction, ascending is the default order.

=>slides: Pg. 5

We can also specify ascending by typing ORDER BY name ASC. If we want to sort in descending order, we can instead type ORDER BY, name, DESC, for descending. Postgres also allows us to sort by more than one column.

=>slides: Pg. 6

For example, we can sort by both state of residence and name by typing ORDER BY state, name. This statement will sort first by state, but if some rows have the same state of residence, it will order them by name. See, for example, Texas and Oregon. Notice that since we did not specify a direction, both fields are sorted in ascending order. You can even combine columns and sort in different directions.

=>slides: Pg. 7

For example, we may want to sort our states in descending order, but sort the names within each state in ascending order. We could do this by specifying a direction for each column, ORDER BY state DESC, name ASC. In other words, sort state in descending order, but sort name in ascending order.

=>slides: Pg. 8

SQL also provides a shortcut for referencing columns in the ORDER BY statement. Notice that name is column 1 in our SELECT clause, and that state is column 2 in our SELECT clause. Instead of ORDER BY state, name, we could refer to the columns by their position number, ORDER BY 2, 1, 2 being state and 1 being name. This query returns the same result. Although this may be convenient, make sure that you consider readability and interpretability of your code when you use these shortcut methods.

# Introduction to Aggregate Functions

=>slides: Pg. 9

SQL offers a host of functions known as aggregate functions. An aggregate function performs a calculation across a set of values to return a single value. Some of these functions include COUNT, which counts rows in a specified table or view; SUM, which calculates the sum of a given set of values; AVG, which calculates the average of a set of values; MIN, which finds the minimum value in a set of values, and MAX, which finds the maximum value in a set of values.

=>slides: Pg. 10

To use an aggregate function, we include it in our SELECT clause. Let's look at this list of first names, grade levels, and ages for a group of students. For example, to return the average age of these students, we could use this query. The average age of the group is 17. Notice that just like with any other column, we can alias the aggregate calculation. See here that we've renamed the AVG(age) column as avg\_age.

=>slides: Pg. 11

Aggregate functions can be used to perform more sophisticated analysis. For example, what if we want to know what the average age of our group is by grade level?

=>slides: Pg. 12

Aggregate functions work with the GROUP BY keyword to group these result sets. This code tells Postgres that we want to select the grade level and average age, but that we want the average age grouped by grade level. We'll see here that the average age for our freshman is 15, while our juniors are 17 and our seniors have an average age of 19.

=>slides: Pg. 13

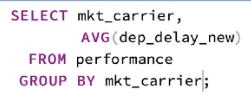
If we wanted to know the minimum age per grade level, we might try this query, but this query would not run and the system would return an error. It is important to note that if you use an aggregate field in addition to other fields in your SELECT statement, all other fields must be listed in the GROUP BY clause. Otherwise, PostgreSQL doesn't know how to group the results and will return an error. We need to add the GROUP BY clause to this query.

=>slides: Pg. 14

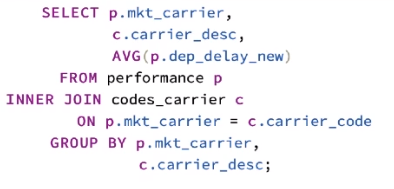
Now, let's use our database of airline on-time performance to explore aggregate functions. Our database of air carrier performance statistics includes a variety of information related to delays and cancellations. For example, the dep\_deplay column indicates how many minutes a flight was delayed prior to its departure.



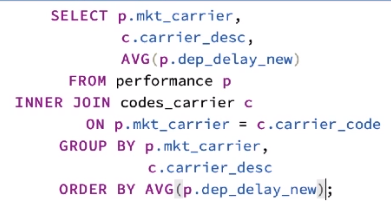
We can use the average aggregate function to calculate the average departure delay for airline flights in our sample. We simply call the aggregate function AVG, for average, on the dep\_delay column FROM the performance table and execute the query. The average departure delay for all flights in our January data is approximately 13.28 minutes.



