4. Limiting Your Results

Contents

[1. Overview 1](#_Toc46)

[2. Introducing The WHERE Keyword 1](#_Toc28538)

[3. Specific Conditions 1](#_Toc28892)

[4. I LIKE Pattern Matching 2](#_Toc26581)

[5. Null Values 2](#_Toc5820)

[6. Putting It Together 2](#_Toc32514)

[7. Operator Precedence 2](#_Toc5613)

[8. Summary 2](#_Toc5608)

# 1. Overview

=>slides: Pg. 1

So far in this course, we have queried the database to return either all or specific columns of interest. Although there are use cases where this may be our end goal, more often than not we want to only return certain records based on criteria that we specify.

=>slides: Pg. 2

In this module, we will learn how to do exactly that using the WHERE clause. I'll introduce you to relational operators to specify the data that we would like to see. We can even use more complicated patterns with the LIKE keyword. I'll also share with you null values, which are an important special character value in SQL.

# Introducing The WHERE Keyword

=>slides: Pg. 3

Almost everything that we do in SQL is just a modification of the basic SELECT query that you've already learned. For example, this code retrieves two fields, first\_name and last\_name, from a given table, person. To specify criteria to limit the records returned by our query, we can use the WHERE keyword. The WHERE clause is made up of the WHERE keyword and any limiting criteria. The simplest criteria is when a field is equal to a particular value. Let's look at the framework.

=>slides: Pg. 4

We begin with SELECT to specify our columns of interest. Next, we use FROM to specify where these columns are stored in what table. And finally, we add our WHERE clause. Our WHERE clause allows us to give criteria that we are interested in that the system will use to filter our matching rows. And remember, we always want to end with a semicolon.

=>slides: Pg. 5

For example, we may only want to see the first name and last name from the person table when a person has the first name Shelby. To put this into SQL, we just add this criteria to the WHERE clause. =>slides: Pg. 6

We now have a WHERE clause that details the specifics of our request. If we execute this query on the sample person table, we would get the following result set.

=>slides: Pg. 7

It is important to note that our WHERE criteria is case sensitive. This can become an issue when working with databases that store information in all uppercase or all lowercase, or in databases that have suffered from inconsistent data entry by multiple people.

# Specific Conditions

=>slides: Pg. 8

In addition to querying for records that have a particular field that equals a specific value, we can use a variety of other comparison operators. These operators are grounded in logic and relational algebra. But don't worry, they're simple to remember. We have equal, which you already have been introduced to, and not equal to, less than or great than, and greater than or equal to, or less than and equal to. We call these comparison operators because they tell PostgreSQL to compare the specified field against a specified value. Notice that equals and not equals can be applied to all data types. The remaining operators are generally used on numeric, integer, and date values. Multiple criteria can be combined using the keyword AND.

=>slides: Pg. 9

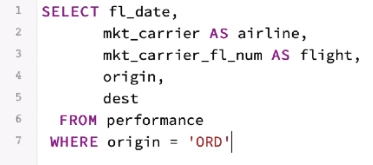
For example, assume we have a table of city populations. If we want to see the population of Louisville, Kentucky, we might type SELECT city, state, population FROM city\_population WHERE city = Louisville. Let's look at the results. What happened? Several states have cities with the same name.

=>slides: Pg. 10

To return the correct field, we would want to establish specific criteria for both the city field and the state field. For example, SELECT city, state, population FROM city\_population WHERE city = Louisville AND state = KY. Now we see the information that we were searching for.

=>slides: Pg. 11

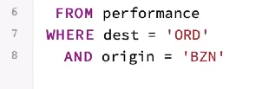
Let's use our database of on-time performance data for airlines to apply some of these comparison operators to filter for specific conditions. In our sample database of air carrier performance statistics, we have a variety of columns to choose from.



Let's say that we live in Chicago and are interested primarily in flights that originated in Chicago. Perhaps we are interested only in the date of the flight, the airline, the flight number, the origin, and the destination. Notice that we can use an alias to make certain columns easier to interpret. For example, mkt\_carrier is relabeled airline and the mkt\_carrier\_fl\_num column is relabeled simply flight. This query should look familiar. The SELECT keyword specifies the columns of interest, the FROM keyword tells Postgres that we want to retrieve those fields from the table named performance. Now we'll use the WHERE clause to limit our results to only those flights that originated in Chicago. The airport code for Chicago O'Hare is ORD; therefore, we want only those flights WHERE origin = ORD. When we run this query, we'll see that we now only get information for flights that originated from Chicago O'Hare. What if conversely we wanted to know only those flights that had Chicago O'Hare as a final destination?



