5. Joining for Further Insight

Contents

[1. Overview 1](#_Toc10910)

[2. Keys to the Relationship 1](#_Toc19273)

[3. Inner Joins 2](#_Toc32135)

[4. Table Aliases 3](#_Toc3392)

[5. Outer Joins 4](#_Toc300)

[6. Full Joins 5](#_Toc14546)

[7. Implementing Joins 5](#_Toc8595)

[8. Summary 6](#_Toc6759)

# 1. Overview

=>slides: Pg. 1

In this module, we'll explore how to join tables.

=>slides: Pg. 2

So far in this course we've learned how to select data from a single table and apply filters using the WHERE clause. An important use of SQL, however, is the ability to combine records and data from multiple tables. One of the most common ways to do this is by using joins. In fact, joining data is what makes a relational database truly relational. When we join tables in SQL, we combine tables based on information that is common, or shared, between the tables. There are three primary types of joins, inner joins, outer joins, and full joins. We will take a look at each type of join in this module.

# Keys to the Relationship

=>slides: Pg. 3

When we use a relational database, each table stores information about particular segments or types of data. For example, we may have one table that stores customer information, and another table that stores order information. How do we match, or join, information between these two tables? In SQL, information is joined through the common fields, or keys. Keys are fields that describe relationships between tables. There are two types of keys, primary keys and foreign keys. A primary key is a column or a set of columns that when taken together have no duplicate values across rows. The primary key uniquely identifies each record in the table. A foreign key is a set of one or more columns in a table that refers to the primary key of another table. The foreign key in a table is used to point to data in another table.

=>slides: Pg. 4

In the customer table, for example, customer\_id is a primary key. This ID number uniquely identifies a row of information that is applicable to a specific customer. In the orders table, order\_id is a primary key. This ID number uniquely identifies a row of information that pertains to a specific order. Notice that the orders table also includes the customer\_id column. This customer\_id allows us to join customer information to the order information. The customer ID tells us which customer placed the order, therefore the customer\_id is a foreign key on the orders table and refers to data stored in the customers table. In this instance, customer\_id 125 is the foreign key for order 9002. This tells us that Clair Fletcher placed this order.

# Inner Joins

=>slides: Pg. 5

The first type of join that we will consider is an inner join. An inner join returns all rows from two or more tables that meet our joined condition. The fields that are being joined must exist and match in both of the tables being joined. This can be represented by the Venn diagram shown at left. Notice that only those rows that exist in both Table A and in Table B would be selected.

=>slides: Pg. 6

Let's say that we have our customer and order tables, and that we want to see all of the columns from both tables if a customer has ever placed an order. This is an example of an inner join, and this is how the querying might look. Let's examine the structure of this query. The SELECT and FROM clauses should look familiar to you. Remember, these tell us what column we are interested in and what table the query should use. However, the column names may look a bit different. The asterisk wildcard tells us we want all columns from our table, but with a join we are pulling data from two or more tables. In this case, PostgreSQL requires us to specify what fields we are interested in from each table. We do this by prefacing the field name with the table name, separated by a period. So, this code tells us that we want all fields from the customers table and all fields from the orders table. The INNER JOIN and ON keywords are new. INNER JOIN simply tells Postgres the type of join we are interested in. After we specify the type of join, we specify the table we want to join to. So, in this example, we are saying we want to take the customer table and use an INNER JOIN to join it to the orders table. The ON clause tells us how we want to join the tables. What fields do we want the tables to have in common to join them? Remember that joins will usually be based on primary or foreign keys. In this example, both the customers table and the orders table have a customer\_id number. This is the primary key on the customers table, and it is a foreign key on the orders table. We want to join the tables using this key, so we tell SQL to join ON customers.customer\_id = orders.customer\_id.

=>slides: Pg. 7

Running this query will give us the following results. All customer IDs from the customer table were joined with order information for those customers who have placed an order. Notice that John Smith, customer 121, has no records in the order table and does not appear in the result set. Estella Dodd, customer 124, has two records in the orders table, and thus appears twice in the result set. As we discussed earlier in this course, indiscriminate use of the asterisk wildcard is generally not best practice and is not recommended. Notice in this result set that we have columns we may not need. For example, customer\_id appears twice.

