7.3.2

Join the Tables

The carefully crafted database foundation has been filled with data, and now we're able to perform queries. These queries have helped Bobby complete the task that was asked of him, but the number of retirement-ready employees was staggering.

Bobby's been asked to dive back into SQL and create a separate list of employees for each department. We can only gain so much information from our data as it stands, so we'll need to start combining it in different ways using joins.

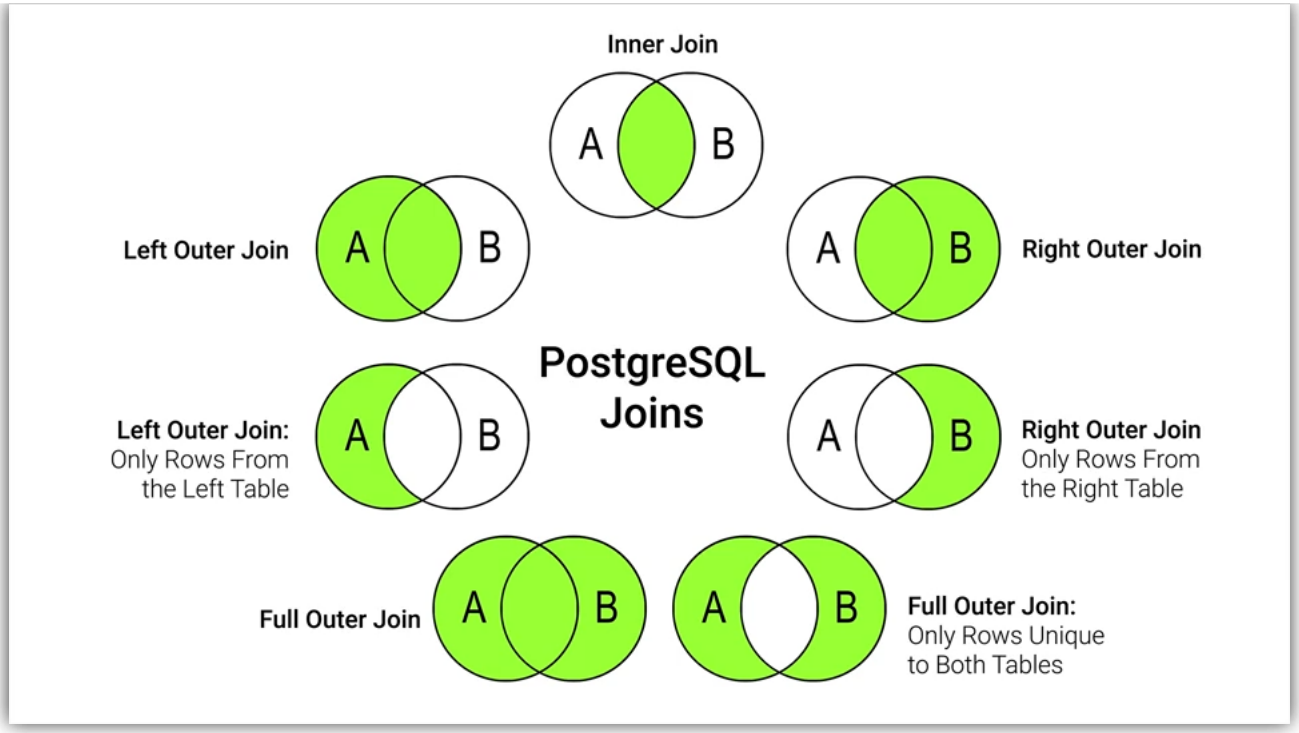
Merging DataFrames in Pandas is very similar to joining tables in SQL. There are several different types of joins and each one will yield a different result. We'll cover each type of join available in SQL as well as where and when to use it.

The list Bobby provided was big and difficult to absorb as one large file. More than 40,000 employees? Retiring around the same time? Yeah, that's a lot.