We can simply change origin in our WHERE clause to destination. If we run our query again, we'll see that we get only those flights where Chicago O'Hare was the final destination. Remember that we can also use the AND clause to add additional criteria.



For example, maybe we want flights where Chicago O'Hare was the destination, but we're interested only in flights that originated in Bozeman, Montana. The airport code for Bozeman, Montana is BZN. That is, we want flights WHERE the dest is Chicago O'Hare and WHERE origin = BZN. If we run this updated query, notice that we now see only those flights that originated in Bozeman, Montana with the destination of Chicago O'Hare.

# I LIKE Pattern Matching

=>slides: Pg. 12

Relational operators require a specific comparison value; however, the LIKE keyword is a logical operator that allows us to find records where a field matches a specific pattern. The general format of the LIKE statement is WHERE, field name, LIKE, pattern.

=>slides: Pg. 13

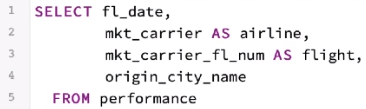
For example, remember our sample person table. This query used the equality relational operator to find students who had the first name Shelby. We could also write this using the LIKE keyword; WHERE first\_name LIKE Shelby is the same as WHERE first\_name = Shelby.

=>slides: Pg. 14

This isn't a very interesting use of the LIKE keyword. Importantly, the pattern can include not only regular characters, but also wildcard characters. Introducing wildcards allows us a great deal of flexibility when choosing our criteria. There are two wildcards that can be used with the LIKE keyword; the percent sign and the underscore. The percent sign represents 0 or more characters, while the underscore represents exactly 1 character.

=>slides: Pg. 15

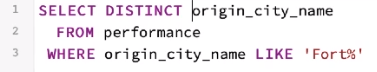
Now let's return to the airline on-time performance database to see how to implement the LIKE keyword and how we can use these wildcards in pattern matching.



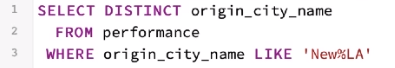
In our airline performance database, the origin\_city\_name column indicates the full name and state of the city where the flight originated. Let's say that we want information on the flight date, airline, flight number, and origin city from our performance table. This query should look familiar. If you run this, you'll see that we get this information for all flights. Perhaps we are conducting an analysis and we are interested only in those cities that start with the word Fort, such as Fort Myers, Florida in the fourth row of this result set.



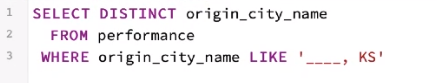
We can use the LIKE keyword to look for that pattern. In this case, we would type WHERE origin\_city\_name LIKE Fort, followed by the percentage wildcard. The percentage wildcard represents any number of wildcard characters. This criteria tells Postgres that we are interested in any origin city name that starts with Fort, regardless of the additional number of characters that follow. If we run this query, notice that the query returns information for all flights where the origin\_city\_name starts with Fort. Perhaps instead of these individual flights, we are interested only in a list of unique origin cities.



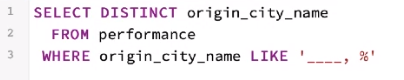
We can revise our query to eliminate flight date, airline, and flight, and tell Postgres that we are interested only in distinct city names. This query returns the four cities in our sample that begin with Fort. Wildcards can work anywhere in the LIKE criteria. For example, maybe we want only those distinct origin cities that are in Florida. We can use the percentage wildcard at the beginning of our criteria to indicate that we are interested only in results that end in FL, regardless of the number of characters that precede it. PostgreSQL will filter our data and return only those values that are in Florida.



In fact, we can even use the percentage sign in the middle of a pattern. For example, if we were to change our criteria to New%LA, the state code for Louisiana, we are asking Postgres to return only those cities that start with New, have any number of characters in the middle, and end with LA. If we run this query, we get only one result, New Orleans, Louisiana. The result begins with New and ends with LA. The wildcard in the middle captured the remaining characters. Remember that while the percent sign represents an unlimited number of characters, each underscore represents exactly one character.



For example, what if we want only those cities in Kansas that have four letters in their name? We can specify that we want exactly four letters in our pattern by 4 underscores, \_\_\_\_, KS, the state code for Kansas. If we execute this query, we get one result, Hays, KS. We can combine percentage wildcards and underscore wildcards to further refine our LIKE statement. For example, maybe we are interested in all four-letter origin cities in our dataset.



