Intro to databases tutorial

In this tutorial, you'll learn how to:

- Write SQL statements to retrieve data from a PostgreSQL database
- Filter data with `WHERE` clauses

You'll use a database that a local pizza shop might use to run their business. The `PizzaShop` database has tables that represent pizzas, toppings, sales, and customers.

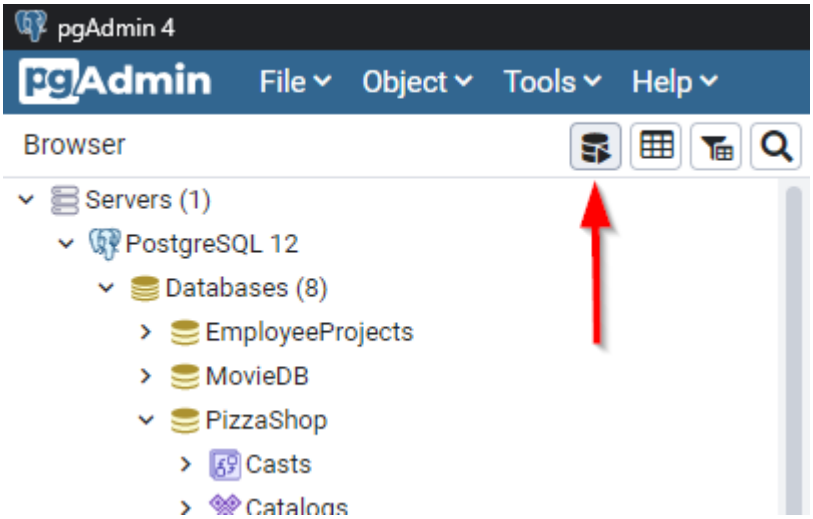
Before you get started with this tutorial, you must setup the `PizzaShop` database if you haven't already. The "Database setup" lesson in the Intro to Tools unit for PostgreSQL shows you how to do this.

You'll write your statements in **pgAdmin**. pgAdmin connects to a PostgreSQL database server so you can run SQL statements and perform other database-related functions. Refer to the unit about PostgreSQL in the Intro to Tools section of this course for details on getting started with PostgreSQL and pgAdmin.

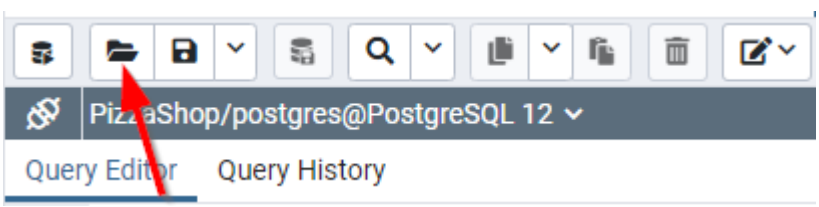
Getting started

In pgAdmin, connect to PostgreSQL and expand the **Databases** node by double-clicking it or clicking the caret on its side. Locate the "PizzaShop" database in the list, and click the caret on its side to expand it. This establishes "PizzaShop" as the database for your statements to run on.

Finally, click the **Query Tool** button to open the **Query Editor** which allows you to enter SQL statements—also known as queries:



All queries in this tutorial are also in the `.sql` files in the `tutorial-final` folder. If you encounter an issue or want to run the pre-typed queries, you can open these files in pgAdmin by using the **Open File** button in the **Query Tool** window:



