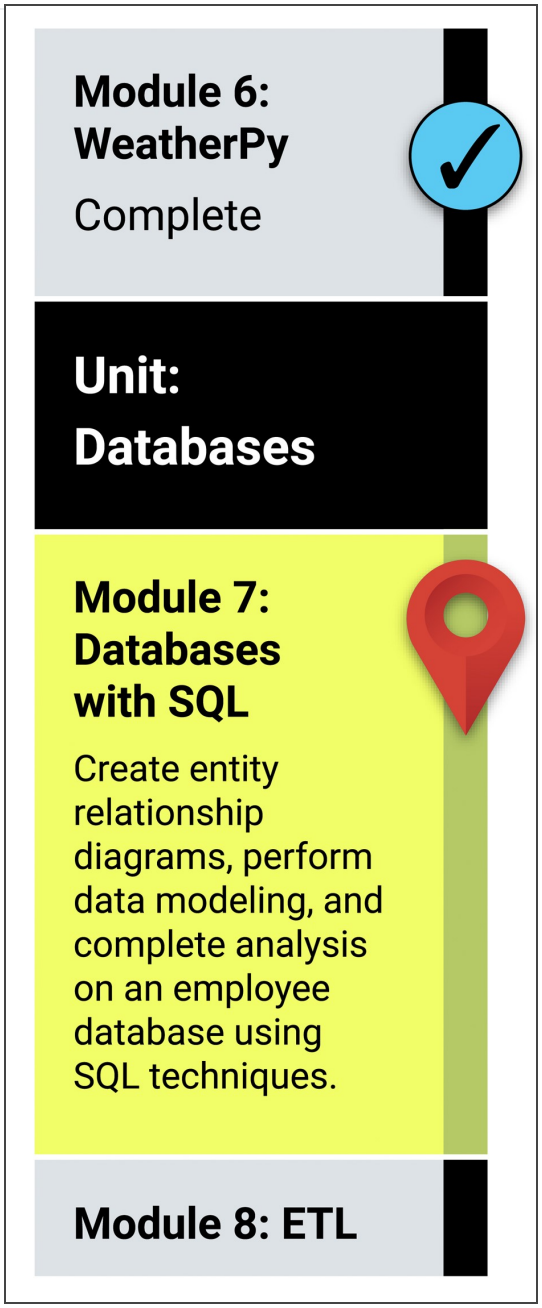
7.0.1: Exploring Databases with SQL

7.0.2: Module 7 Roadmap

Looking Ahead

In this module, you'll learn about data modeling, engineering, and analysis. Applying your knowledge of DataFrames and tabular data, you'll create entity relationship diagrams (ERDs), import data into a database, troubleshoot common errors, and create queries that use data to answer questions. Databases are used everywhere—small and large businesses, and even individuals working on personal projects—and SQL is one of the most widely used query languages in use. Its ability to organize and query data, especially on a large scale, makes SQL knowledge a highly sought after skill in the workforce.



What You Will Learn

By the end of this module, you will be able to:

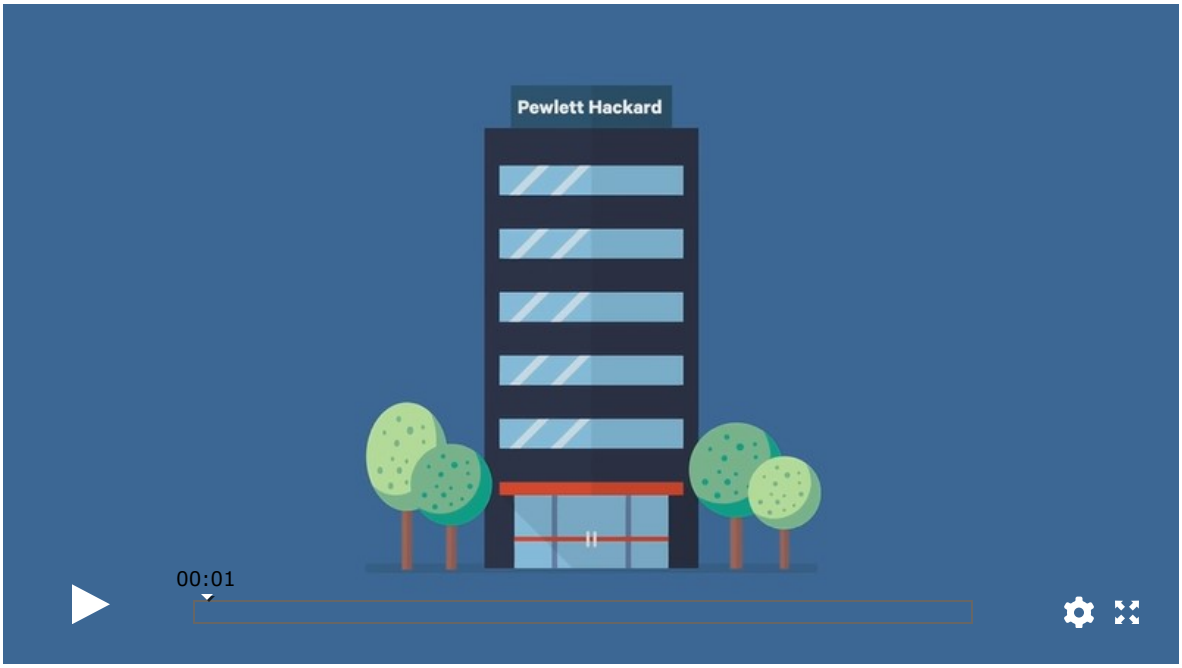
- Design an ERD that will apply to the data.
- Create and use a SQL database.
- Import and export large CSV datasets into pgAdmin.
- Practice using different joins to create new tables in pgAdmin.
- Write basic- to intermediate-level SQL statements.

Planning Your Schedule

Here's a quick look at the lessons and assignments you'll cover in this module. You can use the time estimates to help pace your learning and plan your schedule.

- Introduction to SQL (15 minutes)
- Getting Started with PostgreSQL and pgAdmin (45 minutes)
- Data Modeling (1 hour)
- Data Engineering (3 hours)
- Data Analysis (4 hours)
- Application (5 hours)

7.0.3: Welcome to Pewlett Hackard!



7.1.1: Download and Install Your Tools

Bobby is excited to get started on this new project. PH has fallen a bit behind in the database department, so it will be a huge achievement to get this organized for the company. To help Bobby prepare for his analysis, he'll need to download his tools: PostgreSQL and pgAdmin. He'll use Postgres to create a database, and pgAdmin to work with the data he'll be importing. These tools are packaged together in a single download, so let's get started with the setup!

Much like using Visual Studio Code (VS Code) to create Python programming scripts, SQL requires a code editor with the ability to execute the queries you create. We also need to set up a local database on our own computer. This database is where we'll house all of our data. This way, all of our datasets are centrally located and we won't need to search for them.

PostgreSQL and pgAdmin will be our gateway into the SQL universe.

PostgreSQL

PostgreSQL, typically referred to as just “Postgres,” is a **relational database system**. This type of database consists of tables and their predefined relationships.

Think of it like this: Each CSV file’s data will be loaded into a table. If there are six CSV files, then there will also be six tables in Postgres. “Relationships” are how each table relates to another. We’ll create tables and define relationships as we progress through the module, so don’t worry if this seems confusing right now—we’ll get lots of practice.

Another aspect of Postgres is that it will create a local server on your computer, which is where the databases we create will be stored. Then the databases will store the tables and the data. It’s a rather intricate filing system.

> **Note**
> For more about PostgreSQL, see the **PostgreSQL documentation (https://www.postgresql.org/docs/manuals/)** and the **PostgreSQL tutorial (https://www.tutorialspoint.com/postgresql/)**.

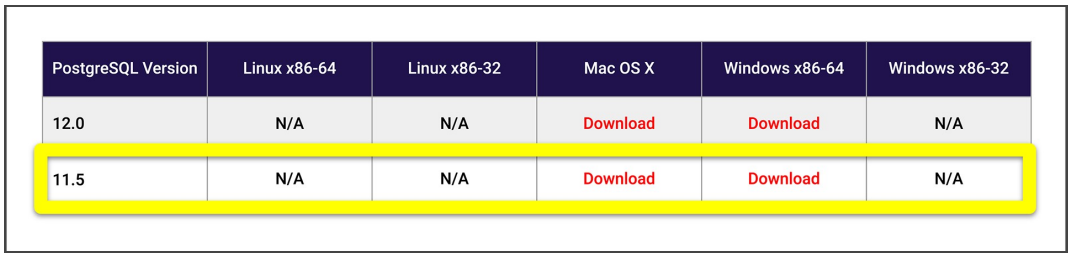
pgAdmin

pgAdmin is the window into our database: it’s where queries are written and executed and where results are viewed. While Postgres holds the files, pgAdmin provides the access. All SQL actions take place within these two programs, so let’s install them.

Installation

Visit the [PostgresSQL download website (https://www.enterprisedb.com/downloads/postgres-postgresql-downloads)](https://www.enterprisedb.com/downloads/postgres-postgresql-downloads) to initiate your download. **Be sure to choose the correct download option for your operating system.** Both Postgres and pgAdmin are downloaded together as a package, so the following installation instructions will cover both.

Be sure to select Postgres version 11.5 to install. We're installing this version instead of 12 because 11.5 is a stable release, meaning that it has been tested and debugged as much as possible and will not generate many errors.



PostgreSQL Version	Linux x86-64	Linux x86-32	Mac OS X	Windows x86-64	Windows x86-32
12.0	N/A	N/A	Download	Download	N/A
11.5	N/A	N/A	Download	Download	N/A

> **Note**
> For more about pgAdmin, see the [pgAdmin documentation (https://www.pgadmin.org/docs/)](https://www.pgadmin.org/docs/).

Follow the installation instructions for your operating system.

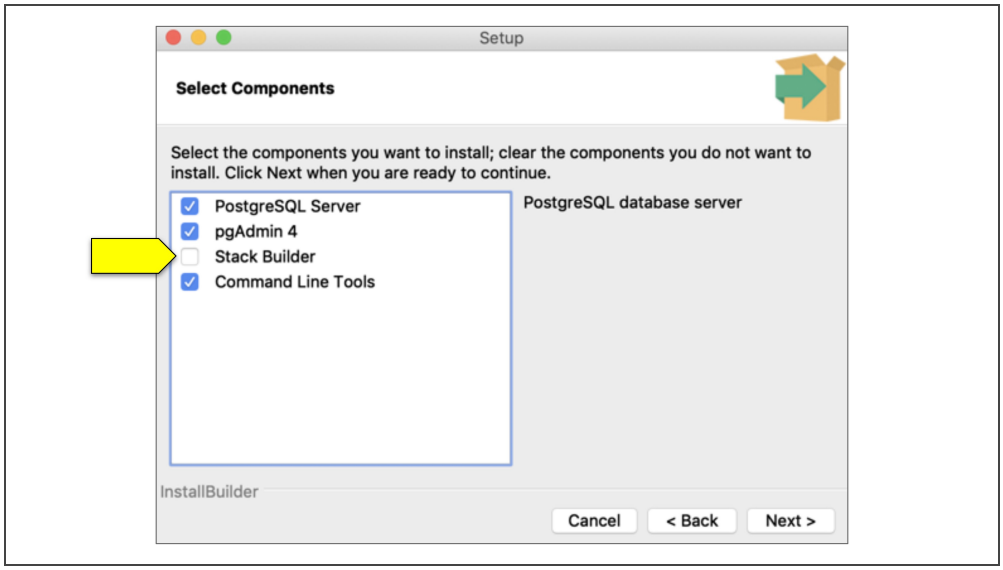
> **Important**
> During installation, you'll need to create a password. **Be sure to record it**, as you'll use it to access your SQL database.

macOS

After downloading PostgreSQL, follow these steps:

1. Double-click the postgresql-11.5-2-osx file.

2. Follow the setup wizard's prompts to begin installation. The software will be installed in /Library/PostgreSQL/11.
3. An InstallBuilder window will show the components selected for installation. Be sure to uncheck Stack Builder's box. Stack Builder is used to install Postgres add-ons, but we won't need it for our project.