In this column, city and state are separated by a comma. We can use our four underscores, which represent the pattern of four specific characters, followed by a comma, which means that the data must have a comma following the four characters, and then followed by a percent sign, which says that whatever values come after the comma are acceptable in the result set. If we run this query, we'll see a total of 12 origin cities in our dataset that have a four-letter name.

=>slides: Pg. 16

What if instead of wanting to match patterns, we wanted to find those records that did not match the pattern? PostgreSQL allows us to do just that by using the NOT keyword. The NOT keyword tells SQL we want to find those fields that do not match the pattern specified by LIKE.

=>slides: Pg. 17

Now we have seen relatively tidy examples in this module, remember that pattern matching can be particularly useful when we are dealing with messy datasets. The type of pattern matching performed by the LIKE keyword is known as fuzzy matching. It allows us to deal with those data that are less than perfect. PostgreSQL also allows the use of regular expressions for more complicated patterns to match various character combinations. Regular expressions can be used to validate entries, find data entry errors, or search for useful patterns in our data. This is a powerful advanced feature that is outside the scope of this course. As you develop more advanced SQL, however, just be aware that this capability does exist.

# Null Values

=>slides: Pg. 18

PostgreSQL includes two criteria statements specifically for dealing with null values, but what exactly is a null value? Null is a special character in SQL and relational databases. When you work with certain programming languages, we often consider null to be equivalent to 0. However, this is not true in SQL. In database language, null is not a specific value like 0 or a blank space. Rather, null is an indicator of sorts. It indicates when a field has a missing or unknown value. A field is null only when no data of any kind has been entered into that field. Think about a table of people and their ages.

=>slides: Pg. 19

Let's say that in these data Brenna did not tell us her age. It would not be appropriate to make her age 0. Not only is that clearly false, but it could skew our calculation of averages or other analysis. Age is a numeric variable, so we also couldn't use the blank space character to substitute in the data. Note that regardless of variable type, this would be a very poor practice. To a database, even a blank space character is a value of some kind. The best thing to do from a data perspective in this situation is to let the database know there is no information in this field. Therefore, we simply do not enter anything at all, and SQL recognizes the contents of this field for this record as a null value.

=>slides: Pg. 20

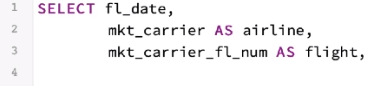
The two keywords in PostgreSQL specifically for dealing with null values are IS NULL and IS NOT NULL. These keywords are unique among SQL keywords because they can be applied only to determine the existence of the special null value. They have no other use. However, we can think of these keywords as relational operators that we've seen before. IS NULL is the same as saying Field = nothing. IS NOT NULL is the same as saying Field not equal to nothing.

=>slides: Pg. 21

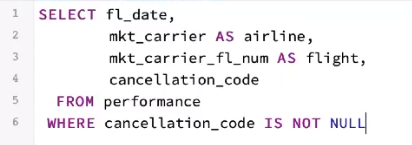
Note that you could not use standard relational operators to make these comparisons. Because of the special nature of null values, you must use the IS NULL or IS NOT NULL syntax.

=>slides: Pg. 22

Now, let's revisit our database of on-time airline performance to find null values and to find values that are not null.



Our performance dataset includes a column named cancellation\_code that indicates a reason code for every cancelled flight. If a flight was not cancelled, there will be not be a code in this column. As usual, we have our query to return flight date, airline, and flight number.



Let's add the cancellation\_code to our list of columns from the performance table. Now we'll add our WHERE criteria. Let's say that we are interested only in flights that have been cancelled. That is to say, we want those records that have a value in the cancellation\_code field. Another way of thinking of this is that we want those records WHERE the cancellation\_ code IS NOT NULL. When we run this query, PostgreSQL returns a result set of 18, 976 rows of flights that were cancelled for various reasons.



We can also do the inverse and ask Postgres to return only those flights where the cancellation\_code IS NULL. In this result set, we can see that over 602, 000 flights were successfully operated in January. These flights return a null value in the cancellation\_code column. IS NULL and IS NOT NULL have many uses, but these keywords can be particularly useful when we are diagnosing problems when analyzing a dataset.

=>slides: Pg. 23

For example, arithmetic operations involving a null value will always return null. Null values may have unintended consequences on our analysis or our results, so it can be a good idea to check for these values.

# Putting It Together

=>slides: Pg. 24

As we've already seen, we can combine multiple criteria statements using the AND keyword. In fact, there is no limit to the number of criteria that we can combine. The AND keyword is a logical operator that simply means that all of the conditions must be true. If a row from the table matches both of the conditions we specify, it will be included. Another logical operator that we can use to combine criteria is the OR keyword. Unlike AND where all conditions must be true, the OR keyword says that either this condition OR that condition must be true. If a row from the table matches either condition we specify, it will be included.

