SQL joins tutorial

In this tutorial, you'll learn:

- How to interpret database diagrams to determine how tables are related
- How to use SQL JOIN clauses to combine data from multiple tables in a database query
- The difference between INNER and OUTER joins and when to use one or the other
- The difference between a LEFT and RIGHT join

Database diagrams

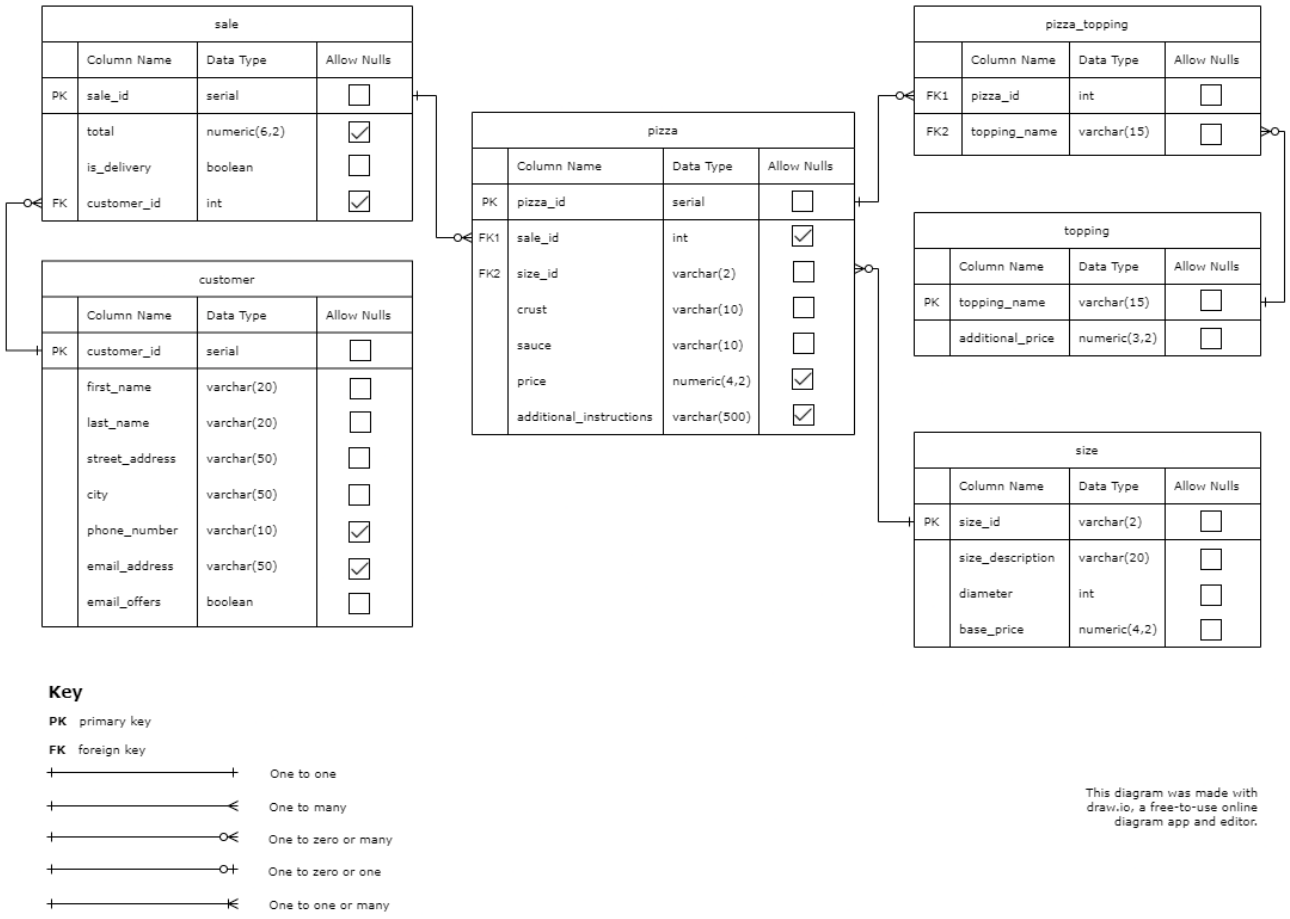
As you have seen in the previous tutorials, the records in database tables often represent a single thing, whether it's a pizza, a topping, or a customer. These things are often referred to as **entities**.