Bobby will need to break down that large list into smaller pieces, then present those to his boss instead. We'll have a little bit of tweaking to do and a few more queries to build.

Make Sense of Tables with Joins

Right now, there are seven tables available for us to use. Between them, we have all of the information we need to help Bobby create his new lists, but the data is in separate tables. We could present multiple lists and explain the connections, but that's messy and confusing. Instead, we need to perform a join on the different tables. In SQL, combining two or more tables is called a **join**.



Much like joining Python DataFrames, we can join SQL tables together using a common column. We don't have to join complete tables. We can specify which columns from each table we'd like to see joined. Take another look at our ERD.

Bobby's boss wants the same list of employees—only he wants them broken down into departments. We can already see that the Employees and Departments tables don't have a common column between them.

Which tables would you join together to help Bobby create a list of employees grouped by their departments?

- ○ Join the Employees table and the Managers table on the dept_no column.
- ○ Join the Employees table and the Managers table on the emp_no column.
- ○ Join the Employees table to the Managers table, and perform another join on the Departments table.
- ◉ Join the Employees table to the Dept_Emp table on the emp_no column. ✔

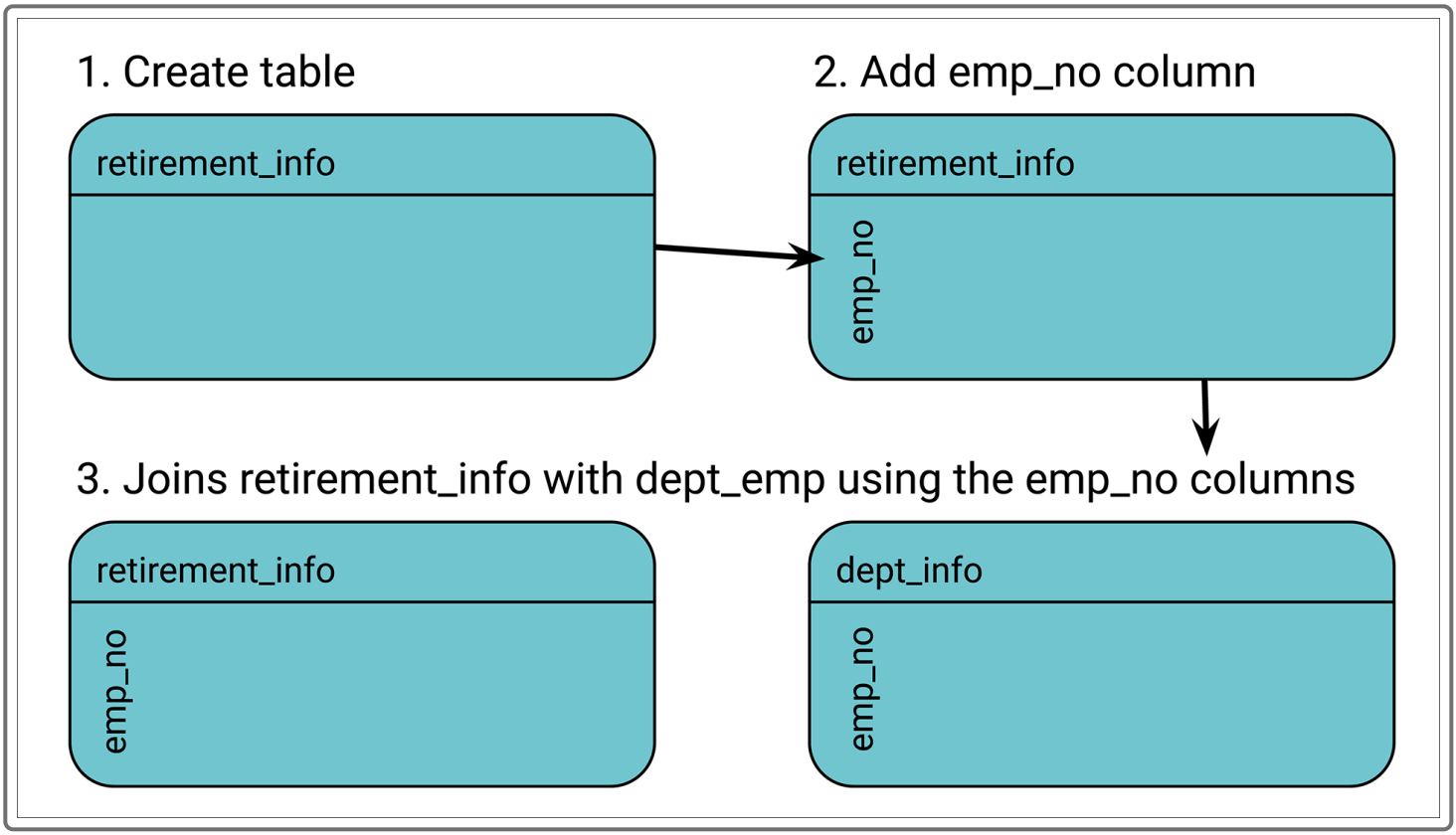
Feedback
Correct. Nice work! We'd only need to join the two tables with no extra steps to get the list for Bobby's boss.