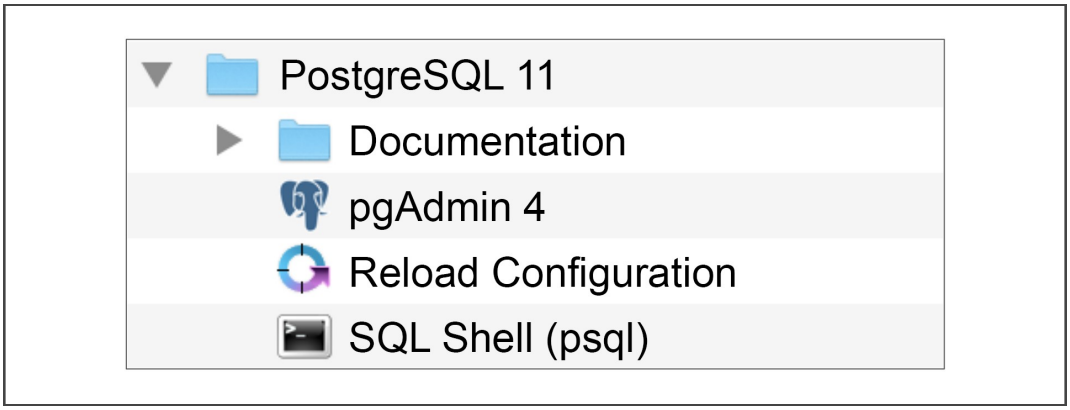


4. Add your data directory to /Library/PostgreSQL/11/data, where data will be loaded and stored.

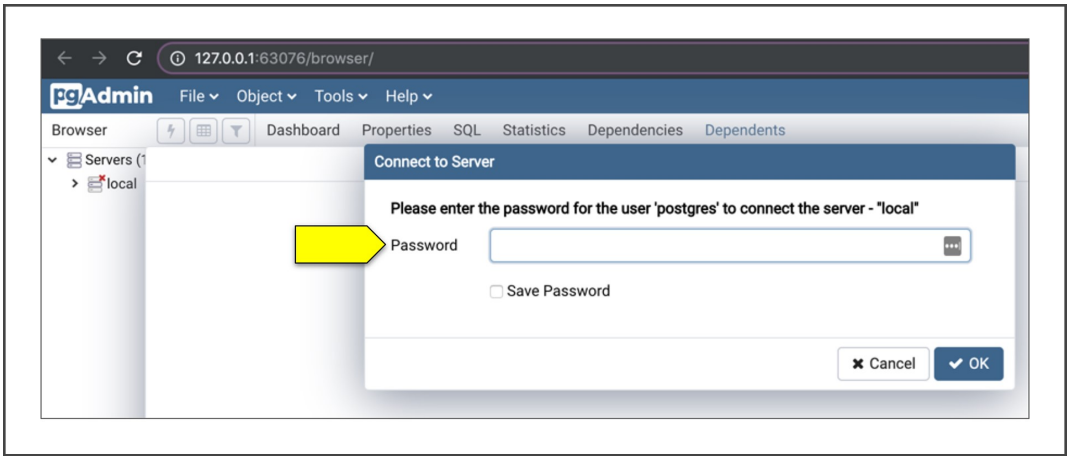
Note: Since you'll be prompted for your password every time you start up Postgres and pgAdmin, it's very important to **record it for future use**.

5. Continue to use the default port 5432. Under Advanced Options, set the locale as "[Default locale]."
6. After reviewing your preinstallation summary, click "Next" to begin the installation.

When the installation is complete, your Mac's Applications section will contain a new folder with the following:



To confirm your installation, start pgAdmin by navigating through your Application section and double-clicking the pgAdmin 4 icon (this will launch a new browser window). Then, double-click to connect to the default server (local) and enter your password.

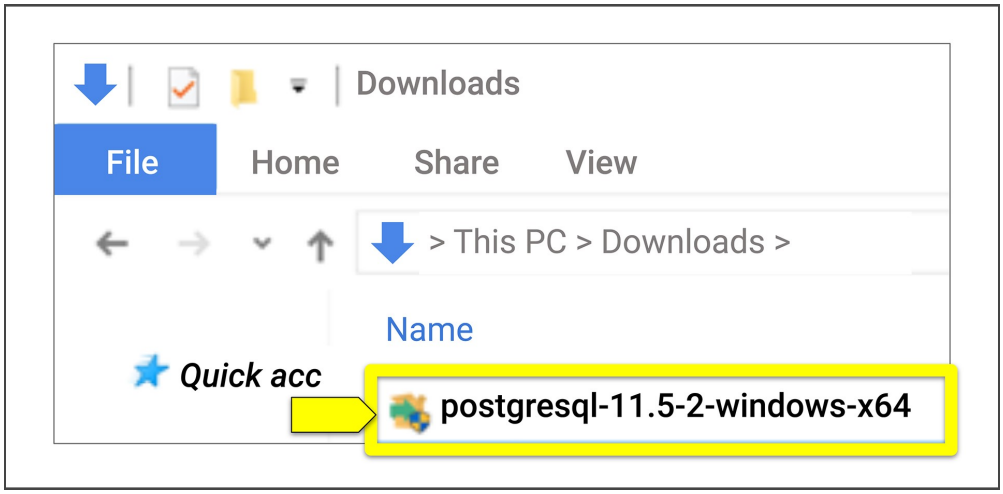


You now have access to your first SQL server.

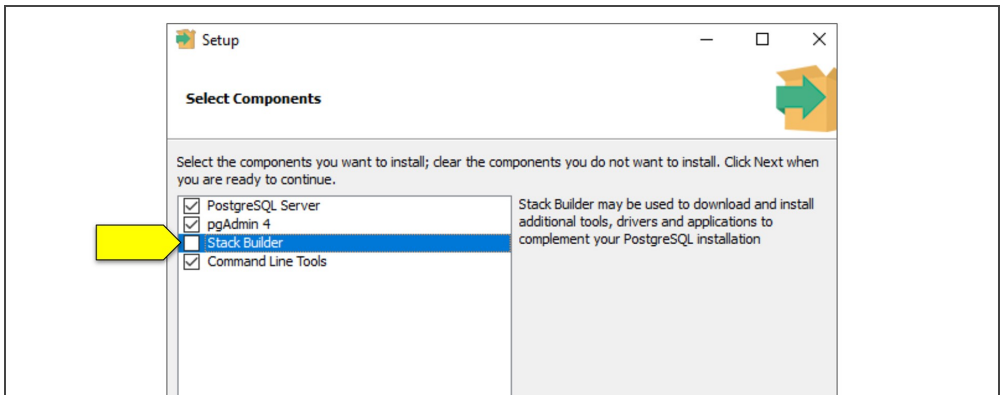
Windows

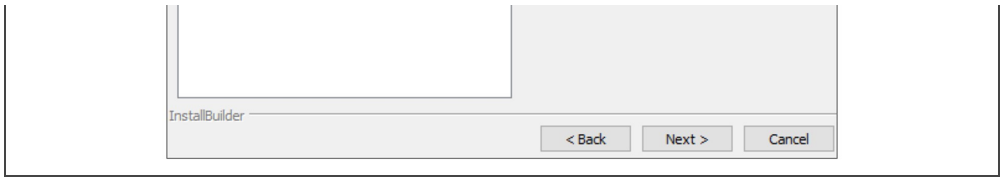
After downloading PostgreSQL, follow these steps.

1. Double-click the postgresql-11.5-2-windows-x64 file.



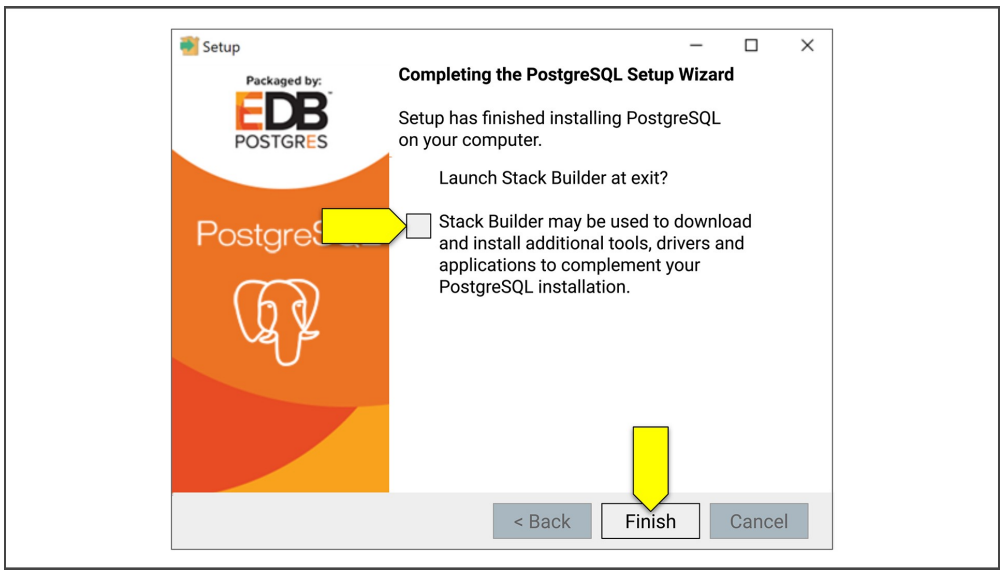
2. Follow the setup wizard's prompts and install PostgreSQL to /Library/PostgreSQL/11, the default location. To find the installation files, first look in the Library folder and then in PostgreSQL.
3. An InstallBuilder window will show the components selected for installation. Be sure to uncheck Stack Builder's box. Stack Builder is used to install Postgres add-ons, but we won't need it for our project.





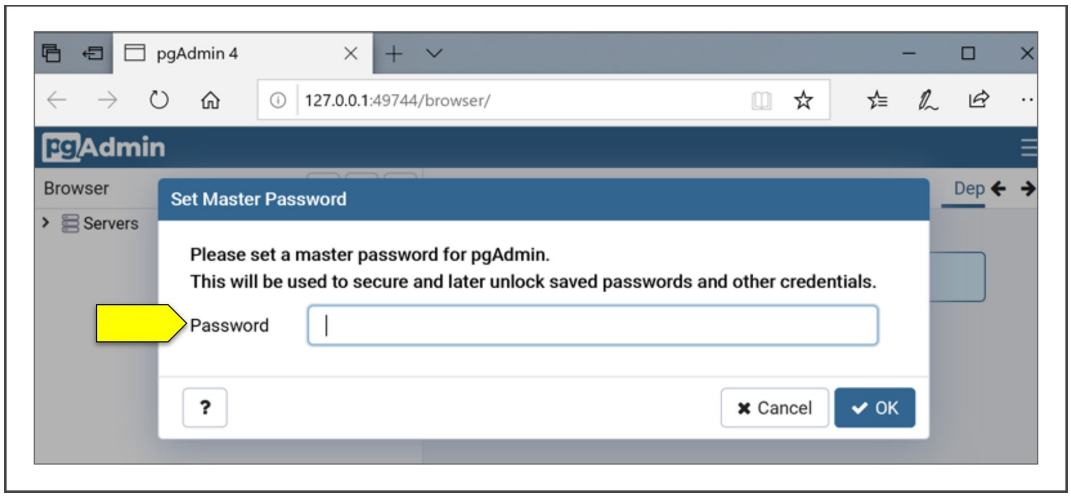
InstallBuilder might not let you uncheck Stack Builder's box. That's okay—it won't harm anything if it is installed.

4. Add your data directory to /Library/PostgreSQL/11/data, where data will be loaded and stored.
5. Continue to use the default port 5432. Under Advanced Options, set the locale as "[Default locale]."
6. After reviewing your preinstallation summary, click "Next" to begin installation (this may take a few minutes).
7. If the final setup screen prompts you to launch Stack Builder at exit, uncheck this box and click "Finish."



8. If prompted, restart your computer to complete the installation.

You should be able to access the Postgres 11 folder from your Start Menu. To confirm your installation, start pgAdmin (a new browser window will launch) and double-click to connect to the default server and enter your password.



Enter your password and log in to access and work with pgAdmin and Postgres.

7.1.2: Identifying Data Relationships

Bobby has all of his tools set up, which is great. The installation wasn't too terrible, and he's ready to start creating databases and importing the data. Except that's a little hasty. A new project is exciting, but let's slow down and take a breath, then look at the data we'll be importing.

By taking a quick look at each CSV, we will have a better understanding of what our data actually looks like. What data types are involved? How many CSV files are there? Is the data all easy to read? By answering these questions early, we'll know if we need to make any adjustments to the data before importing it.