=>slides: Pg. 8

We should rewrite this query to return only specific columns of interest. This code would limit our results to provide a simple list of order dates and amounts by customer first name and customer last name.

=>slides: Pg. 9

We can also use the WHERE clause to limit our results based on criteria from either table. The WHERE clause must follow the ON criteria. When listing WHERE criteria, be sure to specify which table contains the field of interest. If we are interested only in customers with the last name of Dodd, we could add WHERE customers.last\_name = Dodd to this query to narrow our result set.

=>slides: Pg. 10

Inner joins are the most common type of join in SQL; therefore, SQL has two shortcut methods of specifying this join type. The three statements are functionally equivalent. If you type simply JOIN, Postgres defaults to an inner join. You can also omit the specification altogether and list the names of the two tables separated by a comma. If you do this, SQL implies an inner join. However, since you did not specify a join explicitly, you cannot use an ON clause. To use ON, you must have a matching JOIN keyword. In this case, you would specify your JOIN criteria as the first line of the WHERE clause. I include these three methods here so that you can identify them as you are looking at code written by others, but it is best practice when writing code to be clear about the type of join behavior that you expect. Therefore, I strongly recommend that you always specify inner join and explicitly list your ON criteria.

# Table Aliases

=>slides: Pg. 11

Before we look at the other types of joins, let's look at an important concept that can make writing these queries easier, table aliases. We've already seen a form of aliasing, column aliases, earlier in this course. When we type something such as SELECT order\_amount as amount, we are aliasing the order\_amount column. The column is not actually renamed in the underlying database, but it is given a temporary label in the query.

=>slides: Pg. 12

In addition to columns, we can also alias tables. Table aliases are particularly helpful in more complicated queries that request data from multiple tables, and can be very useful when authoring joins. For example, let's look at the query we wrote to find selected columns of information on orders placed by customers with the last name of Dodd. An alias can be assigned to a table using the FROM clause. After the table name, simply type AS and the alias to relabel the table. Tables can also be aliased via a shortcut. You can omit the AS keyword and just follow the table name with a space and the alias. For example here, we've aliased the customer's name as simply c, and the orders table as simply o. This shortcut method is commonly used in practice. Instead of repeatedly typing the entire name of the table in the query, we can instead refer to the tables by their aliases. Notice in the SELECT clause that the customers table is now referred to as c and the orders table is referred to as o.

=>slides: Pg. 13

Aliasing tables is common practice when writing SQL, but different people use different conventions when creating their labels. Some people will use the first letter of the table name as their alias. This is the convention used in this example. Others will assign a shortcut alias that clearly signals what table is being referenced. For example, customers may be shortened to cust, and orders may be shortened to ord. These shortcuts are still clear what table we're referencing. Others, myself included, prefer to use an ABC convention where the alias is advanced by one letter each time that a table is added. FROM customers a, INNER JOIN orders b, and so on. To me, this can be easier to read when analyzing more complicated queries. Others, however, argue that this is an arbitrary assignment and that the other labels more clearly indicate what table is being referenced.

=>slides: Pg. 14

As usual, there are no hard and fast rules when writing SQL to specifying aliases. You can use the entire table name, or you can alias it. You can use the first letter of the table name, you can create a shortcut, you can use the ABC sequential convention, or you can use what works for you. As you begin to write more SQL, you'll discover the practice that you think works best. Just remember to be consistent so that your code is easily readable by others.

# Outer Joins

=>slides: Pg. 15

Let's get back to looking at the other types of joins. Remember that in an inner join, we return only those records where the join field exists on both tables. The second type of join is an outer join. Outer joins can be either a left outer join or a right outer join. The direction specifies which table we want to have precedence. Let's illustrate this using two tables, Table A and Table B. The left join selects all records from the left table, A, along with any records that exist in the right table, B, that meet the join condition. If there is no match on Table B for a given row from Table A, the row will still appear in our result set. In a right join, we do just the opposite. We want to select all records from the right table, Table B, along with any records from the left table, Table A, that meet the join condition. All rows from Table B will appear in the result set regardless of whether there's a match in Table A or not.