You've also seen that tables can relate to one another. For example, the `pizza` table references the `size_id` column in the `size` table. These are called **relationships**.

It could be difficult to keep all the entities and their relationships in your head. Whether you're joining an existing project, or you've come back to something you created after a few months, you need a way to look at a database and understand its entities and relationships. An **entity relationship diagram**, or ERD, can help you do that.

An ERD typically displays entities as boxes, with the table name, columns, and data types. The relationships are shown as lines between the tables, often pointing at the columns that are shared between tables, like the `size_id` column.

Here's an ERD for the `PizzaShop` database that you've been working with:



In this diagram you might recognize some of the tables and columns that you've been working with. You can also see which columns are linked between tables, establishing their relationships.

The columns linked to another table are known as **foreign keys**. The `size_id` column in the `pizza` table is a foreign key. Foreign keys are typically linked to the **primary key** of another table. The `size_id` column in the `size` table is a primary key.

During this tutorial, you'll write SQL queries that join the information between tables on these relationships. Feel free to refer back to this diagram as you work.

Open the `joins-tutorial.sql` file in pgAdmin. Add the queries for this tutorial under the corresponding sections. All of these queries use the `PizzaShop` database.

Part one: Inner joins

Joining `pizza` and `size` tables

First, review the data in the `pizza` and `size` tables:

```
SELECT pizza_id, size_id, crust, sauce
FROM pizza;

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

The `pizza` table has a `size_id` column on it, which is linked to `size_id` in the `size` table.

What if you wanted to display the diameter of each pizza? You can use a `JOIN` clause to join the two tables on this related column.

This is the syntax for a `JOIN` clause:

```
FROM table_1
JOIN table_2 ON table_1.primary_id = table2.foreign_id;
```

So, to join the `pizza` and `size` tables based on the `size_id` column, use this query:

```
SELECT pizza_id, size.size_id, size_description, diameter, crust, sauce
FROM pizza
JOIN size ON pizza.size_id = size.size_id;
```

If you run that query now, you can see that you're pulling in information from both tables.

> Note the `size.size_id` in the `SELECT` clause. If you want to use a column that appears in both tables, you need to specify which table you want it from.

Look at the ERD again, and notice the line between the `pizza` and `size` tables. The line has a `|` on the `size` side and a three-pronged line called a **crow's foot** with a circle on the `pizza` side. This line represents the relationship between the two tables, which is a **one-to-many relationship**. This means:

- A pizza has only **one** size.
- A size can be used by **many** different pizzas.

A pizza can have one size and only one size. A size can be used by as many pizzas as you want, or none at all.

The circle with the crow's foot means zero-or-many, which means that the a `size` doesn't need to appear in the `pizza` table. Often "one-to-many" means "one-to-zero-or-many" in a database relationship. You may see this line without the circle in other ERDs—know that it means the same thing.

Joining `sale` and `customer` tables

Take a look at another one-to-many relationship—the one between the `sale` and `customer` tables. The line that goes between the `customer_id` columns means it's a one-to-many relationship. That means a sale can have only **one** customer, and a customer can have **many** sales.

First, review the data in the `sale` and `customer` tables:

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

SELECT customer_id, first_name, last_name
FROM customer;
```

To combine the data from both tables, so you have the customer names and their totals together, you'd write this:

```
SELECT sale_id, customer.customer_id, total, first_name, last_name
FROM sale
JOIN customer ON sale.customer_id = customer.customer_id;
```

If you look at the data output, you'll see that sales ids 3 and 12 are missing. To figure out why, look at the data again in the `sale` table. Notice that the `customer_id` for those sales are null. Remember that these were walk-in orders, and the pizza shop didn't require a customer record for that type of sale.

When you use `JOIN` as in these examples, it's implicitly an **inner join** which means that every row in each table *must* match with a row in the other. Null values don't match with anything, even other null values.