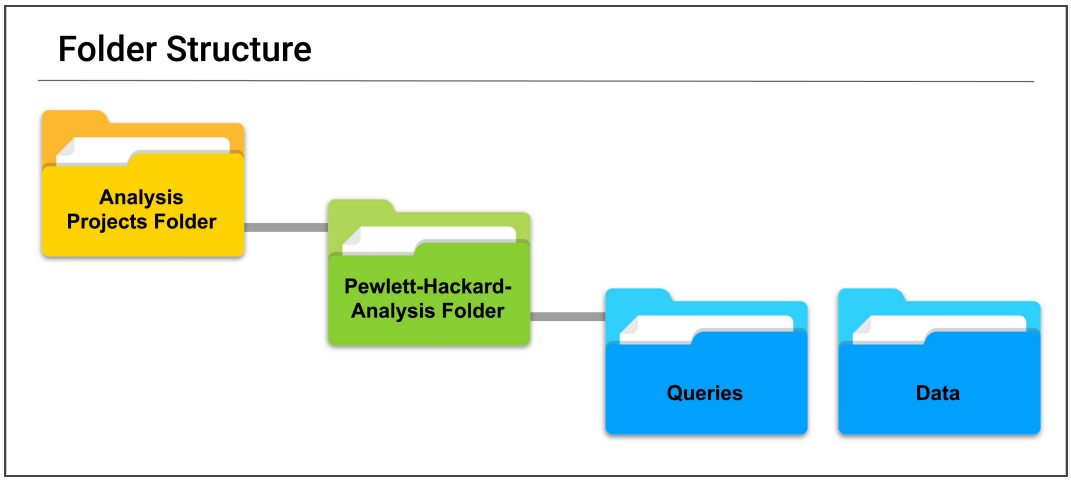
While we're cozying up to the data, let's also look at how the different CSV sheets are connected. Some columns will appear in more than one CSV. We'll take a deeper look into these connections, called primary and foreign keys.

Let's begin the download.

Datasets and Common Columns

Before we can even begin with the actual queries, and before we even load the data into our database, we need to understand what we're looking at. Download our CSV files to take an initial look.

> **GitHub**
>
> Create a new GitHub repository named "Pewlett-Hackard-Analysis." Then navigate to your class folder and clone your new repo within the folder. Set up your folder structure as shown below.



For this module, download the following CSV files and save them in the Data folder. Then commit and push these new files to your repo.

[departments.csv](https://courses.bootcampspot.com/courses/138/files/15023/download?wrap=1)

[dept_emp.csv](https://courses.bootcampspot.com/courses/138/files/15025/download?wrap=1)

[dept_manager.csv](https://courses.bootcampspot.com/courses/138/files/15028/download?wrap=1)

[employees.csv](https://courses.bootcampspot.com/courses/138/files/14679/download?wrap=1)

[salaries.csv](https://courses.bootcampspot.com/courses/138/files/14681/download?wrap=1)

[titles.csv](https://courses.bootcampspot.com/courses/138/files/14726/download?wrap=1)

Now we have six CSVs, each containing different data. Open and review the `departments.csv` file.

dept_no	dept_name
d001	Marketing
d002	Finance
d003	Human Resource
d004	Production
d005	Development
d006	Quality Manager
d007	Sales
d008	Research
d009	Customer Service

There isn't an overwhelming amount of data in this table—only two columns and 10 rows. It's also commonly known as a "lookup table" and

is used to organize data. An example is if we would sort revenue, employee counts, and salaries by department.

Let's look at `dept_emp.csv` next.

emp_no	dept_no	from_date	to_date
10001	d005	6/26/86	1/1/99
10002	d007	8/3/96	1/1/99
10003	d004	12/3/95	1/1/99
10004	d004	12/1/86	1/1/99
10005	d003	9/12/89	1/1/99
10006	d005	8/5/90	1/1/99
10007	d008	2/10/89	1/1/99
10008	d005	3/11/98	7/31/00
10009	d006	2/18/85	1/1/99
10010	d004	11/24/96	6/26/00
10010	d006	6/26/00	1/1/99
10011	d009	1/22/90	11/9/96
10012	d005	12/18/92	1/1/99
10013	d003	10/20/85	1/1/99
10014	d005	12/29/93	1/1/99

There are only four columns of data, but considerably more rows in the spreadsheet. Did you notice the common column, dept_no, shared between `departments.csv` and `dept_emp.csv`?

departments.csv

dept_no	dept_name
d001	Marketing
d002	Finance
d003	Human Resource
d004	Production
d005	Development
d006	Quality Manager
d007	Sales
d008	Research
d009	Customer Service

dept_emp.csv

emp_no	**dept_no**	from_date
10001	d005	6/26/86
10002	d007	8/3/96
10003	d004	12/3/95
10004	d004	12/1/86
10005	d003	9/12/89
10006	d005	8/5/90
10007	d008	2/10/89
10008	d005	3/11/98
10009	d006	2/18/85

Department numbers are listed in both spreadsheets, providing a link between the two. For example, `dept_emp.csv` shows that Employee No. 10009 worked in Department No. 006, and `departments.csv` shows that Department No. 006 is the Quality Management department. We also know Employee No. 10009 joined the Quality Management department on February 18, 1985.

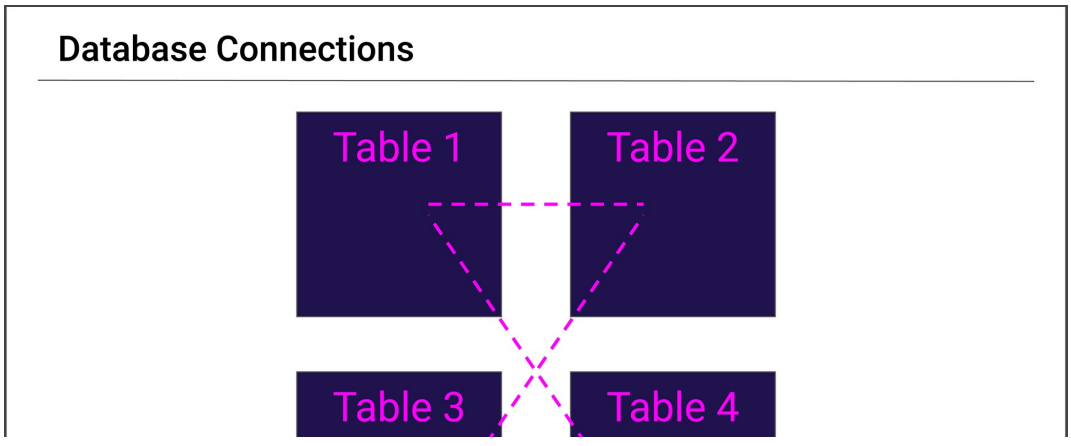
Database Keys

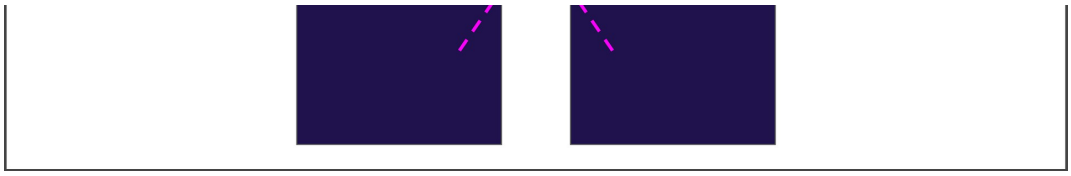
Database keys identify records from tables and establish relationships between tables. There are numerous types of keys. For our purposes, we will focus on primary keys and foreign keys.

Primary Keys

The `departments.csv` file has a dept_no column with unique identifiers for each row (one department number per department). For example, d001 will always reference the Marketing department, across other worksheets. This unique identifier is known as a **primary key**.

Primary keys are an important part of database design. When a database is being created, each table added must include a primary key in the architecture. Primary keys serve as a link between these tables.





In the graphic above, Table 1 has a primary key, or column of unique identifiers in common with Tables 2 and 4. Table 3's primary key is linked only to Table 2. These links trace the relationships between tables. There are times when we'll need to trace two or three links to get the exact data we need. In these cases, we'll pick the data we need from each table. Linking the tables together in this manner is called a **join,** a feature we'll get into later.

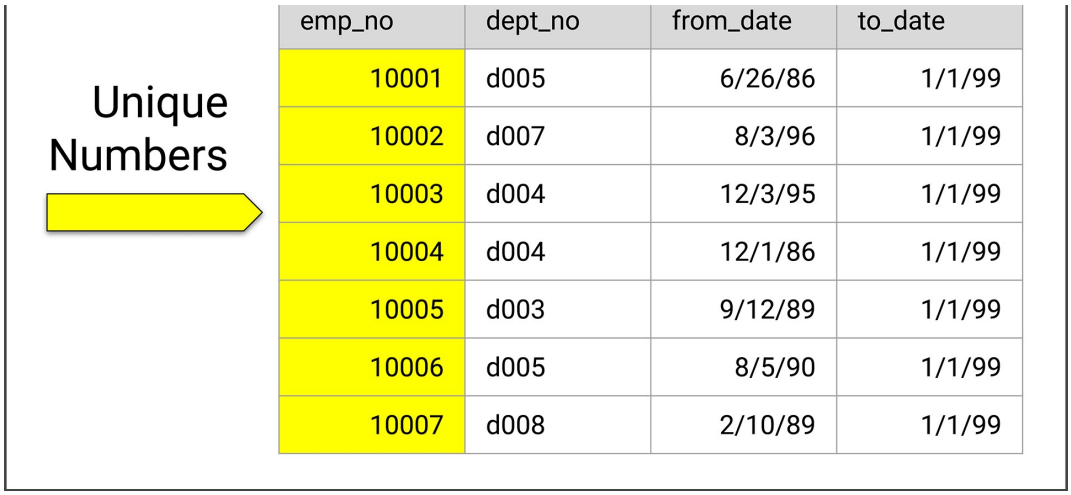
In the second CSV file, `dept_emp.csv`, the "emp_no" column contains the primary key.

dept_emp.csv

emp_no	dept_no	from_date	to_date
10001	d005	6/26/86	1/1/99
10002	d007	8/3/96	1/1/99
10003	d004	12/3/95	1/1/99
10004	d004	12/1/86	1/1/99
10005	d003	9/12/89	1/1/99
10006	d005	8/5/90	1/1/99
10007	d008	2/10/89	1/1/99

We know this is the primary key because each number is unique. For example, the emp_no column holds employee numbers. Each employee will have only one number, and that number won't be used for any other employee.

dept_emp.csv



emp_no	dept_no	from_date	to_date
10001	d005	6/26/86	1/1/99
10002	d007	8/3/96	1/1/99
10003	d004	12/3/95	1/1/99
10004	d004	12/1/86	1/1/99
10005	d003	9/12/89	1/1/99
10006	d005	8/5/90	1/1/99
10007	d008	2/10/89	1/1/99

Open that file and take an initial look at the data.

In which department does Employee No. 106362 work?

- ○ Customer Service
- ○ Development
- ○ Sales
- ○ Research

Check Answer

Finish ▸

Nice work so far! Now test your skills with the following Skill Drill.

Skill Drill

Open the remaining CSVs and identify the primary key and data types in each file. We've already opened `departments.csv` and `dept_emp.csv`, so there are four more to open:

- `dept_manager.csv`

- `employees.csv`
- `salaries.csv`
- `titles.csv`

Foreign Keys

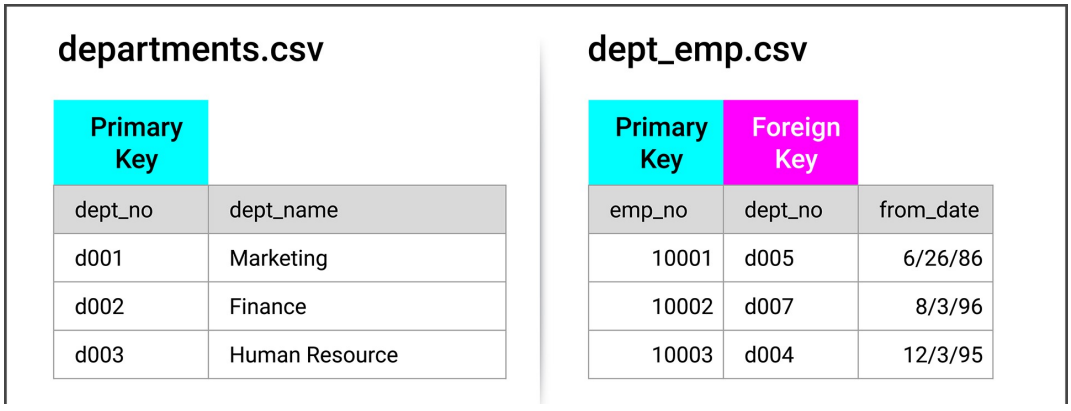
Foreign keys are just as important as primary keys. While primary keys contain unique identifiers for their dataset, a **foreign key** references another dataset's primary key.