=>slides: Pg. 25

Depending on our criteria, there are two special keywords that can make using logical operators even easier. The first of these is BETWEEN. This keyword does just what we think it might do, it returns those records where a field is between two given criteria. More specifically, it will return records where a field is greater than or equal to a specific value, and less than or equal to a specific value; it is inclusive. You can use BETWEEN instead of writing a statement using two separate relational criteria. For example, this query could be used to retrieve those individuals between the ages of 19 and 35. We can use the WHERE statement, age greater than or equal to 19 AND age less than or equal to 35. We could also use the BETWEEN keyword to say that we want to see those records WHERE age is BETWEEN 19 and 35.

=>slides: Pg. 26

When you use BETWEEN, remember that your results will be inclusive. In this example, our query will return Brenda, age 25, Shelby, age 26, and Tom, age 35. Your lower and upper boundaries are inclusive.

=>slides: Pg. 27

Instead of using multiple OR statements, we can use the SQL keyword IN. This keyword lets us provide a list of options that we are looking for in a given field.

=>slides: Pg. 28

Let's say that we are looking for specific students based on their first name. Instead of typing SELECT first\_name, age FROM person WHERE first\_name = Jimmy OR first\_name = Brenna OR first\_name = Elmo, because each criteria is looking at the same field, we could simply type SELECT first\_name, age FROM person WHERE first\_name IN Jimmy, Brenna, Elmo. Each matching value that we are interested in appears in a list following the variable name, separated by commas, and wrapped in parentheses. The query returns records where first name matches any of these listed values. Notice that IN replaces multiple criteria statements where the field is equal to certain values. You cannot use IN to look for multiple patterns using LIKE. You would need to add multiple LIKE statements and combine them using the OR keyword.

=>slides: Pg. 29  
IN has a companion keyword, NOT IN, that can be used to return records when the value does not match any of the values in a list.

# Operator Precedence

=>slides: Pg. 30

When we use logical operators to combine multiple criteria, we need to be aware of a SQL concept called operator precedence. Operator precedence determines the sequence in which operations are performed in our query.

=>slides: Pg. 31

In PostgreSQL, by default, AND has a higher operator precedence than OR. What this means is that an AND statement will be evaluated before an OR statement, regardless of the order we list them in in the WHERE clause. For example, let's look at the table of names that we used earlier, but let's add a variable, hometown. If we want to see students named Shelby or Tom that are from Boston, we might write this query. Take a look at our results. Are these the results you would expect? We see Shelby and Tom from Boston, but the query also returned a student named Shelby whose hometown was Denver. This is because of SQL operator precedence. The AND expression is evaluated first, so the query looks for students who meet both criteria, they are named Tom, and they are from Boston. Only when this query criteria is satisfied do we look at the OR clause because OR has a lower operator precedence than AND. Once the query finds those students named Tom and from Boston, then it will look at OR to find students named Shelby. The concept of operator precedence can be a bit confusing, but luckily it is easy to avoid by the judicious use of parentheses. Parentheses not only make your code easier to read, but also help SQL know how to evaluate multiple criteria.

=>slides: Pg. 32

Let's take another look at this query. I would say that we expected to see only two records in our result set, one for the student Tom from Boston, and one for the student Shelby from Boston. We did not expect to see Shelby from Denver. Simply adding a set of parentheses will give us the result that we expect. The parentheses tell SQL that we want to evaluate the two OR statements before we evaluate the hometown statement. Now we can see that the query first asks for records with the first name of Tom or Shelby, and then applies our second criteria. This makes sense if we think about what we want to see in plain English. We want records where the first name is Tom or Shelby, and then we only want to see those students if their hometown is Boston.

=>slides: Pg. 33  
The main takeaway here is that by using parentheses when writing more complex expressions, we can avoid any confusion that may arise from the default order of precedence. This is a best practice and can save us a lot of trial and error with our code when unexpected results occur.

# Summary

=>slides: Pg. 34

In summary, using the WHERE clause, we can add a great deal of capability to our SQL statements, we can use comparison operators to retrieve records that match our prescribed criteria, and we can use LIKE keywords to match specific patterns. The AND and OR logical operators allow us to use multiple criteria. When combining multiple logical operators, we should always use parentheses to make our desired order of operations clear. Now we have the knowledge to select only the records and columns that we are interested in from small and large tables alike.

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