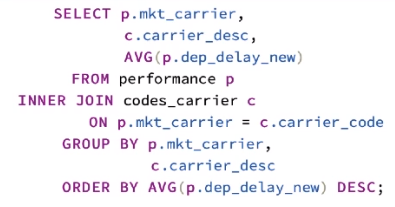
We can use GROUP BY with the aggregate function to calculate the average departure delay grouped by airline. This can help answer the question which airline had the best and which airline had the worst on-time departure performance. When we run this query, we see the average departure delay calculated by carrier.



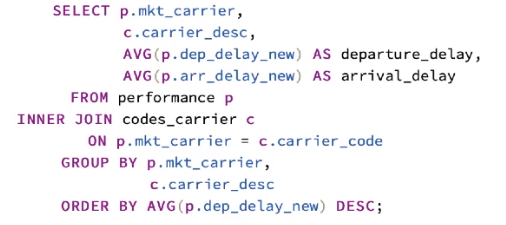
Let's use an inner join, which you learned in the previous module, to make these data easier to interpret. Simply add a table reference to the data from the performance table, p, and add an INNER JOIN to the codes\_carrier table on carrier\_code. We are interested in the carrier\_desc field from this table, so we add it to our SELECT clause. Note that because we now have an additional nonaggregate field, it must also appear in the GROUP BY statement. We now group our aggregate calculations by both carrier code and carrier description. Running this query, we now see the name of the airline in addition to the carrier code, along with the average departure delay.



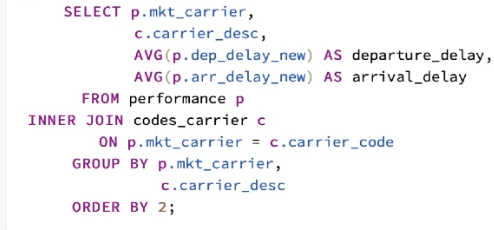
If we are interested in which airline has the shortest average departure delay, we can use an ORDER BY statement on the aggregate field. Remember that by default ORDER BY is in ascending order. We see that Hawaiian Airlines has the shortest average departure delay at just over 5 minutes.



If we are interested instead in which airline has the lengthiest delay, we can specify ORDER BY descending, DESC. This query shows that JetBlue Airways had the lengthiest average departure delay at approximately 24 minutes.



You can also use multiple aggregates in one SELECT statement. For example, if we are interested in both departure delays and arrival delays, we can simply add an additional average aggregate function on the new column, arrival\_delay. We will group by the same fields, so all we need to do is execute this query again. We now see both the average departure and arrival delays for each airline. In this case, we may want to sort the carrier name alphabetically.



We can either specify ORDER BY carrier\_desc or use the SQL shortcut and specify ORDER BY 2, the second column.

# Filtering Aggregate Results

=>slides: Pg. 15

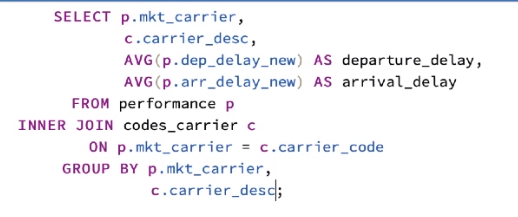
The HAVING keyword is similar to the WHERE keyword. WHERE is designed to filter single rows. HAVING, on the other hand, is designed to filter groups or aggregates.

=>slides: Pg. 16

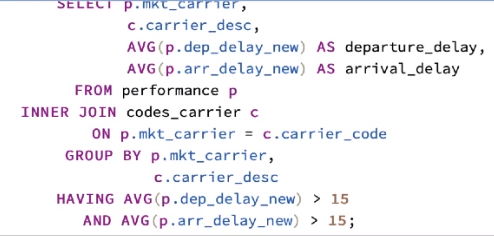
Returning to our table of students, grade levels, and ages, perhaps we are interested in only those grade levels where the average age is less than 19. We must use the HAVING clause to specify our criteria because we are applying a filter to values calculated using an aggregate function. HAVING always follows GROUP BY and can be interpreted as GROUP BY grade\_lvl, then return those groups HAVING an AVG(age) of less than 19. Running this query, we now see the result set we expect. Our freshman and juniors have an average age of less than 19.