=>slides: Pg. 16

For an example, let's return to our customer and order tables. Remember that in an inner join, we wanted to see all of the customers from both tables if a customer has ever placed an order. If a customer had never placed an order, however, they would not appear in our result set. Perhaps we want a list of all customers, regardless of whether they've placed an order. But if they have placed an order, we want to know details about that too. This is the perfect example of a left join. Remember that a left join returns all rows from the first table and joins that data to any rows from the second table meeting the join condition. Here's how we would construct this query. Notice that instead of INNER JOIN, we specified a LEFT OUTER JOIN. As before, we then specify what table we want to join to. So in this example, we're saying that we want to take the customers table and use a LEFT OUTER JOIN to join it to the orders table. We can read this in plain English as customers ON left, then JOIN on orders. Our ON clause is the same. We want to join the tables using the customer\_id.

=>slides: Pg. 17

Running this query will give us the following results. Notice that all of the records from the customer table, the left table, appear, as well as records from the orders table when the customer has a matching record. John Smith, customer 121, has never placed an order. However, since he has a record in the customers table, a row is still created for him. There is no matching information from the orders table, so these columns are null.

=>slides: Pg. 18

A right outer join is very similar. However, instead of returning all records from the left table, Table A, we want to return all records from the right table, Table B, regardless of whether there is a match or not on the left table. This is how we might construct a right outer join query. Notice that instead of LEFT OUTER JOIN, we've typed RIGHT OUTER JOIN. As before, we've then specified the table we want to join to. So in this example, we're saying that we want to take the orders table and use a RIGHT OUTER JOIN to join it to the customers table. We can read this in plain English as orders on right, then JOIN customer. Our ON clause is the same. We want to join the tables using customer\_id.

=>slides: Pg. 19

To demonstrate how this query works, we will add a record to the orders table that does not have a matching customer ID. See order ID 9006. Perhaps when this ID was taken, the clerk failed to record the customer ID. Running this query will give us the following results. In this case, in our result set we still see order number 9006 for $ 920.40, even though there was no customer ID to match.

=>slides: Pg. 20

Both left and right joins are examples of outer joins. SQL, therefore, allows us to omit the OUTER keyword in the LEFT OUTER JOIN and RIGHT OUTER JOIN statements when we are writing queries. For example, LEFT OUTER JOIN becomes simply LEFT JOIN. LEFT OUTER JOIN is functionally equivalent to LEFT JOIN, just as RIGHT OUTER JOIN is functionally equivalent to RIGHT JOIN. It is generally accepted and common practice in SQL programming to omit the OUTER keyword.

=>slides: Pg. 21

Note that the left outer join is far more prevalent in practice than the right outer join. Generally, the left outer join can be easier to read and interpret. I encourage you to primarily use left joins and to only use right joins when absolutely necessary. For example, this query could easily be rewritten using a left outer join. Notice that to do so, the table names merely swapped position with relation to the keyword operator. Instead of using orders as a RIGHT JOIN, we merely move orders to the left of our join criteria and use a LEFT JOIN.

# Full Joins

=>slides: Pg. 22

The last type of join we will examine is the full outer join. A full join will return all records from the left table and all records from the right table, regardless of whether or not the join condition is met. If there is no match from one table or the other, the missing side will have values indicated as null.

=>slides: Pg. 23

Let's return to our customer and orders table. Remember that in our customers table, we have customers who have never placed an order. In our orders table, we also have the one order that does not have a matching customer ID. If we want to retrieve all rows of data from both tables, including those records without fully matching values, we could write this query. In this example, we are saying that we want to take the orders table and use a full outer join to join it to the customers table. We can read this in plain English as customers fully joined with orders, and it is exactly the same as saying FROM orders FULL OUTER JOIN customers. Our ON clause is also the same. We want to join the tables using the customer ID.