The Dept_Emp table contains both an emp_no and a dept_no. Between the Employees table and the Dept_Em*p* table, we would have:

- Employee numbers
- Employee names
- Their departments

We could combine the two tables to have everything we need for Bobby's boss. Except, the Employees table still contains every employee, not just those getting ready for retirement. Our retirement_info table has the correct list of employees, but not their employee number, so we can't actually join it with anything.

What we'll need to help Bobby do first is recreate the retirement_info table so it includes the emp_no column. With this column in place, we'll be able to join our new table full of future retirees to the Dept_Emp table, so we know which departments will have job openings (and how many).



Recreate the retirement_info Table with the emp_no Column

The first task we'll help Bobby with is recreating the retirement_info table so it contains unique identifiers (the emp_no column). This way, we will be able to perform joins using this table and others.

What is the code for dropping a table?

- `DROP retirement_info;`
- `DROP TABLE retirement_info;` ✔
- `DROP TABLE retirement_info CASCADE;`
- `DROP TABLE retirement_info`

Feedback

Correct. Nice work! This is the correct syntax and format for dropping a table. Because we originally created this table from a query, we haven't formed any connections to other tables and the `CASCADE` constraint isn't needed.

Drop the current retirement_info table. At the bottom of the query editor, type `DROP TABLE retirement_info;` and then execute the code. Next, we're going to update our code to create the retirement_info table. Keeping the ERD in mind, we know we want the unique identifier column included in our new table. Add that to our select statement.

```
-- Create new table for retiring employees
SELECT emp_no, first_name, last_name
INTO retirement_info
FROM employees
WHERE (birth_date BETWEEN '1952-01-01' AND '1955-12-31')
AND (hire_date BETWEEN '1985-01-01' AND '1988-12-31');
-- Check the table
SELECT * FROM retirement_info;
```

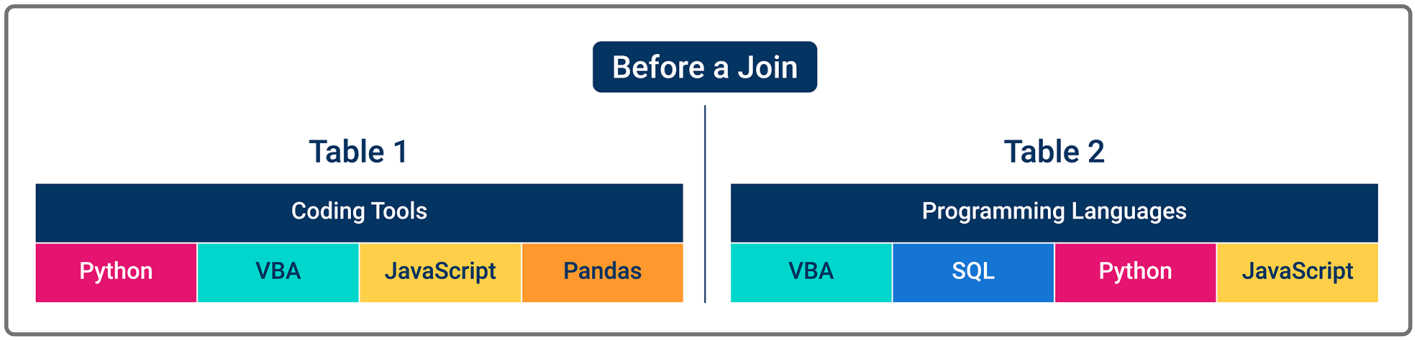
After executing this code, the retirement_info table that's generated now includes the emp_no column. We are ready to begin combining different tables using joins to help Bobby create the new list his boss has requested.