Think about it like a phone number. You have your own number. It's your number, assigned to your phone, and unique to you. This is your primary key. Your friend also has a primary key: his or her own phone number.

When you save your friend's number in your phone, you're creating a reference to that person, also known as a foreign key. Your phone has lots of foreign keys (such as parents, doctors offices, friends, and other family), but only one primary key.

Likewise, when your friend saves your number in their phone, your number is now a foreign key in their phone. Saving these keys connects the devices. They show the relationship between your phone and your friend's phone.

Compare our first two CSVs again by looking at the following image.



departments.csv

Primary Key	
dept_no	dept_name
d001	Marketing
d002	Finance
d003	Human Resource

dept_emp.csv

Primary Key	Foreign Key	
emp_no	dept_no	from_date
10001	d005	6/26/86
10002	d007	8/3/96
10003	d004	12/3/95

In this example, dept_no shows up in both datasets; as an identifier (or primary key) in one and as a reference (or foreign key) in the other. This demonstrates the link between employees and which department they work in.

We could continue to look for connections between the datasets, or we could create a roadmap of the content. Our roadmap would serve as a quick reference diagramming the different datasets and their interconnections. Additionally, it could be used as a reference guide later, when we begin to create queries to access all of the data.

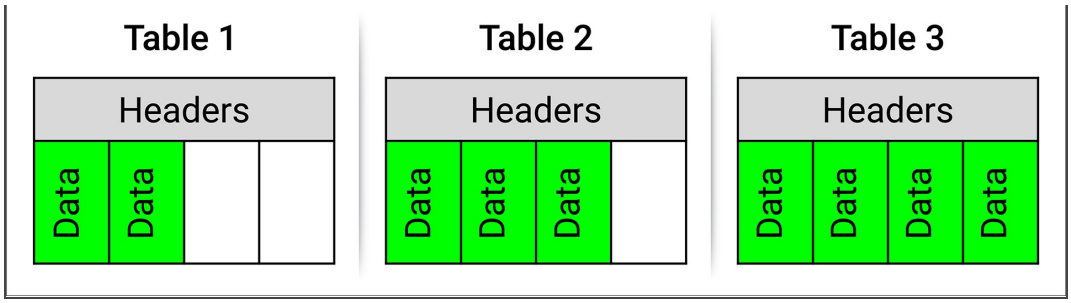
7.1.3: Determine Entity Relationships

So far, we've looked over our data and familiarized ourselves with it, and we have a better grasp on the connections between the data files through primary and foreign keys. It's quite a lot to take in, but we've gotten pretty far—let's keep up the momentum!

Another crucial part of getting a database ready is preparing a solid foundation. We're doing part of that by familiarizing ourselves with the data, but how will it be represented in SQL? SQL organizes data into tables, and each CSV's data will live in its own table: six CSV files will mean six SQL tables. Let's look more into these tables and their properties.

Table Structure

When working in Excel and Visual Basic for Applications (VBA), we're working directly with worksheets with data. In SQL, the same worksheets we have been exploring are organized into tables instead. They are similar to DataFrames in that they have headers and indexes, with data in columns and rows. Take a look at the following images.



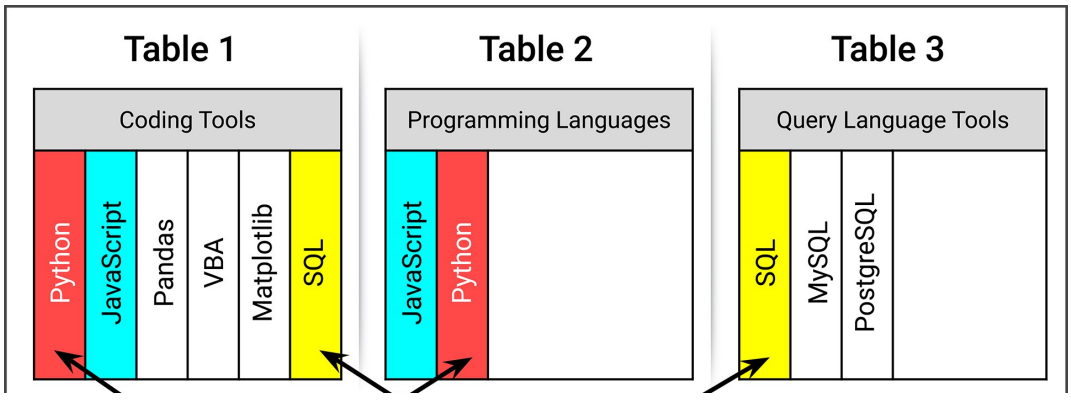
Next we'll cover how table structure comes into play when creating an entity relationship diagram.

Entity Relationship Diagrams (ERDs)

An **entity relationship diagram (ERD)** is a type of flowchart that highlights different tables and their relationships to each other. The ERD does not include any actual data, but it does capture the following pertinent information from each CSV file:

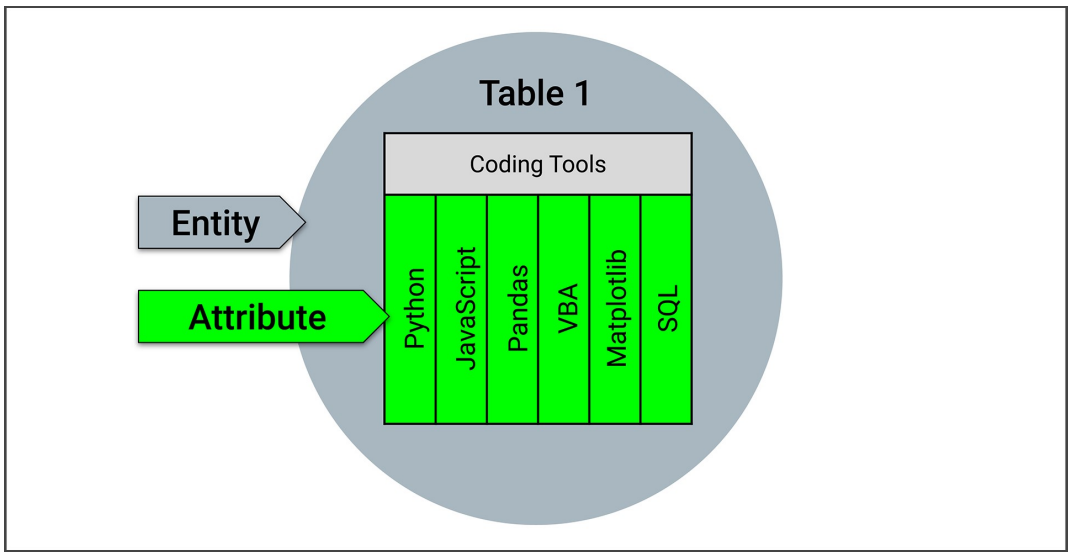
- Primary keys
- Foreign keys
- Data types for each column

The ERD also shows the flow of information from one table to another, as captured in the image below:



In addition to creating new databases, ERDs are used to document existing databases. The visual representation of the tables gives a deeper understanding of the data and the database as a whole.

When creating a diagram, we need to fully understand all of the data being inserted. Database components include tables, known as **entities,** with data, known as **attributes.**



Data types include Booleans, integers, and varying characters (i.e., within a string).

There are three types of ERDs: **conceptual, logical,** and **physical.** Each one builds upon the other—you need the conceptual ERD to build a logical ERD to build a physical ERD. We'll learn how to create ERDs later in this module.

7.1.4: Use the Quick Database Diagrams Tools

Awesome, we've helped Bobby through a large chunk of data modeling! Understanding the data we're working with and finding the connections within is a huge step.

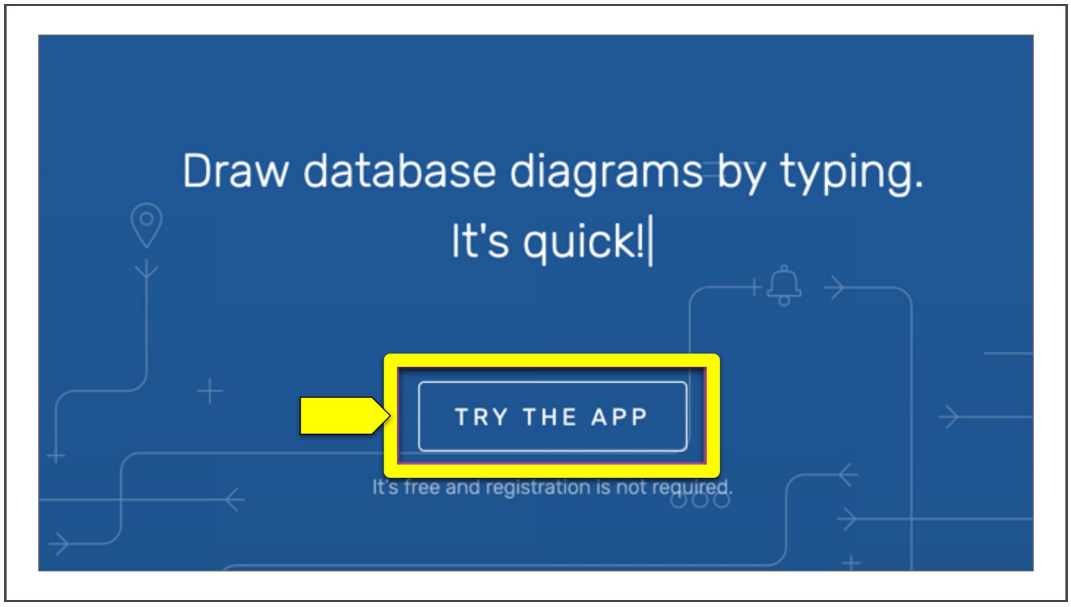
Another step is to create a map of the database. This map will show us each table in the database and the flow of data from one table to another. That's right, we're going to make a fancy flowchart. This provides us with an easy reference to the data without actually accessing it. This is called "modeling the data," and we can get started by creating a diagram with an online tool, instead of trying to make one from scratch. Our flow chart will help us navigate through the relationships more easily than if we had all six CSV files open side-by-side.

Using an online tool called Quick Database Diagrams ("Quick DBD" for short), we'll help Bobby start by familiarizing ourselves with the webpage, then create a conceptual ERD.

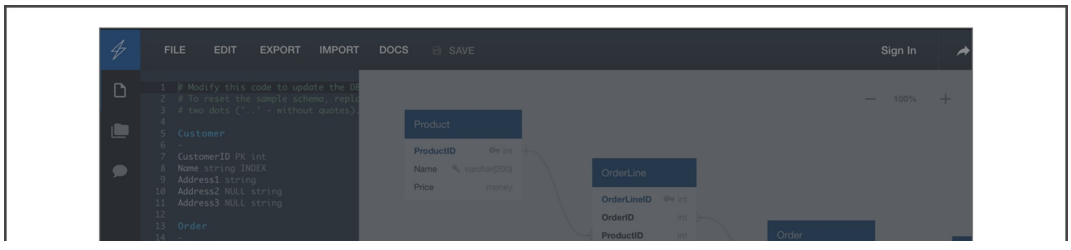
Quick DBD

To create our diagrams, we'll use Quick DBD through [quickdatabasediagrams.com (http://quickdatabasediagrams.com)](http://quickdatabasediagrams.com). Quick DBD is a great resource—it's intuitive and creates clean and comprehensible ERDs that are easily exported as image files.

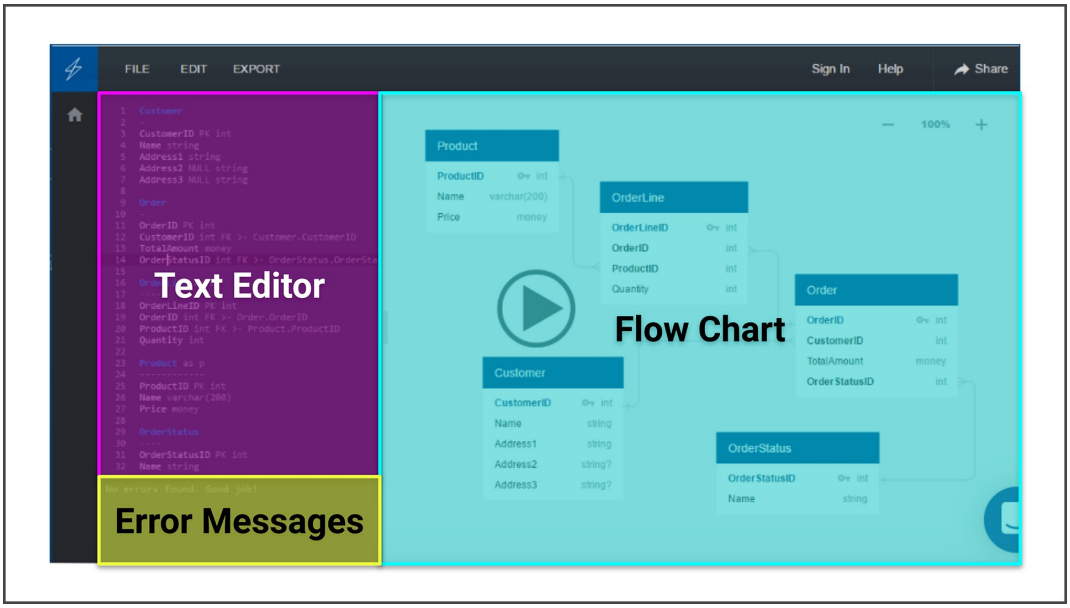
When you visit Quick DBD's site, click the "Try the App" button shown in the following image to get started. You don't need to create an account to make your first diagram.