=>slides: Pg. 17

Let's look at the use of HAVING by using our airline on-time performance data.



Remember from our earlier demonstration this query, which returned the average departure delay and the average arrival delay for each air carrier.



We can use the HAVING clause to narrow these results. For example, maybe we are interested in only the airlines having both a departure delay and an arrival delay in excess of 15 minutes. To do this, simply use the HAVING clause below the GROUP BY statement. We specify which aggregate fields we are interested in, and what limiting criteria we want to impose on our result set. In this case, we are looking for those airlines with a departure delay greater than 15 minutes and an arrival delay greater than 15 minutes. When we run this query, we will see that both filter criteria are applied to the aggregate calculations. The result set contains the two airlines that had both an average departure delay and an arrival delay in excess of 15 minutes.

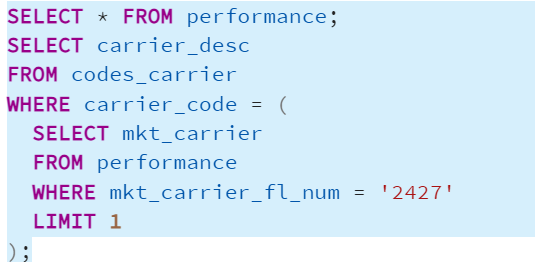
# Understanding Subqueries in PostgreSQL

=>slides: Pg. 18

**Single-row Subquery**

A subquery that returns a single row of data.

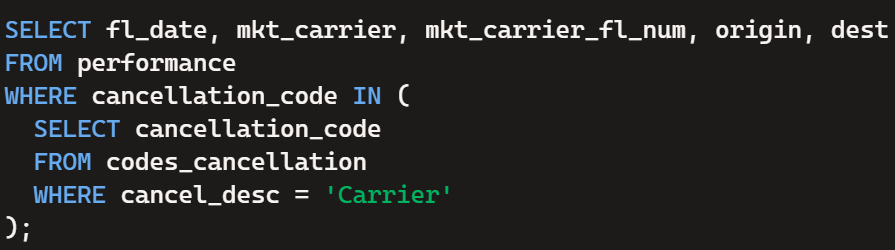
**Example**: Find the carrier description for a specific flight number.



**Multiple-row Subquery**

A subquery that returns multiple rows of data.

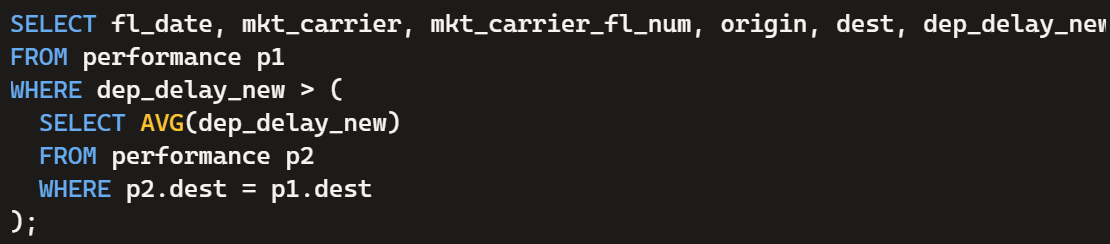
**Example**: Find all flights that have been cancelled due to carrier issues.