> Note: it's valid syntax to write `INNER JOIN` in your query. You can try it with any of the examples in part one if you'd like.

Joining `pizza`, `pizza_topping`, and `topping` tables

Take a look at the `pizza`, `pizza_topping`, and `topping` tables in the ERD. The `pizza` and `topping` tables both have a one-to-many relationship with the `pizza_topping` table. The `pizza_topping` table represents the toppings that go on pizzas that have been ordered.

If you've ever had pizza, you know that a pizza can contain **many** toppings, and a topping can be on **many** pizzas. How do you model such relationship in a database?

This is called a **many-to-many relationship**, which is typically modeled with an *associative table* or a *join table* that each entity has a one-to-many relationship with.

You can join either table to the `pizza_topping` table by its primary key:

```
SELECT pizza.pizza_id, sale_id, size_id, price, topping_name
FROM pizza
JOIN pizza_topping ON pizza.pizza_id = pizza_topping.pizza_id;

SELECT pizza_id, topping.topping_name, additional_price
FROM topping
JOIN pizza_topping ON topping.topping_name = pizza_topping.topping_name;
```

If you need to link the `pizza` and `topping` tables—for example, getting the additional price of each topic—you can use multiple `JOIN` clauses in a single query:

```
SELECT pizza.pizza_id, sale_id, size_id, price, topping.topping_name,
additional_price
FROM pizza
```

```
JOIN pizza_topping ON pizza.pizza_id = pizza_topping.pizza_id
JOIN topping ON topping.topping_name = pizza_topping.topping_name;
```

Part two: Outer joins

Think about the earlier example with `sale` and `customer`. What if you wanted to show all sales, including ones with a null customer id? For that, you can use an **outer join**.

Outer joins come in three types—`LEFT`, `RIGHT`, and `FULL`.

`LEFT` outer join

Take that previous query and add `LEFT` before the `JOIN`:

```
SELECT sale_id, total, first_name, last_name
FROM sale
LEFT JOIN customer ON sale.customer_id = customer.customer_id;
```

Now you see all sales whether or not there's a customer record associated with it. For sale ids 3 and 12, you see `null` for `first_name` and `last_name`. This is a left join. It takes all the records in the *left* table—that's the `sale` table—and matches them with records in the *right* table—`customer` in this case. If there's no matching record in the right table, you get `null` back.

> Note: it's also valid to write `LEFT OUTER JOIN`.

`RIGHT` outer join

So what happens if you change the `LEFT` to `RIGHT`? As you might expect, it takes all the records from the *right* table and matches them up with the *left* table, returning `null` if there's no match.

Take the same query, reverse the table names, and change the join to `RIGHT`:

```
SELECT sale_id, total, first_name, last_name
FROM customer
RIGHT JOIN sale ON sale.customer_id = customer.customer_id;
```

In the previous query, `sale` was the *left* table, and now it's the *right* table. When you run this query, you get the exact same results.

> Note: it's also valid to write `RIGHT OUTER JOIN`.

If you change that `RIGHT` join to a `LEFT` join, or reverse the table names again, you'll get a different result set than the preceding ones:

```
SELECT sale_id, total, first_name, last_name
FROM customer
```

```
LEFT JOIN sale ON sale.customer_id = customer.customer_id;
--or
SELECT sale_id, total, first_name, last_name
FROM sale
RIGHT JOIN customer ON sale.customer_id = customer.customer_id;
```

Either of the queries returns all customers—regardless if they have an associated order or not—and joins only the orders that have a customer. These customers without orders may be ones who registered on the pizza shop's website, but haven't placed an order yet.

Next steps

There's a third outer join type—`FULL`. What happens when you change one of the preceding `LEFT` or `RIGHT` joins to `FULL`?

The last query of the inner join section—the one linking `pizza`, `pizza_topping`, and `topping` together—has duplicate pizzas listed, such as pizza ids 16 and 17. That's because of multiple toppings on a single pizza. What if you just wanted to know the additional charge for each pizza, and not the individual topping names? Hint: use `SUM()` and `GROUP BY`.