The next screen is the text editor. Because it's your first visit, the text editor will have a sample ERD already in place, plus an option to tour the app and accept the website's cookies. Click the "Got it!" button to dismiss the message about cookies and go ahead and skip the tour, we're about to dive in!

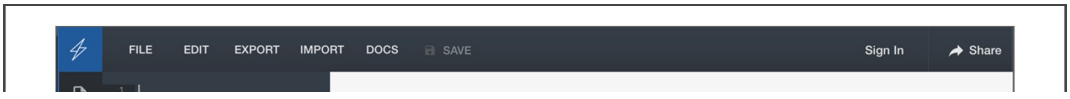


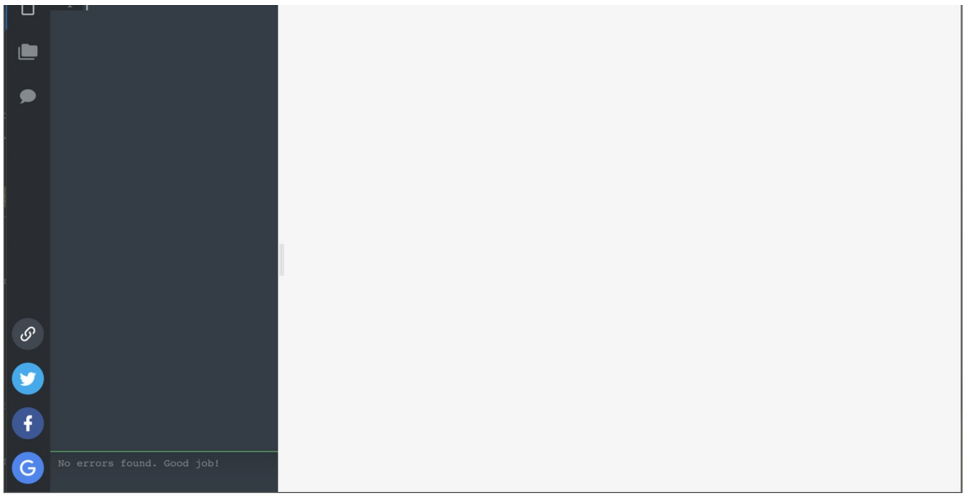
Now you can view our canvas, split into two sections: a text editor and a flowchart workspace. Like the example previously shown, we'll build our tables using the text editor (on the left), and they'll display as a flow chart on the right.



As we build our tables, the flow chart will update, revealing the connections between the tables. Also, if we make a mistake, it quickly becomes apparent: the message box below the text editor will display any errors as they occur.

Instead of altering the sample, we'll go ahead and delete the text in the text editor to clear the canvas for our own use.




No errors found. Good job!

7.1.5: Create ERDs

We are ready to map our data. By now, we're comfortable with what we're working with. We have an idea of the data connections, and we're ready to use our new tool: Quick DBD.

There are several ways to refer to the map we're about to create. It's also called a flowchart, an entity relationship diagram, and a schema. We'll be using all of these terms in this module, though "ERD" is the most specific.

There are three forms of ERDs: conceptual, logical, and physical. We'll start by helping Bobby with the most basic of the three, the conceptual diagram. As we add more information to our tables, such as data types and keys, we'll advance through the more complex diagrams.

Conceptual Diagrams

A **conceptual diagram** is an ERD in its simplest form. To create one, we only need two things: a table name and column headers.

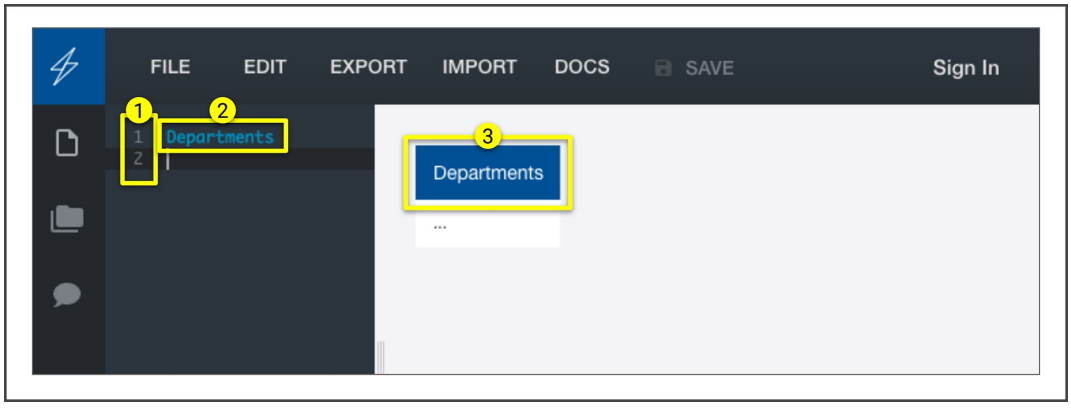
It's simple because we're creating just the *concept* of the diagram. By covering only the basics, it's easier to capture the main points. If we tried to capture everything at once (data types, location of the primary and foreign keys, etc.), we're more likely to overlook a crucial item.

Returning to the Quick DBD website, let's create the blueprint (or schema) for our first table. The schema generates the actual diagram, and we're creating schema instead of coding in the text editor.

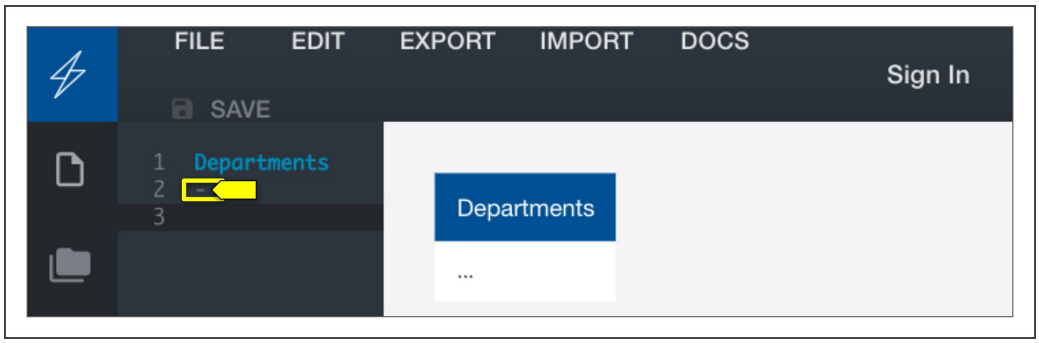
> **IMPORTANT**
>
> **Schema** is a term that will come up often while working in SQL and its extensions (such as MySQL, PostgreSQL, and many others). It references the design of the database, and specifically how the tables and their relationships are mapped out.

We'll use `departments.csv` for the first table, so let's review the worksheet again, as we'll need to know the column names.

The first thing to create in the text editor is the name of the new table. Enter "Departments" on the first line, then hit Enter. Read the following "About Quick Database Diagrams" user notes.

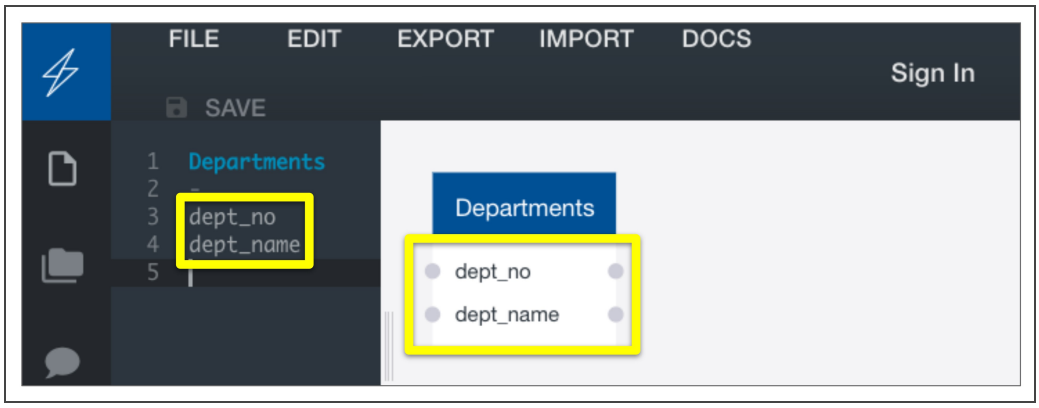


Below the table name, insert a hyphen to signify we're ready to start adding data.



Now add placeholders for your data. Remember, for a conceptual diagram, don't load all of the details. This diagram should only represent the relationships between the tables, so all we need is the name of each column. There are two column headers in `departments.csv`: dept_no and dept_name. Add those to our schema.

On the next blank line in the text editor, input "dept_no" and hit Enter. On the next line, type "dept_name" and hit Enter again. You'll see that the small table has grown a bit!



Did you notice "int" to the right of our table data? That's the default data type assigned by Quick DBD. It's there because we didn't specify what data type each of the headers represent.

What datatype does "int" stand for?

- ○ Intelligence
- ○ Integer

Create a table for the next CSV, `dept_emp.csv` by following these steps:

1. After dept_no, skip one line to leave it blank.
2. On the next line, start the new table by entering its name “Dept_Emp.”
3. On the next line, enter a hyphen to signify a new table, ready for data.
4. On the following lines, enter a column header for each column name.

> **IMPORTANT**
> The syntax is important when creating schema in Quick DBD. When creating tables, make sure to use underscores (_) instead of spaces. A space within a table or column name will generate an error.

Notice the second table has appeared next to the first—this is the beginning of our flow chart.

These tables aren’t locked into place and can be rearranged for a cleaner look. To move a table, left click on the table name and drag to place. It’s a useful feature because as we continue to add tables to the diagram the

screen can become cluttered, and it allows you to organize the tables as you see fit.

The next step in building a diagram is to assign data types, which transitions our conceptual ERD to a logical ERD.

> **Skill Drill**
> Create conceptual diagrams for the remaining CSV files. Remember, we're mapping out the entire database in this one diagram, so we need to include all of the CSV files.

Logical Diagrams

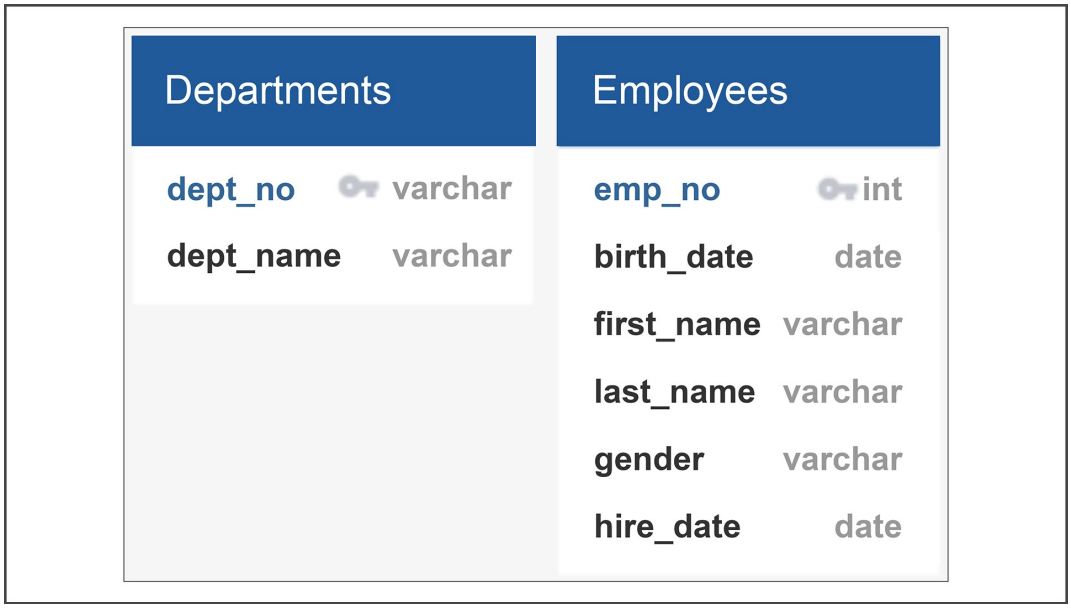
Logical diagrams contain all of the same information that a conceptual diagram does, but the table is updated to include data types and primary keys.