Use Different Types of Joins

Joins are common in every coding language that deals with data. There are several types of joins available for use, depending on what data we want displayed. For example, in the retirement_info table, we want all of the information from the table. We'll be joining it with the dept_emp table, but what information do we want included? Just the dept_no, so we know which department each employee works in.

IMPORTANT

When we talk about joining tables, try to imagine one table on the left and a second table on the right, instead of in a list.



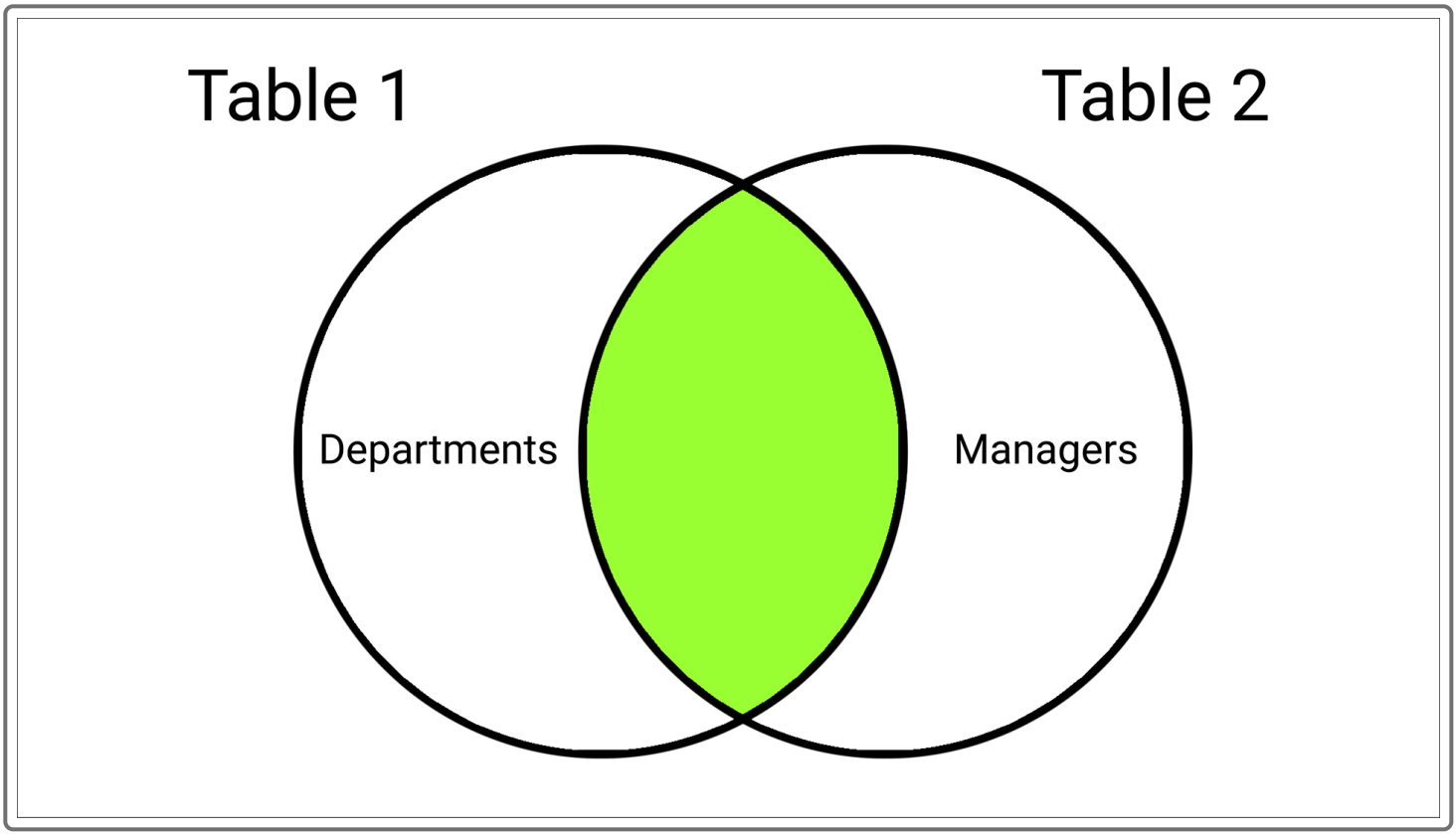
Joins treat the tables in this manner, as if they're side-by-side, and it makes it a bit easier to visualize how the data will be joined.

