Previous Next

Unit 1 / Lesson 2 / Project 5

Joins and CTEs

Estimated Time: 2-3 hours

So far we've worked with one table at a time in our database, but we aren't limited to that. SQL lets you easily combine multiple tables using a `JOIN` clause, or "joins".

Basic Joins

So how does a **join** work? It's quite simple. When writing a `JOIN` clause, you will indicate one or more *pairs* of columns on which you want to join two tables. The two tables will then be joined when the value in each pair matches, returning rows with the data from both tables for every such case.

By default, SQL will perform an **inner join**. This means that the rows returned will have a successful join between the tables. If there is no match between the given columns, the rows will not be returned. Let's give it a try, continuing with the bikeshare data set.

To get a sense of its most basic functionality, let's join the `station` and `trip`

> tables to return the latitude and longitude for the start station for every trip, along with the trip id.

```
SELECT
    trips.trip_id,
    trips.start_station,
    stations.lat,
    stations.long
FROM
    trips
JOIN
    stations
ON
    trips.start_station = stations.name;
```

Let's go through how this join works. Firstly note that the join comes after the `FROM` statement. They actually work together as you are selecting from both the initial table as well as the joined tables. In a two table join the order does not matter, but it is worth noting that the joins will happen in order, which can matter in more complex multi-table joins.

Secondly, note that we're joining the `start_station` column of the `trips` table on the `name` column of the `stations` table. These two columns are how we figure out how to combine the `trips` table and the `stations` table into one composite table.

Thirdly, in our `SELECT` statement we're choosing four columns from the composite (or "joined") table that we'd like to output: `trip_id` & `start_station` from the `trips` table and `lat` & `long` from the `stations`

table. We *could* use `*` here to select all columns from both tables if we wanted to, and you can run that code on your database now if you want to see what the full join table looks like. In practice, though, you almost always want to pull out specific parts of your join.

Table aliases

Let's rewrite the query above to demonstrate **table aliases**, a commonly used SQL feature. This query is *exactly the same*:

```
SELECT
    t.trip_id,
    t.start_station,
    s.lat,
    s.long
FROM
    trips t
JOIN
    stations s
ON
    t.start_station = s.name;
```

Notice that we follow the tables in our from and join statements with a space and then a single letter and then use this shorter name to refer to the tables in our select statement. This looks a lot like the syntax we use to rename columns and works similarly by allowing you to refer to a table with another name. You don't need to use aliases. You could use the actual table name as we did in the first example. However, aliases are very common and you'll see them a lot as you're reading other people's SQL.

Should *you* use aliases? No one agrees on this, including the course authors. As a rule of thumb you *should* use them if doing so and using a descriptive alias would make your code easier to read, or if you *have* to use them because you're joining one table on itself and need two different aliases to refer to it. You probably *shouldn't* make a one-character alias for every table ever just because you hate typing. As with all the code you write, your goal should be to make your code as easy to understand and as easy to maintain as possible.

Types of joins

There are several ways you could theoretically combine two tables. We mentioned above that by default, SQL will perform an **inner join**, which means that the only rows returned are the ones where there is *both* a match on the left table *and* a match on the right table. Because rows are only returned when there is a match on both sides, it doesn't matter which table is on the left and which table is on the right. SQL assumes you want an inner join unless you tell it differently, so using `JOIN` is exactly the same as using `INNER JOIN`.

There are other ways we could choose to join tables, though. Beyond *inner* joins there are also three types of **outer joins**: left outer joins, right outer joins, and full outer joins.

In a **left outer join** every row from the left table will be included in your output, *even if there was no matching row on the right table*. Tables, like text,

are read from left to right, so the left table is the first table you name and the right table is the second table you name. Rows without a match will be filled with `NULL` for the columns from the right table.

Left outer joins are often called just "left joins", and you can perform a left join using `LEFT OUTER JOIN` or just `LEFT JOIN`.

A **right outer join**, or "right join" is exactly the same as a left join, except that all the rows from the right table are returned, even if there is no match. Because the only difference is table order, you could reverse the left and right tables and use a left join instead to accomplish the same thing. You can perform a right join with `RIGHT OUTER JOIN` or `RIGHT JOIN`.

A **full outer join**, also known as a "full join" or just an "outer join" returns all matching records from *both* the left and right tables. This can potentially return very large data sets, enough data to choke your laptop or even a production database server.

The default join in SQL is an inner join because they are much more common than outer joins. And when you *do* want to use an outer join you'll usually end up using a left join.

If you want more practice on JOINs, this tutorial is a good resource as a good resource, and you'll also want to have a look at the postgreSQL documentation on joins.