Part One: Selecting data from tables

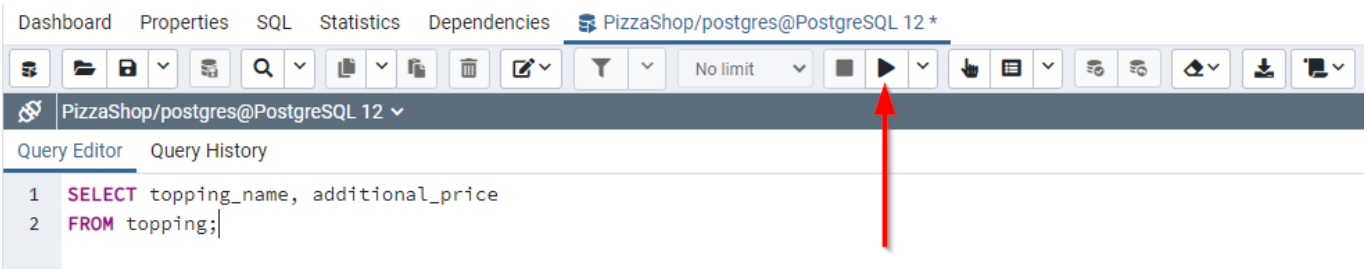
To retrieve data from a table, you use a `SELECT` statement. A `SELECT` statement consists of the names of the columns you want returned and the table they come from.

Topping table

The `topping` table has two columns: `topping_name` and `additional_price`. To get these columns from the `topping` table, write this statement in the **Query Editor** of pgAdmin:

```
SELECT topping_name, additional_price
FROM topping;
```

Run the statement by pressing the **F5** key on your keyboard, or click the "play" triangle button in the toolbar:



In the bottom half of the pgAdmin window under "Data Output", you'll see the data from the `topping` table. The `topping_name` column has values like "Pepperoni" and "Mushrooms"—toppings that you might order on a pizza. The `additional_price` column has numeric values, representing the price that the pizza shop charges to add the topping to a pizza.

Size table

Now, you'll write a statement to retrieve the data from the `size` table.

You can leave the `topping` statement if you want. By default, pgAdmin displays the data of the last statement. If you highlight a statement before running it, pgAdmin returns the data of that statement instead. You can also open a new **Query Editor** by clicking the **Query Tool** button again.

The `size` table represents the different sizes of pizza that you can order. The table has four columns: `size_id`, `size_description`, `diameter`, and `base_price`. To get the data from the `size` table, write and run this statement:

```
SELECT size_id, size_description, diameter, base_price
FROM size;
```

In the bottom half of the pgAdmin window, you'll now see the data from the `size` table. The `size_id` is a single character that represents the size—you'll see the significance of this later. The `size_description` column contains the size spelled out. The `diameter` column is the diameter in inches of each size. The

`base_price` column is like the one you saw in the `topping` table, but this is the price of a pizza before adding toppings.

Part Two: Filtering data using WHERE

Retrieving all the data from a table isn't always what you want, especially if you're looking for rows that match a certain value or condition. That's where the `WHERE` clause comes in.

The `pizza` table represents pizzas that customers have ordered. Write and run this statement to retrieve the data of the `pizza` table:

```
SELECT pizza_id, sale_id, size_id, crust, sauce, price, additional_instructions
FROM pizza;
```

Notice that pizzas have a size, which you see represented by the same `size_id` you saw when you ran the `SELECT` statement on the `size` table. There's a relationship between the `pizza` table and the `size` table. You'll learn more about relationships between tables in a later lesson.

Pizzas also have a crust type, amount of sauce desired, and a price that includes toppings. There's also optional "additional instructions" that a customer can request, such as asking to cut the pizza into squares.

Say you want to return only the small pizzas that customers have ordered. You can use a `WHERE` clause to filter the data down to the ones you're interested in.

Modify the `pizza` statement you wrote to have a `WHERE` clause that gets only small pizzas:

```
SELECT pizza_id, sale_id, size_id, crust, sauce, price, additional_instructions
FROM pizza
WHERE size_id = 'S';
```

Note that the semicolon moved from being right after `FROM pizza` to being after the `'S'`. This is important because PostgreSQL uses semicolons to separate statements. If you leave it after `FROM pizza`, you'll encounter an error.

When you run the modified statement, you'll see that the rows in the **Data Output** section are only for small pizzas.