Returning to the Quick DBD webpage, let's update our schema. Because we already took an initial look at the worksheet, we have already identified the primary key and know what type of data we're working with. Using the following syntax, update our Departments schema:

- Add "varchar" to dept_number.
- Add "varchar" to dept_name.

The table updated its data types to reflect the changes in the text editor, and a key symbol appears next to the dept_no line indicating it is the table's primary key.

Update the second table "Employees" the same way, and add “date” as the data type for the dates.



The relationship diagrams are beginning to really take shape. Next, add foreign keys and physically map the relationships between them.

> **Skill Drill**
> Transition the remaining schema from conceptual to logical for the remaining tables and rearrange them to all fit on the screen.

Physical Diagrams

Physical diagrams portray the physical relationship, or how the data is connected, between each table. There are several different relationships available to keep in mind when making these connections, as shown below:

```
-      - one TO one
-<     - one TO many
>-     - many TO one
>-<    - many TO many
-0     - one TO zero or one
0-     - zero or one TO one
0-0    - zero or one TO zero or one
-0<    - one TO zero or many
>0-    - zero or many TO one
```

The relationships are fairly self-explanatory, too. For example, a **one-to-one** relationship means that a primary key is only referenced in one other table. A **one-to-many** relationship, on the other hand, means that a primary key is referenced in two or more tables within the database.

> **Note**
>
> For more information about entity relationships, see the [Entity Relationships Tutorial (https://www.entityframeworktutorial.net/entity-relationships.aspx)](https://www.entityframeworktutorial.net/entity-relationships.aspx). There are more relationships than listed in the table provided by Quick DBD.

Practicing Using Entity Relationship Diagrams

One benefit to building out each form of the diagram (conceptual, logical, and physical) is that it gives us more exposure to the data and its layout. Working with it often leads to more familiarity, which in turn helps with creating a map of the relationships. Even with the exposure and familiarity, it can still be confusing, so let's keep practicing.

Take a look at the Managers table in our text editor. We already know that the dept_no is also the primary key for the Departments table, and because it's referencing another table, the dept_no in the Managers table is a foreign key as well as a primary key. Add that designation to the schema by adding "fk" right after "pk" in that schema.

Now that we have the foreign key identified, we can create a one-to-one relationship between the Managers and Departments tables.

What is the symbol for a one-to-one relationship?

- ○ --
- ○ -
- ○ ->
- ○ >-

Check Answer

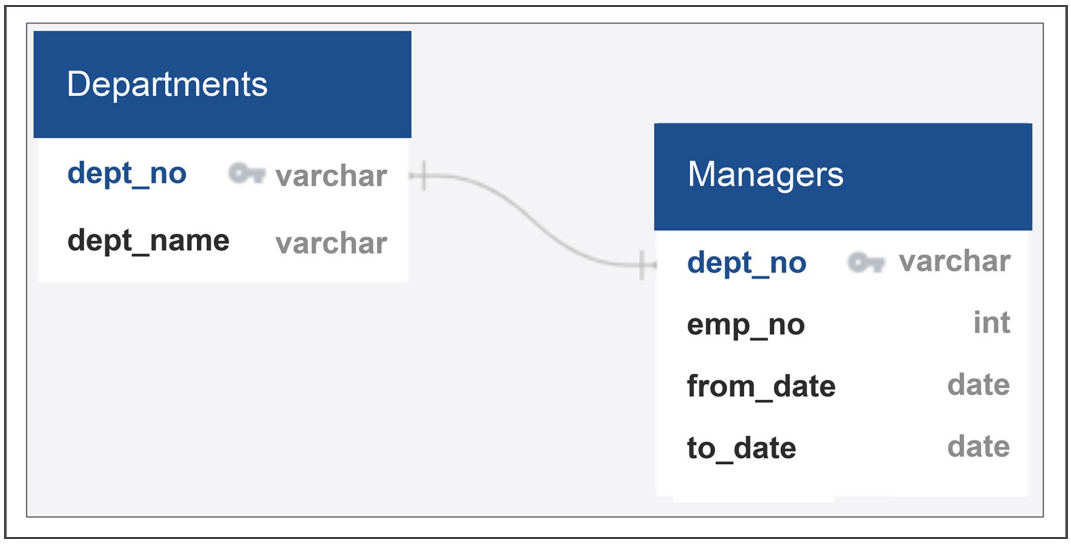
Finish ▸

To add the relationship between tables, place a hyphen (-) after "fk," then a reference to the other table. The reference syntax is

“tablename.column_name.” To reference the Departments table, the line would be updated to look like this:

```
15    Managers
16    -
17    dept_no varchar pk fk - Departments.dept_no
18    emp_no int
19    from_date date
20    to_date date
```

There aren’t any additional updates needed for the Departments table to make this connection. Quick DBD automatically maps the connection for us.



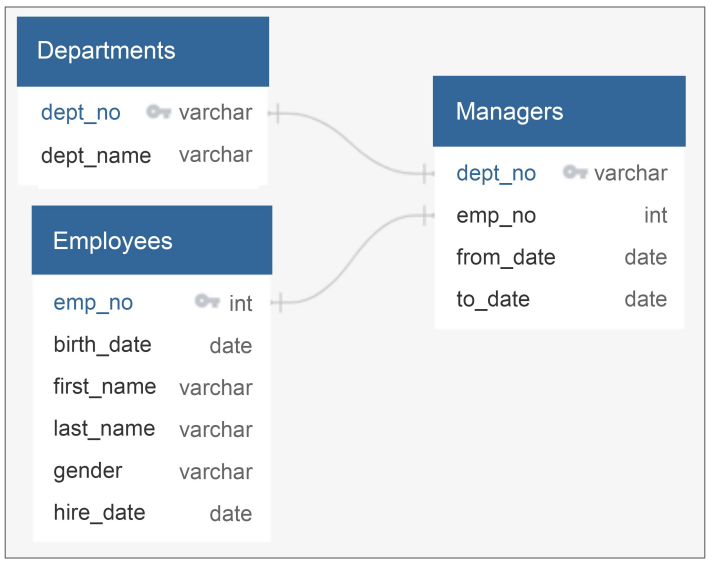
If you rearrange the tables, the connector line will reposition with the tables. This way, after all of the connections are mapped, we can rearrange the tables until the final flowchart is smooth and uncluttered.

Let’s make another connection. The Managers table has another foreign key: the emp_no. The employee number is referenced in the Employees table as well, so we can designate this line as a foreign key. Next, we’ll

add the one-to-one relationship, and then reference the Employees table on the same line.

```
15    Managers
16    -
17    dept_no varchar pk fk - Departments.dept_no
18    emp_no int fk - Employees.emp_no
19    from_date date
20    to_date date
```

Once again, Quick DBD has picked up the relationship and drawn a connection without additional updates to other table schema.



Now test your skills with the following Skill Drill.

Skill Drill

Complete the following relationships:

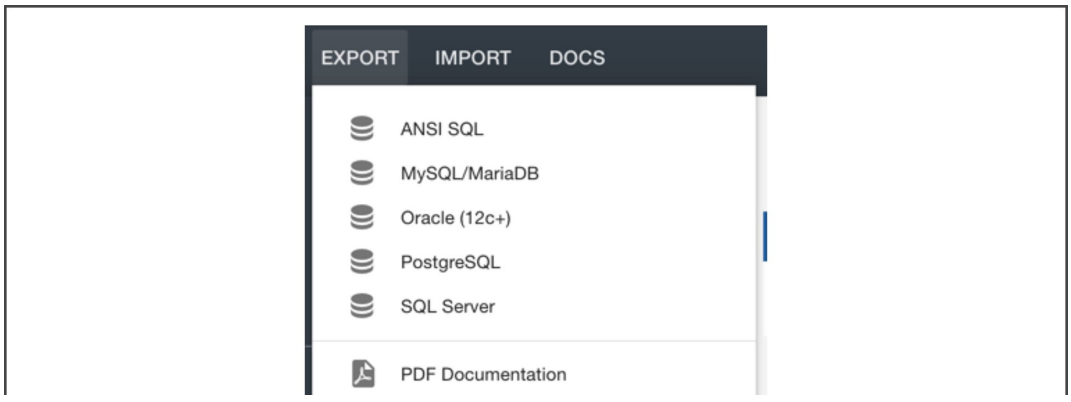
1. In the Dept_Emp schema, create a one-to-many relationship to both of the Employees and Departments tables.

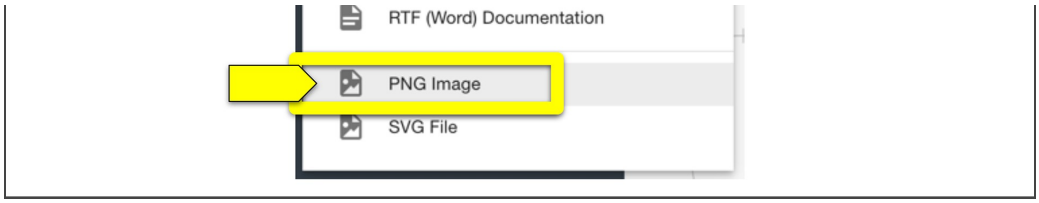
2. In the Titles schema, create a one-to-many relationship to the Employees table.
3. In the Salaries schema, create a one-to-one relationship to the Employees table.

Your final flowchart should have the same connections, as shown below, though the layout may not match exactly.

Now that the database roadmap is complete, you'll need to save your diagram as an image and schema as a text file.

To save your flowchart, click the "Export" button at the top of the Quick DBD window, then click "PNG Image" to save.



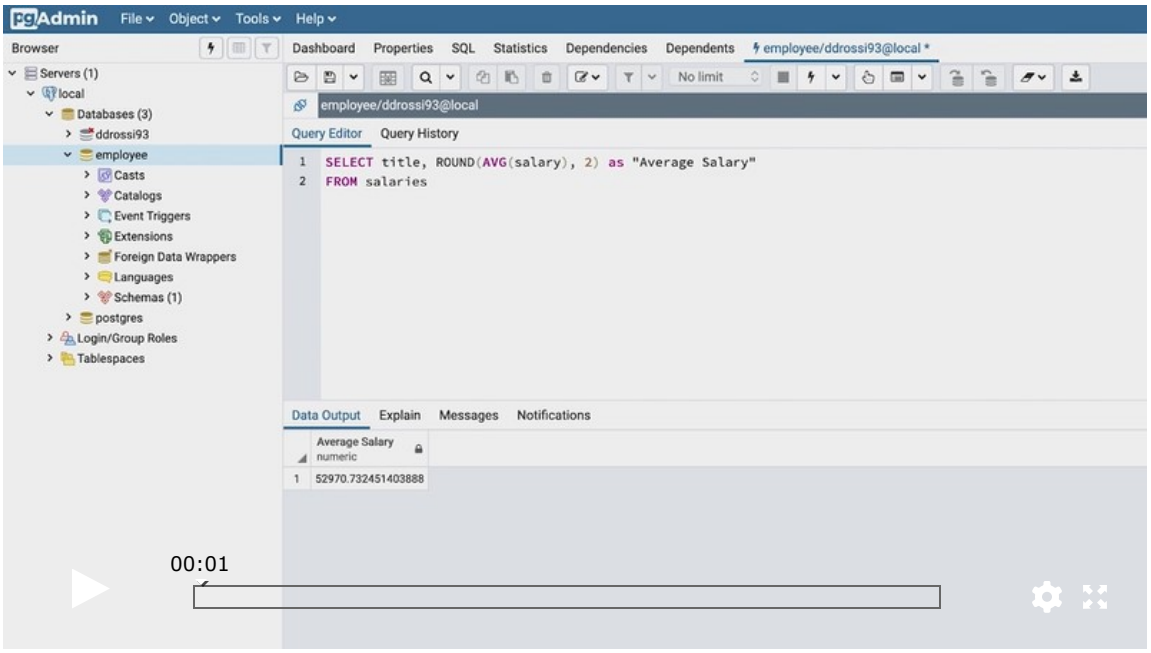


The image will be automatically saved in your Downloads folder as `QuickDBD-export.png`. Move your image to your Pewlett-Hackard-Analysis folder and rename it `EmployeeDB.png`.