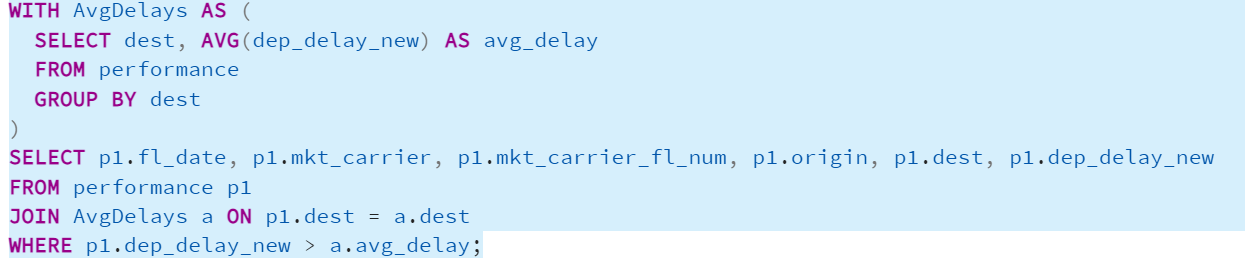
**Correlated Subquery**

A subquery that refers to columns from the outer query and is executed once for each row processed by the outer query.

**Example**: Find flights that have a departure delay greater than the average delay of their destination.



To improve the performance of this correlated subquery, you can use a Common Table Expression (CTE) to precompute the average delays for each destination. This can help reduce the number of times the subquery is executed. Here’s how you can rewrite your query using a CTE:



This query first calculates the average departure delays for each destination and stores the results in the AvgDelays CTE. Then, it joins the performance table with the AvgDelays CTE to filter the flights with departure delays greater than the average delay for their respective destinations.

# Database Views in PostgreSQL

=>slides: Pg. 19

**a. Introduction to Views**

* **Definition**: A view in PostgreSQL is a virtual table that provides a way to present data from one or more tables in a customized manner.
* **Purpose**: Views are used to simplify complex queries, enhance security, and provide a layer of abstraction.
* **Creation**: Views are created using the CREATE VIEW statement.

**Scenarios for Using Views**

1. **Simplifying Complex Queries**:
   * **Scenario**: You frequently need to join multiple tables to retrieve flight information along with the carrier description.

### Execute the View

To see the data in the flight\_info view, use a SELECT statement:

SELECT \* FROM flight\_info;

You can also add WHERE clauses or other SQL conditions to filter the data further:

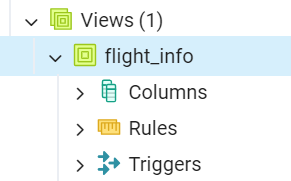
SELECT \* FROM flight\_info

WHERE dep\_delay\_new > 30;

### Dropping the View

If you need to remove the view, you can use the DROP VIEW statement:

### View the Definition of the View



**Enhancing Security**:

* **Scenario**: You want to restrict access to certain columns, like delays and cancellation codes.

**Providing a Layer of Abstraction**:

* **Scenario**: Abstract the complexity of multiple tables and provide a simplified interface for querying flight delays.

**Aggregated Data**:

* **Scenario**: Present aggregated delay data for each carrier.

**Join Operations**:

* **Scenario**: You need to join the performance table with cancellation codes.

# Writing High-Performing SQL Queries

=>slides: Pg. 20

**Tips and Techniques**

**Use Indexes Wisely**

* **Purpose**: Speed up data retrieval.
* **Example**: Create an index on the fl\_date column in the performance table.

**\*\*Avoid SELECT \*\*\***

* **Purpose**: Reduce I/O overhead.
* **Example**: Select only the necessary columns from the performance table.

**Use JOINS Efficiently**

* **Purpose**: Optimize query performance.
* **Example**: Use an INNER JOIN to get flight information along with carrier descriptions.

**Limit the Number of Rows Returned**

* **Purpose**: Improve performance by reducing the amount of data processed.
* **Example**: Retrieve the first 10 rows of performance data.

**Use Appropriate Data Types**

* **Purpose**: Ensure efficient storage and quick access.
* **Example**: Verify and use appropriate data types in the performance table (e.g., character varying instead of text for fixed-length strings).

**Write Simple and Understandable Queries**

* **Purpose**: Easier to maintain and optimize.
* **Example**: Use CTE to simplify a complex query.