Multiple conditions

If you also wanted to know which small pizzas were also thin crust, you could combine these conditions with `AND`:

```
SELECT pizza_id, sale_id, size_id, crust, sauce, price, additional_instructions
FROM pizza
WHERE size_id = 'S'
AND crust = 'Thin';
```

Now if you run the statement, you only get the small thin crust pizzas.

You can also use `OR` to say that you want one condition, or the other, to be true. For example, if you change the `AND` to `OR` in the previous statement, you'll get all small pizzas regardless of crust, and non-small pizzas that have thin crust:

```
SELECT pizza_id, sale_id, size_id, crust, sauce, price, additional_instructions
FROM pizza
WHERE size_id = 'S'
OR crust = 'Thin';
```

Not equal

`WHERE` clauses aren't only used for retrieving values that match a certain value. You can also use them to filter out a specific value.

If you wanted to write a statement that returned everything *except* small pizzas, you can write the query like this:

```
SELECT pizza_id, sale_id, size_id, crust, sauce, price, additional_instructions
FROM pizza
WHERE size_id != 'S';
```

When you run this statement, you'll get all medium and large pizzas.

Greater than or less than

You don't have to test for a value being equal to another. You can also test for values that are greater than or less than a certain value.

To retrieve all pizzas that cost more than $15, you can change your `WHERE` clause to this:

```
SELECT pizza_id, sale_id, size_id, crust, sauce, price, additional_instructions
FROM pizza
WHERE price > 15;
```

To get pizzas that cost less than $10, you can change it to this:

```
SELECT pizza_id, sale_id, size_id, crust, sauce, price, additional_instructions
FROM pizza
WHERE price < 10;
```

To get values that are greater/less than *or* equal, you can use `>=` or `<=`:

```
SELECT pizza_id, sale_id, size_id, crust, sauce, price, additional_instructions
FROM pizza
WHERE price <= 10.99;
```

Try running these statements and see how the results change.

Boolean values

Testing for a boolean value is the same syntax as testing for a string value, but you don't use quotation marks.

Run this query, which selects all columns from the `sale` table:

```
SELECT sale_id, total, is_delivery, customer_id
FROM sale;
```

The `is_delivery` column is a boolean value, indicating if the sale was a delivery or not.

To get all sales that were delivery, you can use the `WHERE` clause to test for that value:

```
SELECT sale_id, total, is_delivery, customer_id
FROM sale
WHERE is_delivery = true;
```

Null values

Testing for null values is a bit different, though you still use a `WHERE` clause.

Looking at the `sale` table again, the `customer_id` column represents the customer that ordered a given sale. Some of the values are a number, and some are `[null]`. The sales with a null `customer_id` are walk-in orders where the customer didn't provide their info. The pizza shop doesn't need your name and address if you walked in to place an order for takeout.

To get sales that don't have an associated customer record, you can get the rows that have a null value by writing your statement like this:

```
SELECT sale_id, total, is_delivery, customer_id
FROM sale
WHERE customer_id IS NULL;
```

When you run that, you'll see all the walk-in orders with no customer record.

Conversely, if you want all the sales that *do* have a customer record, you can change it to `IS NOT NULL`:

```
SELECT sale_id, total, is_delivery, customer_id
FROM sale
WHERE customer_id IS NOT NULL;
```

Next steps

You don't always have to retrieve all the columns for a particular table. You can `SELECT` just the ones you want. Try removing one or two columns from one of the SQL statements in this tutorial, and see how the results change.

If you don't know the names of the columns, or you want a quick way to get all of them, you can write `SELECT * FROM ...` instead. The `*` means "return all columns." This is helpful for a quick query, but it's a best practice to always name the columns you want. You'll learn why that's important in a later lesson.

Explore the `customer` table by writing your own `SELECT` statements. Can you get all customers in a particular town? What about customers who haven't provided an email address and phone number?