Now we have a complete entity relationship diagram to reference as we help Bobby work through the analysis.

Next, we need to save the text we've written to create the schema. The free version of Quick DBD doesn't permit saving online without paying for that space. Instead, we'll save our text locally. By keeping a local copy of the schema, we can access it whenever we want. If we need to update a table, we can open the saved schema instead of having to recreate it from scratch.

The following video walks you through the process of saving the Quick DBD schema text.



Because we've already saved a PNG, we can use the image to reference our ERD from now on; feel free to bookmark and close the website before moving on.

Nice work! Now you're ready to create a database.

> ### Add, Commit, Push
> With these new files saved in our repo folder, we can add them to our repo with `git add`, and then push them up with `git push`.

7.2.1: Create a Database

You've created an entire map of a database! That's no small feat—a lot of thought goes into creating an ERD. We've already put in quite a bit of effort to create a really solid database foundation. Now that it's finished, we can help Bobby create a database in Postgres. Remember, Postgres is where our data will actually live once we import it.

Also, remember how we've been calling the ERD a map? This is where we'll really be able to implement its uses as such. Using the ERD, we will be able to create each table for each CSV. These tables will be tailored specifically for the data in each file, and their creation has already been planned out in our ERD. All we need to do is plug in the information!

We'll start by opening pgAdmin and familiarizing ourselves with the GUI (graphical user interface). A GUI is an interface that often uses visual indicators to help users, such as us, navigate through a program. We'll use the GUI to create our database, then connect to it.

Launch pgAdmin

When writing queries in SQL, data is organized into tables, as shown in the ERD. Return to the pgAdmin window we opened earlier.

Hint

If you have closed your pgAdmin window, or shut down the program completely, you can open a new one by locating the pgAdmin icon and clicking it to start the software again. Then, follow the steps for your operating system.

macOS

1. Open your launchpad and find the pgAdmin 4 icon.

2. Click it to start the program.

   If a new window does not automatically open, then click the icon from the toolbar and select “New pgAdmin 4 window.”

Windows

> Search for pgAdmin or double-click the shortcut on your desktop to start the program.

Connect to the Server

If you've been disconnected from your server, locate it in the menu to the left, then single-click the local server to initiate a connection. At this point, you will be prompted to enter the password you created during installation.

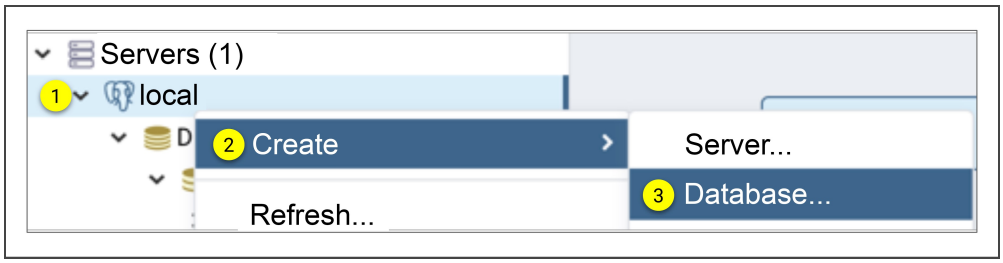
After connecting to the server, you should see that there is already a database named "postgres."

This is the default database that is created when the pgAdmin and Postgres package was installed. Instead of using this database, though, we'll be creating one specifically for Bobby's work with Pewlett Hackard.

Create a New Database

To create a new database to hold the employee information, follow these steps:

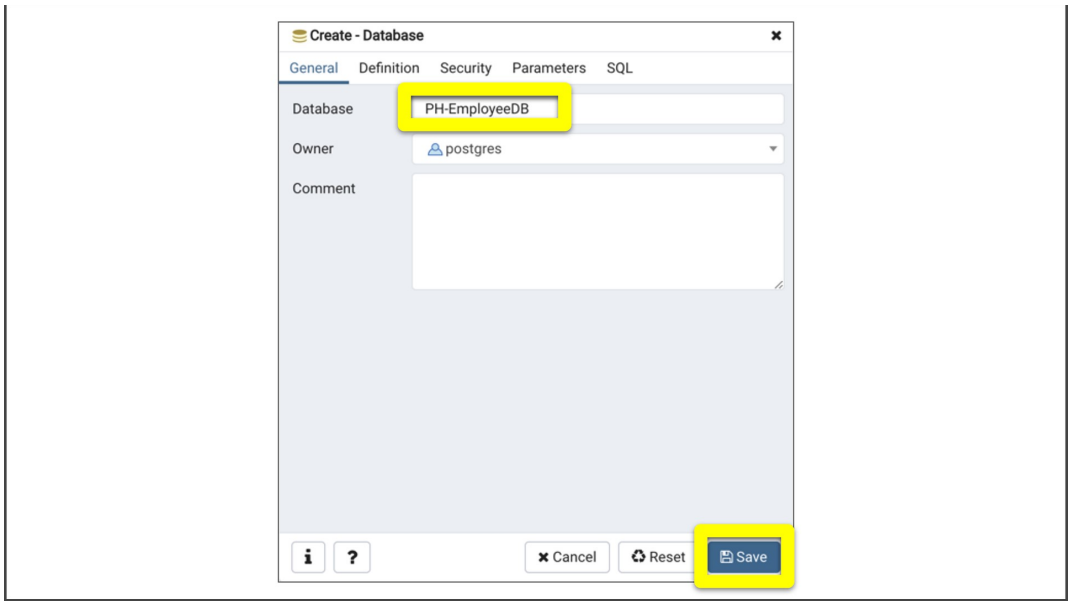
1. Right-click on “local.”
2. Hover your pointer over “Create.”
3. From the menu that pops up on the right, click “Database.”



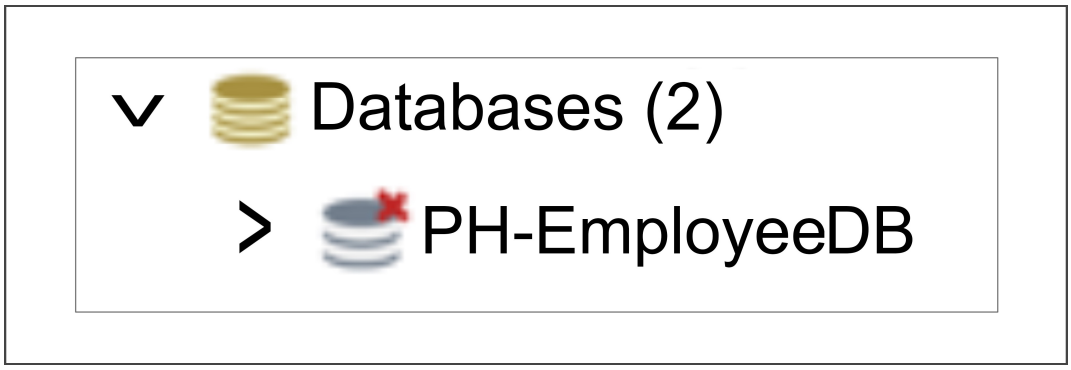
After following these steps, a form will pop up prompting for information. We’ll want to name the database something relevant to the data it will house, such as “PH-EmployeeDB.” The owner is set to Postgres by default, which is fine for our purposes.

> **Hint**
>
> It’s okay to assign your own name as the owner also—this is your work.
>
> Later, as you expand your SQL skills and work with more complex databases, the owner is often named after the role of the SQL developer(s) working in it. This is to easily designate who has been working in the database.

Click “Save” to create a new database.



After the new database has been created, the database count in pgAdmin will have increased to two, and the new database will be listed.



A red X beside the new database's name indicates we aren't yet connected to it, but it is there and ready for use. Click on the new database to connect. Once connected, we'll be able to create tables and import data.

7.2.2: Create Tables in SQL

We've helped Bobby get more comfortable with pgAdmin, and now we have a database ready to go. Good thing we've already completed all of the other prep work with the data—we're ready to begin creating tables!

Using our final ERD as a guide, we'll create six tables, one for each CSV file. During the creation, we'll even map out the primary and foreign keys and assign data types. This will all be completed using pgAdmin's query editor, which is a code editor for SQL, much like VS Code is for Python.

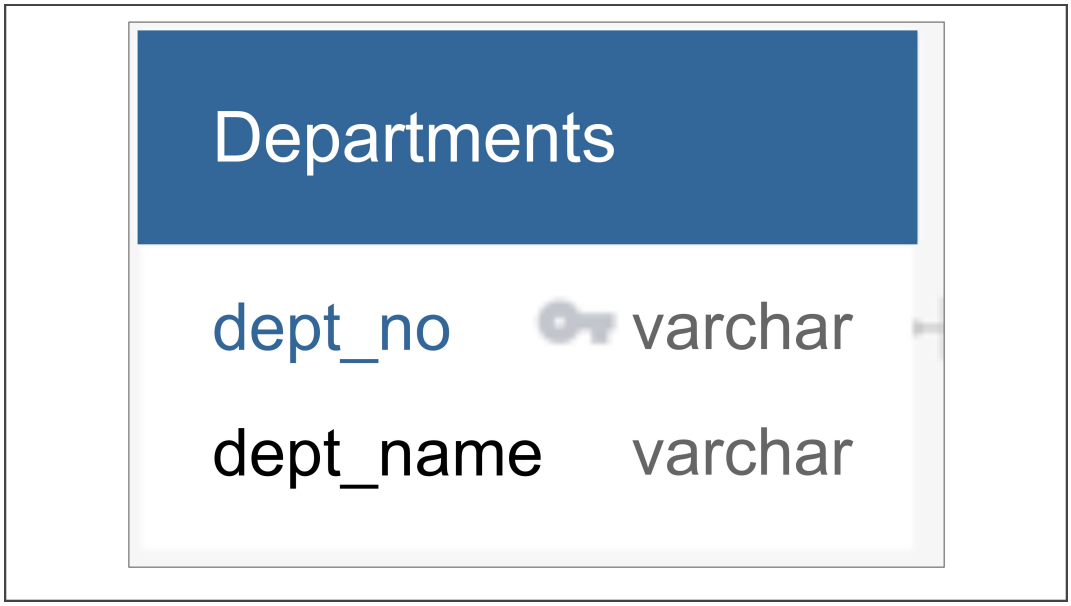
These table creation statements will be our first introduction to Structured Query Language. A statement is a block of code that, when executed, sends a command to the database. Let's get started.

Review Diagrams

We have a database ready to go, so now we need to prep it for data. Take a quick look at our diagram to see what we need for the next step: recreating the same tables in the diagram in our SQL database.

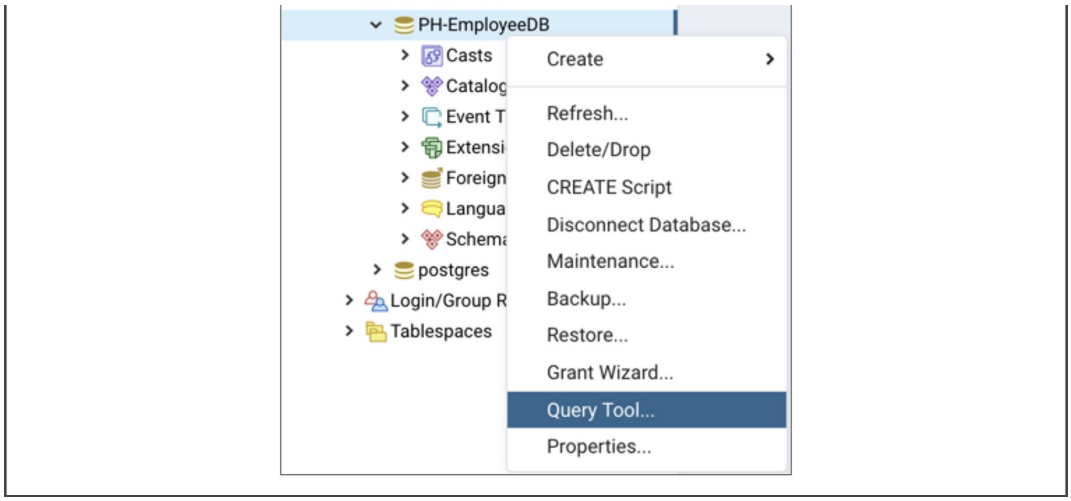
Remember all the thought and care we put into creating the ERD? We took the extra time then because it's easier to iron out any mistakes or fix an error in a tool like Quick DBD than it is to recreate a table in Postgres.

