3. SELECTing Your Data

Contents

[1. Overview 1](#_Toc30058)

[2. Introduction to SELECT 1](#_Toc8985)

[3. Explicit Selection 2](#_Toc29612)

[4. Selecting Distinct Values 3](#_Toc32642)

[5. Summary 4](#_Toc1023)

# 1. Overview

=>slides: Pg. 1-2

In this module, we are going to learn about one of the most basic building blocks of SQL, the SELECT statement. Through a series of examples, we will learn how to query the data to select fields of interest from a database. We'll also discuss how to limit our results to unique values using the DISTINCT keyword. Throughout this module, I'll also share some important formatting considerations.

=>slides: Pg. 3

When we want to retrieve information from a database, we query the database for that information. For example, we may want to know what flights departed from Atlanta, or what flights had departure delays. To do this, we can write a SQL statement that will query the database. In fact, these statements are commonly referred to as queries. This query can help us to solve the questions we ask of our data. =>slides: Pg. 4

Although SQL is a defined programming language with clearly defined statements, it does not have formatting requirements. When we write queries, it is helpful to use consistent formatting. Doing so will help you easily reach your code as you write increasingly complicated queries, and will help others to more easily interpret your code. Readable code should always be an end goal when writing your queries. We'll discuss formatting as we move through the course. For the moment, note that as a best practice, SQL keywords, or commands, are generally capitalized while other words are in lowercase. This can be helpful in understanding what parts of your code refer to system-specific language and what parts of your code reference database tables and fields.

# Introduction to SELECT

=>slides: Pg. 5

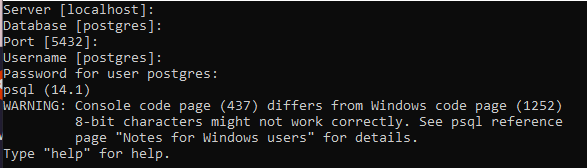
Words that tell SQL to do something, or programming commands, are keywords.

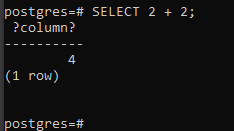
Technically they are called reserved keywords because they are reserved for use by Postgres itself, but in practice they're generally just referred to as keywords.

=>slides: Pg. 6

The most fundamental keyword in SQL is the SELECT command. As its name implies, SELECT is the command that allows us to retrieve selected data from the database. We can specify what columns we want returned from what tables, or what records we want returned based on the value of certain columns.

=>slides: Pg. 7





In fact, we don't even need to select from a table to see it in action. For example, SELECT 2+2 will perform the requested calculation and return the answer, 4, right in the SQL window.

=>slides: Pg. 8

As you can see, the SELECT statement can be very simple. Most of the time, however, we'll want to use the SELECT command to query the database. Take a look at this framework. This framework allows us to provide information to the database about the data that we would like to see. After the SELECT keyword, we can list the names of columns that we are interested in, while the FROM clause tells us in what table these columns are stored. Table names and field names are called identifiers. Notice that this framework ends with a semicolon. Although PostgreSQL will often allow you to execute a statement that does not end in a semicolon, it is best practice to end all SQL statements with a semicolon. When you begin to write more complicated queries and programs, multiple statements in one query window must be separated by semicolons, so it's good practice to do that now.

=>slides: Pg. 9

If we want to see all columns from a given table, we can use an asterisk as a wildcard. Typing SELECT \* is the same as saying SELECT all columns. Therefore, if we type SELECT \* FROM tablename, we would return all of the columns from that table.

=>slides: Pg. 10

Let's look at the asterisk wildcard in action using our database of flight on-time performance. Notice that our database currently contains one table called performance. We want to see all of the columns and data that presently exist in that table. Since we know we want to see everything, it makes sense to apply the asterisk wildcard.



Typing the command SELECT \* FROM PERFORMANCE is the SQL equivalent of saying show me all the fields from the performance table. After typing this and instructing pgAdmin to execute the command, we can see that the database returns all the contents of the performance table. There are 19 columns of data, and if we were to scroll, we could see that the entire 621, 461 rows of data, or records, were returned. There are times that it can be useful to ask the system to retrieve all of the data. For example, if you were writing a complicated SQL query, you may need to investigate the data more thoroughly. There are also instances when you may have a very small table and it is easier to review all of the data at once.

=>slides: Pg. 11

Generally speaking, however, using the asterisk in production code is considered bad practice. When databases contain a large amount of information, queries using the wildcard can run very slowly.

=>slides: Pg. 12

In addition, your code is more readable and useable to others if you list the specific columns that you are interested in. This is known as explicit notation because you are being explicit about what columns you want the query to return. Explicit notation will also make your code easier to troubleshoot.

# Explicit Selection

=>slides: Pg. 13

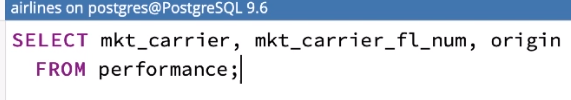
To tell the database what columns or fields we are interested in, we can simply list them after the SELECT keyword. Each column name should be separated by a comma. After listing these column names, we can then use the FROM keyword to tell the database what table we want to query to retrieve these fields. Later in this course we will look at how to combine SELECT and FROM to query data from multiple tables, which is a technique called joining data. For now, however, let's focus on retrieving data from a single table.

=>slides: Pg. 14

This SQL code would return first name and last name for every record in a person table. Notice that the SELECT keyword specifies the fields we are interested in, in this case first\_name and last\_name, while the FROM keyword specifies the table the database should look at to return these fields.