Which type of join would be best suited for this task? Let's take a look at what is available with Postgres.

Inner Join

An **inner join,** also known as a **simple join,** will return matching data from two tables. Look at the Departments and Managers tables in our ERD. Imagine that Departments is Table 1 and Managers is Table 2 in the diagram. It's

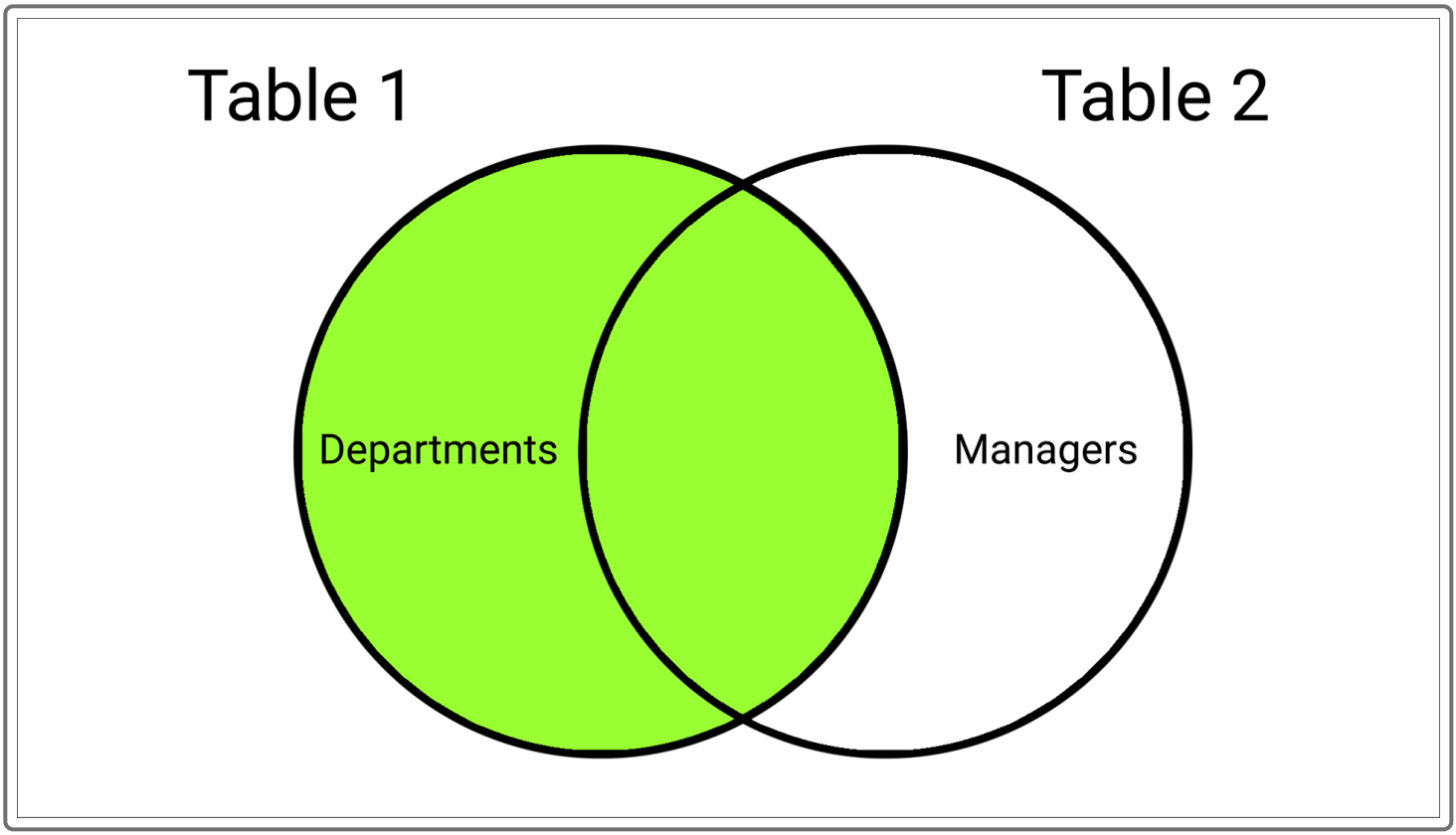
important to have the order of the tables correct, too. SQL views them as the first and second tables, or the left and right tables.