=>slides: Pg. 24

Running this query will give us the following results. In this case, we see that the order that did not have a customer ID that matches still appears in our result set, even though it did not have a corresponding record in the customers table. We also see that our customers who have not placed any orders, and therefore have no matching records in the orders table, still appear in our results. Notice that the full join is also a type of outer join. Just like left and right joins, SQL allows us to omit the outer portion of the FULL OUTER JOIN keyword when we are writing queries.

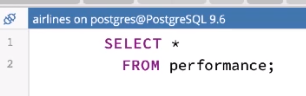
# Implementing Joins

=>slides: Pg. 25

Joins are frequently used with database tables known as lookup tables. Lookup tables, or validation tables, are database tables that contain data to determine the value of codes. An example might be a lookup table of state codes where each state name was spelled out alongside each abbreviation. We could join the state abbreviation in the orders table to the state abbreviation in the lookup table to retrieve the full state name for each order. Lookup tables are commonly used in practice with data such as products IDs, semester codes, month abbreviations, and the like.

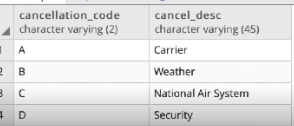
=>slides: Pg. 26

Let's use our sample database of airline on-time performance data to illustrate the use of joins. In this example, we'll use a lookup table to convert our marketing carrier codes to actual airline names.



Here we are again in our database of airline performance statistics for the month of January. The main table we've used to this point is the performance table, which includes a row for each flight operated on a given day for a given market carrier in the United States. Notice that we've added two additional tables to this database, the codes\_cancellation table and the codes\_carrier table. Both of these tables are what are known as lookup tables. That is, they store data that allows us to look up the value of given codes. Let's quickly look at the contents of these two tables.





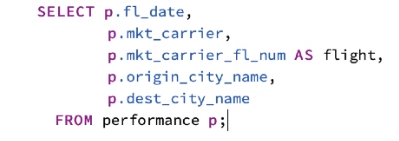
The codes\_cancellation table contains two columns, cancellation\_code and cancel\_description. The cancellation\_code is a one-character code that indicates the reason a flight is cancelled. The cancel\_description column includes descriptive text to better explain that code. For example, code B means the flight had a weather delay.

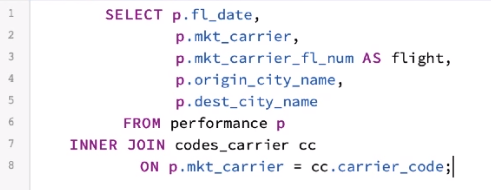


The codes\_carrier table also contains two columns, carrier\_code and carrier\_description. Carrier\_code is the two-digit alphanumeric code that the federal aviation administration assigns to each airline in the United States. The carrier\_description column indicates the name of the air carrier.

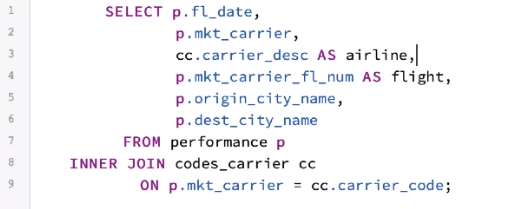


We now return to our performance database.

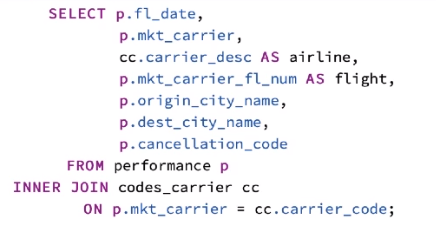


Let's select the columns that contain data we may want to use for our analysis. In this case, we'll select the flight date, market carrier, flight number, origin city, and destination city. We will also use a table alias for the performance table, p. This gives us a list of flights operated by the given carrier each day, along with origin and destination city information. This probably looks familiar to you. Now let's join these data to other tables to make this data more informative. Consider that we may not know what airline the market carrier code represents. We can use the codes\_cancellation lookup table to solve this problem. 