=>slides: Pg. 15

Let's return to our database of flight on-time performance to see the SELECT statement in action. Our database currently has one table, performance, that contains 19 fields. Let's say that we are only interested in the airline, the flight number, and the departure city code. Notice that using the asterisk, wildcard, and the SELECT clause would return all fields. To limit our results to the three fields we are interested in, we should use explicit notation.



As we're thinking of the field names, remember that you can use the graphical pgAdmin interface to see what fields are in the table. After clicking Execute, we can examine the results. Notice that now PostgreSQL only returns three columns, the three columns that are specified in our SELECT clause. Our query returns these three data points for all records in our performance table. Notice that if we make a typo or ask the system to return a field that does not exist in the table, the system will return an error message.



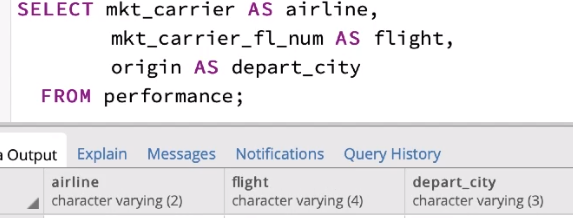
For example, if we incorrectly list the flight number field, we would receive this error.



If you receive this message, simply check the fields you've specified and make sure they're listed correctly, and that they do indeed exist in the table specified in the FROM clause. PostgreSQL will often attempt to suggest the correct field name, which may be useful in diagnosing your error.



Let's look again at our query for airline, flight number, and departure city code. I am sure you would agree that some of our column names are not very user friendly. Relabeling these columns might be useful to us when we want to share our results or use our results in more complex queries. Fortunately, PostgreSQL allows us to easily assign an alias to our column names.



Simply type AS, and the new field name after the original column name. For example, we can adjust this query to easily alias our columns with friendlier names. Notice that mkt\_carrier becomes airline, while mkt\_carrier\_fl\_num becomes simply flight. We can see that our results are the same, however the column name has been changed in our result set. In this query, I have used spacing to align all SQL keywords on the same character boundary, followed by the identifiers and other details. Essentially, the keywords are right-aligned while the identifiers are left-aligned. This creates what is called a river in typography down the middle of our code. This formatting allows the reader to easily scan over the code and separate the fundamental SQL from the specific implementation detail. Remember, there are no hard and fast rules on formatting. Your primary concern should be consistency and readability.

# Selecting Distinct Values

=>slides: Pg. 16

The minimum number of fields you can specify is one, and there is no maximum. You could explicitly notate all fields that exist in a given table. Sometimes, however, it can be useful to look at the distinct values in a database. Distinct values are unique values.

=>slides: Pg. 17

For example, think about a classroom. Many people have common names. Perhaps a classroom has three students named Katie, two students named Amy, and one student each named Jason, Alec, and Shannon. As you can see, the classroom has a total of eight students. Assume these names were stored in a database table called students. If we wanted to see the first name of all students, we might write the following SQL. This SQL tells Postgres that we would like it to return all values for the student's first name that are stored on the students table. The query would correctly return the following result set with eight rows. What if instead we wanted to know the distinct first names in the classroom? Distinct means that regardless of if the value occurs more than once, we only want to return the given value one time. That is, we want to see the unique values that exist in the column. To tell Postgres that we want it to return distinct values, we use the DISTINCT keyword. The DISTINCT keyword is placed immediately after the SELECT keyword, and before we specify our fields. If we run this query, notice that we now have fewer results, only five rows. Although we have three students named Katie and two students named Amy, this query only returned those values once; therefore, this result set is a list of unique, or distinct, values.

=>slides: Pg. 18

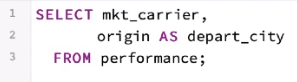
Now, let's apply the DISTINCT keyword in our on-time flight performance database.



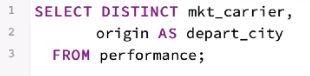
Let's say that we are interested in what airlines are listed in our table as marketing carriers. We know that this information is stored in the mkt\_carrier column. Running the query, SELECT mkt\_carrier FROM performance returns the airline for each record in our database. In fact, it is not until the 2196th record that we see a different airline code.



If, however, we change this query to SELECT DISTINCT mkt\_carrier FROM performance, notice that instead of returning over 600, 000 records, like our first query, this query returns only 11 records. The difference is that this query returned only unique values. In other words, there are 11 airlines listed as unique marketing carriers in the performance table. DISTINCT can also be used to find unique combinations.



For example, this query will return the city that each airline departs from. Notice that this query again returns a row for every record in our table. But let's say that we actually only want to see a list of cities that the airline flies from.



That is to say we want a list of unique departure cities for each airline. All we need to do is slightly adjust this query by adding the DISTINCT keyword. Now we have some useful information. We have a list of the cities served by each airline, at least for outbound or departing flights. There are 1161 rows in this result set. The moral of the story is the DISTINCT can be used to return distinct values when a single column is listed in the SELECT clause, or to return distinct combinations when more than one column is listed.

# Summary

=>slides: Pg. 19

To summarize, we used the SELECT keyword to tell PostgreSQL to return fields from the database. The FROM keyword generally accompanies SELECT and tells Postgres what table we want to use. Finally, while by default SELECT returns all values, we can employ the DISTINCT keyword to limit results to unique values, or to combinations of the specified fields.

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