Say we want to view the department number and name as well as the managers' employee numbers. We only want the matching data from both tables, though. This means that if there are any NaNs from either table, they won't be included in the data that gets returned. So if a department doesn't have a manager, it wouldn't be listed in the data.

Left Join

A **left join** (or "left outer join") will take all of the data from Table 1 and only the matching data from Table 2. An inner join is different because only the matching data from the second table is included. So how do we visualize a left join?



In our previous example, we were looking at the Departments and Managers tables. Instead of an inner join, where only the matching data is returned, picture a left join instead.

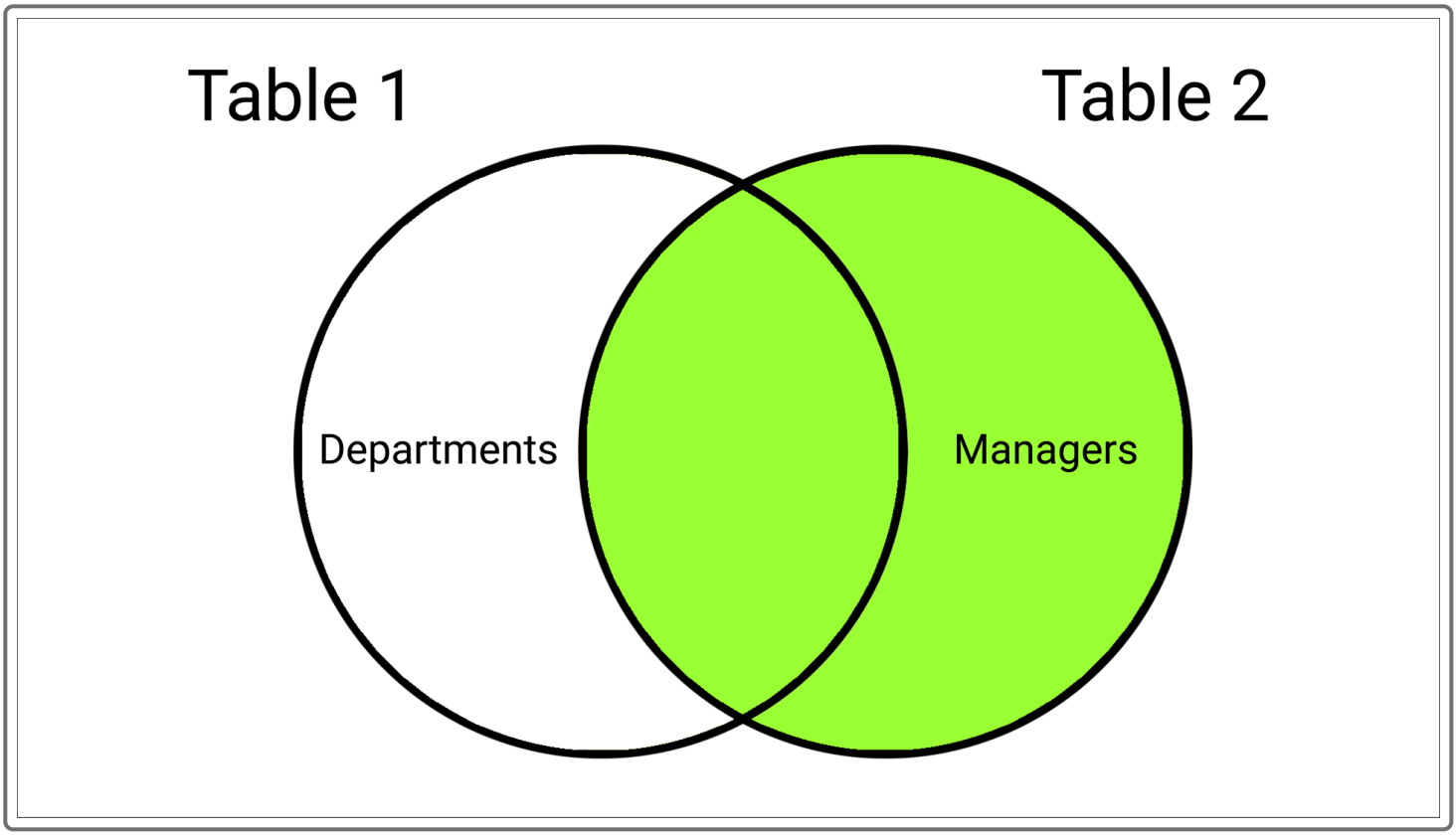
This time, when rows from the Managers table do not have matching data for every row in the Departments table, a NaN is inserted into that column and row intersection instead. This way, all of the data we want included from Table 1 is still present, and data from Table 2 isn't rearranged and mismatched (blank spaces aren't automatically filled with present data).

If we wanted to add information from the Managers table to the Departments table, we'd use a left join. This way, every row of the Departments table would be returned with or without manager information. If a department doesn't have a manager, a NaN would appear in place of actual data, and it would look as follows:

dept_no	dept_name	emp_no	from_date	to_date
d0001	Marketing	NaN	NaN	NaN

Right Join

The **right join** (or "right outer join") is the inverse of a left join. A right join takes all of the data from Table 2 and only the matching data from Table 1.