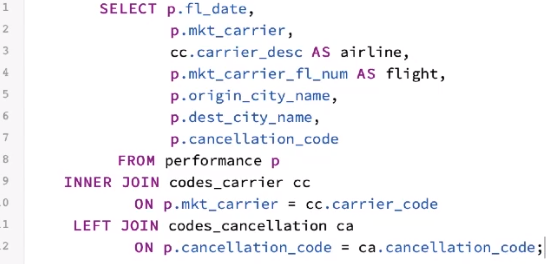
Let's add an inner join to join the performance table to the codes\_carrier table, which we'll alias cc. This tells Postgres that we want to return data only when there is a matching row in both the performance and the codes\_carrier table. In this case, we want the data in the two tables to match on the foreign key, mkt\_carrier, in the performance table, and the primary key, carrier\_code, in the codes\_carrier table. An inner join is appropriate here because there should be no record in the performance table that does not have a matching row in the codes\_carrier table. If we run this query now, we'll notice that there is no difference in our result set. This is a common mistake. We must include the new columns we wish to see by adjusting our SELECT clause.



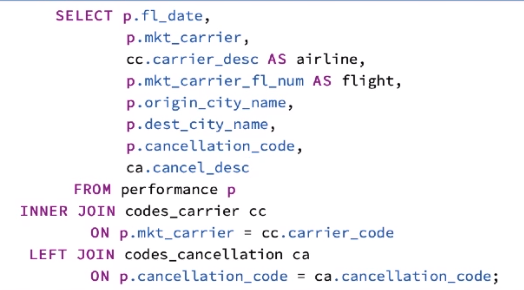
Simply add the carrier\_description field to the SELECT clause, being sure to include the cc alias to correctly point Postgres to the codes\_carrier table. For clarity, we will alias this field as airline. Running this query again, we can see that we now have both the mkt\_carrier code from the performance table, as well as the description from the codes\_carrier table. We can see that UA is United Airlines, and if we look further in our dataset, we can see that AS is Alaska Airlines, and DL is Delta Airlines. We can also use multiple tables when we are joining by building on to our existing query. The performance table also includes delay and cancellation information. For example, the cancellation\_code column in the performance table contains an alphabetical code to indicate why a particular flight was cancelled.



Let's add this field to our query. If we execute this query, we'll see that a majority of flights have a null value in this column. In fact, we must scroll to United Airlines flight 2034 from Houston to Mission before we see a value in this field. The value B is not informative if we are doing analysis. We need to know why this flight was cancelled. To do this, we can join our performance table to the codes\_cancellation lookup table.



To add an additional join, we simply continue writing our query below the most recent join. In this case, we want to left join our performance table to the codes\_cancellation table, which we will alias ca. We want to return values from the codes\_cancellation table when a matching cancellation code exists on the performance table. Why a left join? Remember, the majority of flights do not end up cancelled. Therefore, the majority of the time the cancellation\_code field should be null. There would be no match in the codes\_cancellation lookup table for these records. We want to see the data for all flights, regardless of if they were cancelled. The left join returns all rows from the left table, which in this case is the performance table, and that will be the result that we seek.



We add the cancellation\_description field from the codes\_cancellation table to our SELECT statement. After we run this query, again, notice that the cancellation\_code and descriptions columns are primarily null. This is what we expect. If we look again at United Airlines flight 2034, however, we see the cancellation code B. We also see that the description column has been added from the lookup table.

# Summary

=>slides: Pg. 27

In this module, we learned to join tables based on keys using three primary join types, inner joins, outer joins, both left and right, and full joins. Thinking of joins in terms of Venn diagram representations can be very useful as you learn them and select the join to apply and practice. Let's briefly review these join types. The inner join selects all records from Table A and Table B where the join condition is met in both tables. The left outer join selects all records from Table A, along with records from Table B where the join condition is satisfied. The right outer join does the reverse. It selects all records from Table B, along with records from Table A for which the join condition is met. And the full join selects all records from Table A and Table B, regardless of whether the join condition is met. Joining provides the ability to combine related information between tables. Joining and filtering data with the WHERE clause are two of the most fundamental elements of powerful analysis using SQL.

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