CTEs (common table expressions)

You might have noticed that the result of every SQL query you write is a table. In case you hadn't: the result of every SQL query is itself a table. That means you can use joins not just to join tables on existing tables, but also to join them on the results of other queries. One way to do that is to use **common table expressions**, or CTE's. There are two basic ways to use CTE's in SQL: step processing for queries (running a query that is too complex for a single execution and instead requires discrete steps) or preprocessing to facilitate a join.

Let's go through a preprocessing example here. Recall before when we generated the average latitude and longitude of every city. What if we wanted to also include a count of the number of trips that started in each of those cities? This could be useful if you want to map the cities with information about trip volume.

If we tried to do this through a single query we might try something like this:

```
SELECT
    s.city,
    AVG(s.lat) lat,
    AVG(s.long) long,
    COUNT(*)
FROM
    stations s
JOIN
    trips t
ON
    t.start_station = s.name
GROUP BY 1;
```

However, this query is actually *incorrect*. When working with `JOIN`s, the join happens *before* any aggregate functions. So in the example above we're actually taking the average of the latitude and longitude for every trip that occurred, so we'll be skewed to more popular station's coordinates. To do this properly we can use a CTE.

CTE's start with the form `WITH __expression__ as (...)`.

This will create another, intermediate table for you to work with and join on. It's easiest to see in action, so let's rewrite that query above, this time using a CTE.

```
-- Set up the CTE to create a "locations" table.
WITH
    locations
AS (
    -- A simple query to get the averages of lat and long on a city
    SELECT
        city,
        AVG(lat) lat,
        AVG(long) long
    FROM
        stations
    GROUP BY 1
)

-- Joining the locations table we created with the trips table to co
SELECT
    l.city,
    l.lat,
    l.long,
    COUNT(*)
FROM
```

```
    locations l

-- We need an intermediate join to go from locations to stations
-- because the trips table does not have a "city" column.
JOIN
    stations s
ON
    l.city = s.city
JOIN
    trips t
ON
    t.start_station = s.name
GROUP BY 1,2,3;
```

Let's walk through how this query operates quickly. Firstly, the CTE, `locations`, groups stations by city name to find the average of the coordinates. You are then creating a new temporary `locations` table with an entry for lat and long for each city. That `locations` table is then joined with the trips table.

But we can't directly join the `locations` table, which has a `city` column onto the `trips` table because the `trips` table doesn't have a `city` column to join on. In order to relate locations to trips we must first join `locations` back on the `stations` table (which *does* have a `city` column), then join *that* to the `trips` table on the common `start_station` and station `name` columns. Using multiple joins like this to relate two tables you can't join directly is very common.

All of this comes together to give the average lat, long, and count of the number of trips per city.

CASE

For the final SQL topic we'll cover right now, let's talk about `CASE` statements. `CASE` statements allow you to set up conditions and then take action in a column based on them. It is also common to combine `CASE` statements with `COUNT` to do conditional counts. The most common form for case statements is `CASE WHEN __condition__ THEN __value__ ELSE __value__ END`.

Let's see it in action:

```
SELECT
    (CASE WHEN dockcount > 20 THEN 'large' ELSE 'small' END) statior
    COUNT(*) as station_count
FROM
    stations
GROUP BY 1;
```

This `CASE` statement looks at the `stations` table and labels each row either `'large'` or `'small'` depending on the value of `dockcount` for that row, and then counts how many rows there are for each case. The group by statement makes it so we are counting based on the station size.

You can find more about case statements in this simple tutorial if you'd like to go deeper. They represent the beginning of some more advanced SQL.

Drills

1. What are the three longest trips on rainy days?
2. Which station is full most often?
3. Return a list of stations with a count of number of trips starting at that station but ordered by dock count.
4. (Challenge) What's the length of the longest trip for each day it rains anywhere?

Save your SQL queries in a gist or a GitHub repository and submit a link below. There are many ways to solve drills like this with SQL.

Queries running a little slow? This is a large database for working locally, particularly the status table. It may be helpful to create a smaller version of the status table to help develop these queries so things run faster and you can iterate easily.

Try running something like:

```
CREATE TABLE status_abbreviated AS
  SELECT *
  FROM status
  limit 10000;
```

This will create an additional table with only 10,000 entries from status, making for faster joins and queries. When you think your query is running properly test it against the full table to confirm. Feel free to tune the limit

count to get a trade off between size and speed that you're comfortable with.

Happy querying!

Oh, and if you want to see some example solutions, check here. It is worth noting, however, there are many ways to write a query, so if your solutions look different than this that's fine. Just make sure they work and are reasonably efficient.

Report a typo or other issue

example.com/project

✔ Submit your project

Previous Next