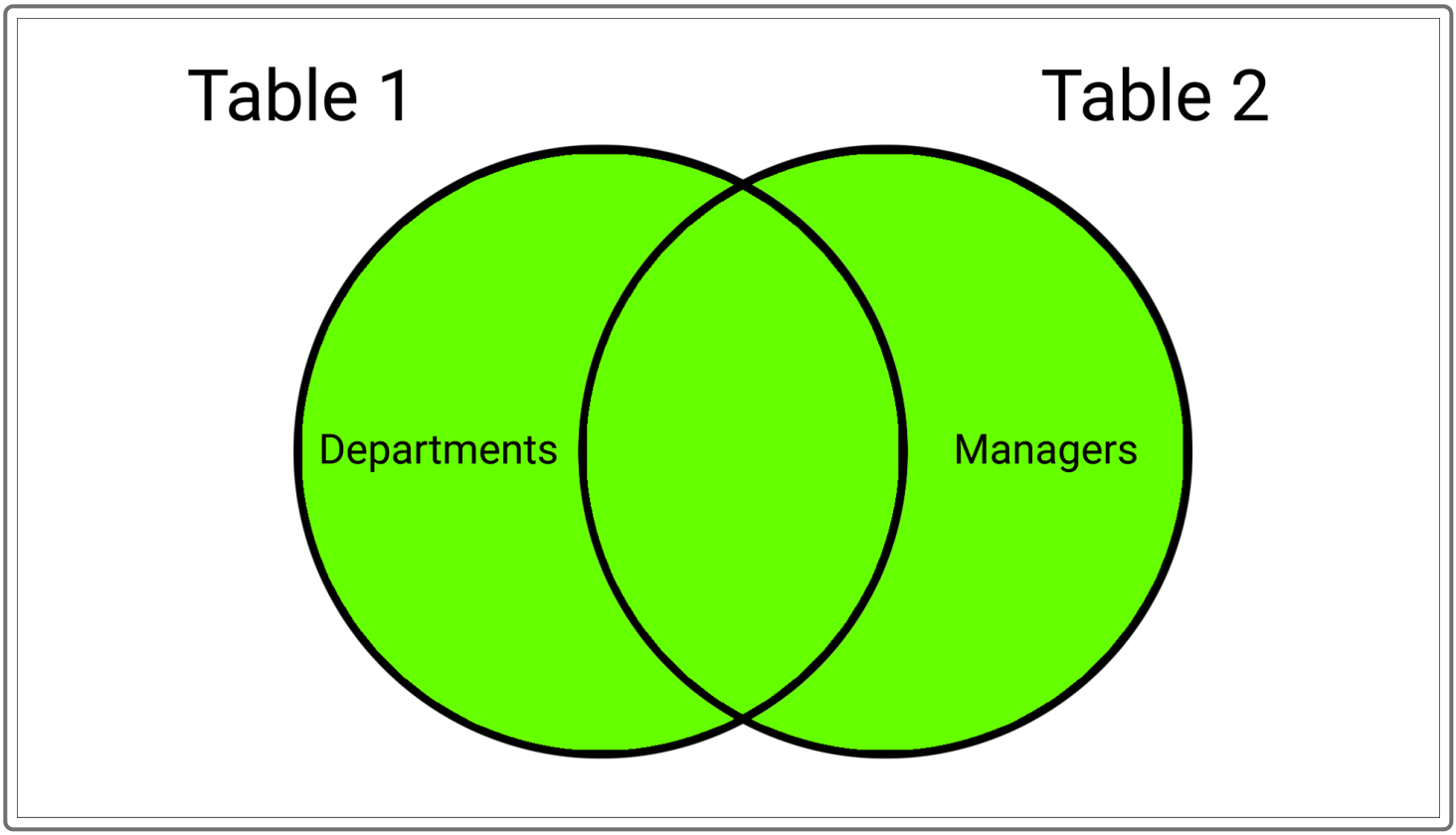


If a manager wasn't assigned to a department, the result would look as follows:

dept_no	dept_name	emp_no	from_date	to_date
NaN	NaN	110183	1/1/85	3/21/92

Full Outer Join

A **full outer join** is a comprehensive join that combines all data from both tables.



The results are potentially massive. This is dangerous for a few reasons:

1. If the data being returned is extremely large, generating it can bog down or even crash your computer.
2. After the query has been returned, navigating the data can also bog down or crash your computer.
3. The data has the potential to be full of null values, or NaNs.

The result of a fully joined table would look like the following:

dept_no	dept_name	emp_no	from_date	to_date
d0001	Marketing	NaN	NaN	NaN
NaN	NaN	110183	1/1/85	3/21/92

IMPORTANT

Use the full outer join with caution!

NOTE

For more information, see the documentation on PostgreSQL joins

(https://www.techonthenet.com/postgresql/joins.php) .