=>slides: Pg. 21

**Use EXISTS Instead of IN for Subqueries**

* **Purpose**: Improve performance with large datasets.
* **Example**: Find all flights that have a corresponding carrier in the codes\_carrier table.

**Optimize Join Order in Complex Queries**

* **Purpose**: Reduce the amount of data processed in joins.
* **Example**: Join the performance table with the codes\_carrier and codes\_cancellation tables.

**Use Query Analyzer Tools**

* **Purpose**: Identify and optimize slow queries.
* **Example**: Use EXPLAIN to analyze a query.

**Avoid Using Functions on Indexed Columns in WHERE Clause**

* **Purpose**: Maintain index efficiency.
* **Example**: Avoid using functions that negate the use of indexes.

**How Performance is Improved**

1. **Filtering Early**:
   * **Benefit**: By applying the WHERE clause early (e.g., WHERE p.cancelled = 1), you reduce the number of rows that need to be joined with other tables. This filtering ensures that only relevant data is processed in subsequent joins.
   * **Implementation**: Ensure that the filter condition is applied before performing the joins.

SELECT p.fl\_date, p.mkt\_carrier, c.carrier\_desc, cc.cancel\_desc, p.origin, p.dest

FROM performance p

JOIN codes\_carrier c ON p.mkt\_carrier = c.carrier\_code

JOIN codes\_cancellation cc ON p.cancellation\_code = cc.cancellation\_code

WHERE p.cancelled = 1;

1. **Joining Smaller Tables First**:
   * **Benefit**: By joining smaller tables first, you can reduce the number of intermediate rows that need to be processed, which can decrease the overall computational load.
   * **Example**: If codes\_carrier and codes\_cancellation are smaller compared to performance, joining them first and filtering the result can be more efficient.
2. **Index Usage**:
   * **Benefit**: Ensuring that indexes are used on the join columns (mkt\_carrier and cancellation\_code) can speed up the join operation by quickly locating matching rows.
   * **Implementation**: Create indexes on the join columns.

CREATE INDEX idx\_performance\_mkt\_carrier ON performance (mkt\_carrier);

CREATE INDEX idx\_performance\_cancellation\_code ON performance (cancellation\_code);

1. **Reducing the Dataset Size**:
   * **Benefit**: By performing joins in an optimized order, the size of intermediate datasets can be minimized, reducing memory usage and improving query execution time.
   * **Example**: Joining codes\_carrier and codes\_cancellation to filter out only the necessary rows before performing the final join with the larger performance table.

### Optimized Query Example

Here’s how an optimized query might look:

**Explanation**:

* **CTE (Common Table Expression)**: First, filter the performance table to include only cancelled flights.
* **Join Order**: Then, join the smaller codes\_carrier and codes\_cancellation tables to the filtered dataset.

### Unoptimized Query Example

In this unoptimized query, the joins are performed first before filtering the performance table based on the cancelled column.

This means all rows from the performance table are joined with the codes\_carrier and codes\_cancellation tables, which can lead to processing a large number of rows before the final filtering is applied.

# Query Plans Overview

=>slides: Pg. 22

A query plan is a sequence of steps that the database management system uses to execute a SQL query. It outlines how tables will be scanned, joined, and processed to return the desired result.

# Identifying Performance Issues in SQL Queries

=>slides: Pg. 25

EXPLAIN SELECT \* FROM performance WHERE dep\_delay\_new > 30;

# Common Performance Issues

=>slides: Pg. 26

# Summary

=>slides: Pg. 27

In summary, ORDER BY allows us to present our results sorted in a way that may be useful to you as an analyst. Aggregate functions, such as SUM or AVG, perform calculations on entire datasets or on groups of data using the GROUP BY keyword. If you perform analyses on groups of data, you can filter the aggregate results by using the HAVING keyword. With these functions in your toolbelt, you're well on your way to using SQL to perform more complex analysis.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*