Take a closer look at the Departments table from the diagram.



With the help of the diagram, we know the structure of this table: two columns with their data types. Also, the table is already named. All we need to do is transfer over the same information.

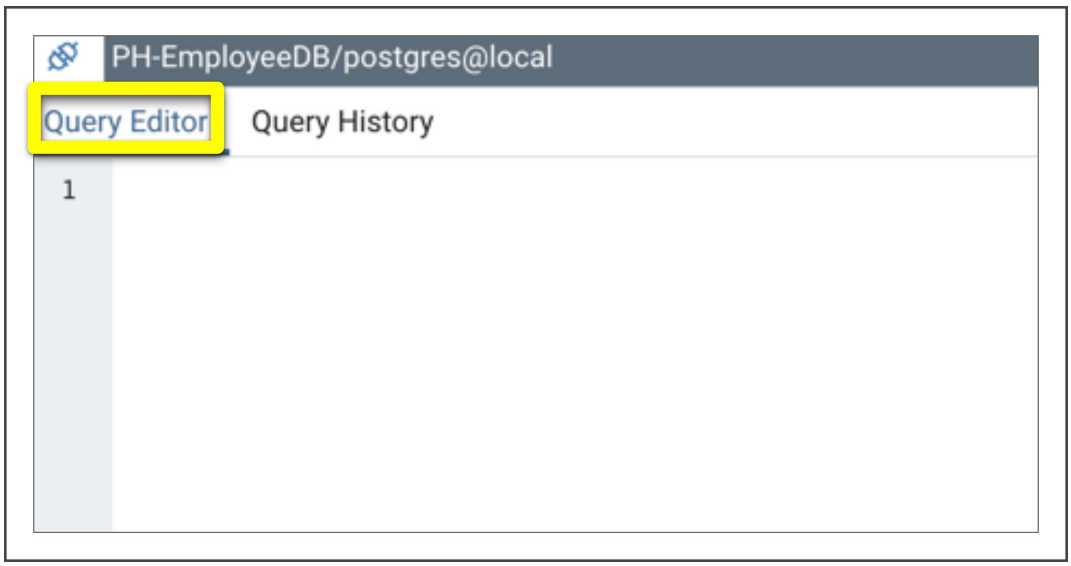
Looking back at the pgAdmin window, right-click on the database PH-EmployeeDB. Then, from the dropdown menu, scroll down to the Query Tool and click to select.



The Query Tool is pgAdmin's text editor, much like VSCode is for Python. This is where we will be helping Bobby construct all of the queries we'll need to perform analysis on the employee database.

With a brand new database, we can't just dive right in. There isn't any data available to query.

After opening the Query Tool, a query editor will appear in the pgAdmin window.



To create our new table, type the following code in the query editor:

```
-- Creating tables for PH-EmployeeDB
CREATE TABLE departments (
     dept_no VARCHAR(4) NOT NULL,
     dept_name VARCHAR(40) NOT NULL,
     PRIMARY KEY (dept_no),
     UNIQUE (dept_name)
);
```

Now we'll break down the code.

Write Statements to Create Tables

The very first line begins with a comment that clearly states the reason for the code. A SQL comment begins with two hyphens with no space between them: (--). Clear and concise comments add to the readability of any coding project for yourself or any other developer who looks at it later. This one comment will apply to all of the `CREATE TABLE` statements, not just the one we are currently building.

Notice we're only adding one comment to capture all of the tables instead of one comment for each table. This is because these types of comments are a bit redundant; the code is already telling us what it's doing and doesn't need excess explanation.

> **Hint**
> There is an ongoing discussion within the coding community regarding comments. One side of the argument is that comments aren't necessary—clean code is. However, there are many strong cases for adding comments.

1. Many employers appreciate code that clearly states what the code is for. Some companies even have developer style guides on writing comments.
2. Comments are often used as placeholders for future code. This helps developers keep track of their work.
3. They are also a way to add copyright and ownership information to proprietary code.

There are many other reasons why comments are important, so getting used to strategically adding clean and clear comments is a good thing to start practicing early.

Break down the three main components of our `CREATE TABLE` statement.

```
  -- Creating tables for PH-EmployeeDB
1 CREATE TABLE 2 departments (
  3 dept_no VARCHAR(4) NOT NULL,
```

1. `CREATE TABLE` is the syntax required to create a new table in SQL.
2. `departments` is the name of the table and how it will be referenced in queries.

   So the table has been named, now the structure needs to be created. The content inside the parentheses is how we'll do that.

3. `dept_no VARCHAR(4) NOT NULL,` creates a column named "dept_no" that can hold up to four varying characters, while `NOT NULL` tells SQL that no null fields will be allowed when importing data.

There are times when we don't want a data field to be null. For example, the dept_no column is our primary key—each row has a unique number associated with it. If we didn't have the `NOT NULL` constraint, then there's a chance that a row (or more than one row) won't have a primary key associated with the data.

What do you think would happen if one of the rows didn't have a unique identifier?

- ○ Nothing, all of the queries would still work correctly.
- ○ None of the queries would work.
- ○ All of the queries would work, but the data might not be included. That's okay though, because what's one line of data?
- ○ Not all of the data would be present in every query, which would skew analysis results and provide incomplete lists.

Check Answer

Finish ▸

Let's continue reviewing our code.

```
4 dept_no VARCHAR(40) NOT NULL,
5 PRIMARY KEY (dept_no),
6 UNIQUE (dept_name)
);
```

4. `dept_name VARCHAR(40) NOT NULL,` creates a column similar to the dept_no, only the varying character count has a maximum of 40.

5. `PRIMARY KEY (dept_no),` means that the dept_no column is used as the primary key for this table.
6. `UNIQUE (dept_name)` adds the unique constraint to the dept_name column.

IMPORTANT

A constraint is a rule that is applied to a column in a SQL table. It will limit the data in a way that provides more accuracy and reliability when working with it.

The unique constraint implies that the data in that column is unique. This ensures that if the table were to be updated in the future, nothing will be duplicated.

Take the table holding all of the department names and numbers, for example. Say that a developer wanted to update that table to add a new department, but reused department number d007.

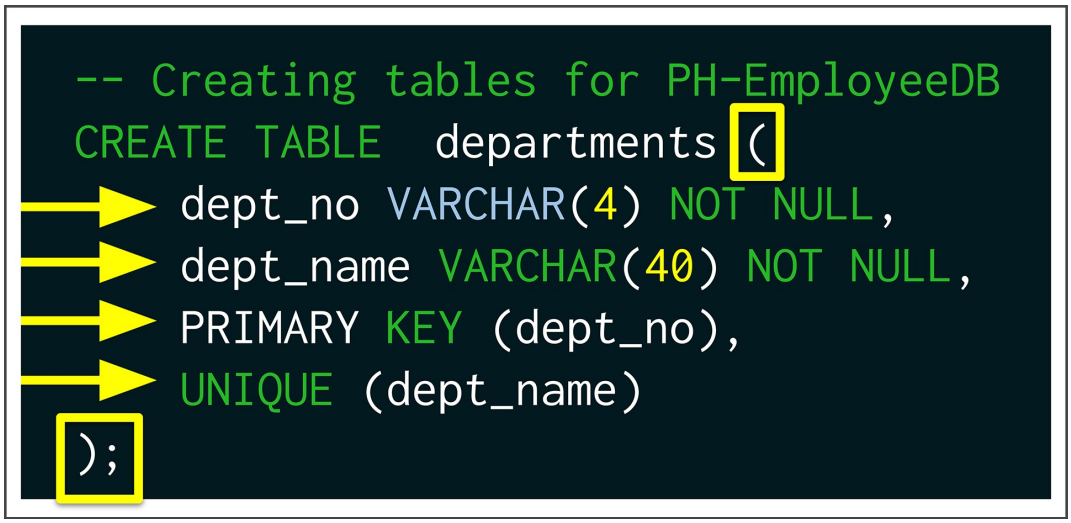
If that were to happen, then d007 would reference current Sales Employees and Employees in the new department. Not a great situation, right? This creates dirty data, or data that is flawed, and would require extensive cleaning to fix.

What does it mean when the `NOT NULL` constraint is applied?

- ○ The rows of the table will not have null values.
- ○ The columns in the table allow null values.
- ○ Null values are not allowed in rows or columns in the table.
- ○ Null values are not allowed in the column.

Check Answer

The closing parenthesis and semicolon signal that the SQL `CREATE TABLE` statement is complete. Any code added after will need to be included in a new SQL statement. A statement is a command that is set up with a certain syntax. Review our code again.



Each statement will have the same syntax, or set of rules to follow, to perform a successful query. For instance, in the `CREATE TABLE` statement, we tell SQL we're creating a table, we name it, and then enclose any additional parameters within a set of parentheses.

The parameters within parentheses, such as the column names and primary key, are all indented. This is to help keep the code clean and readable.

A semicolon at the very end signals that the statement is complete. It's also part of readability in the syntax and if it's left out it may cause errors.

> **Note**
> Notice that the table names and column names are written in different cases. Table names are typically written in lowercase, while the rest of the statement is written in uppercase. This helps

> differentiate between table and column names as well as commands and parameters.

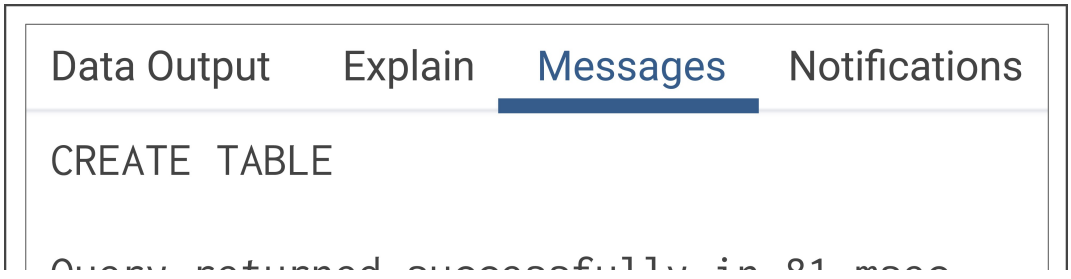
Execute the Code

To save the table to the database, we need to execute the code. To execute code in Python, we first save the `.py` file, then run it in the terminal. In Jupyter Notebook, code is executed by running each cell.

Because we are interacting directly with a database, pgAdmin executes code slightly differently. Right now we have a database, but nothing in it. Imagine that it is a blank canvas. The tables we create are the outline of the landscape, and later it will be filled with color (the data).

In the toolbar of the pgAdmin webpage, find and click the lightning bolt symbol toward the right of the bar. This button runs the code and saves our work to the database.

"Click this button" will run the `CREATE TABLE` statement and save the empty table to the database. A successful execution will return a message confirming the table creation.



```
Query returned successfully in 81 msec.
```

> **Hint**
> The F5 key is the shortcut to run code in pgAdmin.

Troubleshoot Error Messages

Any error message will also appear in the same manner. Encountering errors will happen often and troubleshooting them is a large part of being a developer. Thankfully, each error message encountered will tell us *why* the error occurred. This is great because it helps us, the developers, research and fix the problem.

Practice troubleshooting an error by executing the code to create the departments table again. After running the exact same code again, you'll get this error message:

```
ERROR: relation "departments" already exists
SQL state: 42P07
```

This error occurs because SQL data is persistent and cannot be overwritten if the same command is run again. Once a table has been committed to a database, it is there until a different command is run to delete it.

This helps preserve the integrity of the data already in place. Imagine adding a second Departments table with different data that overwrites the existing table.

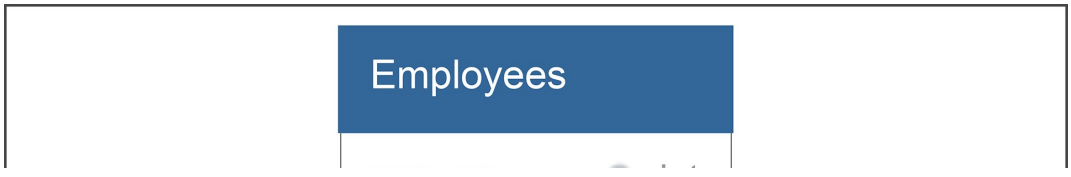
> **Important**
> Data integrity is the quality of the data we're working with. Clean data will yield better results in analysis, and maintaining the data integrity ensures greater accuracy and reliability.
>
> Dirty data is data that contains errors such as duplicates, undefined values (i.e., not a number, or NaN), or other inconsistencies. This is why the `NOT NULL` constraint is in place.

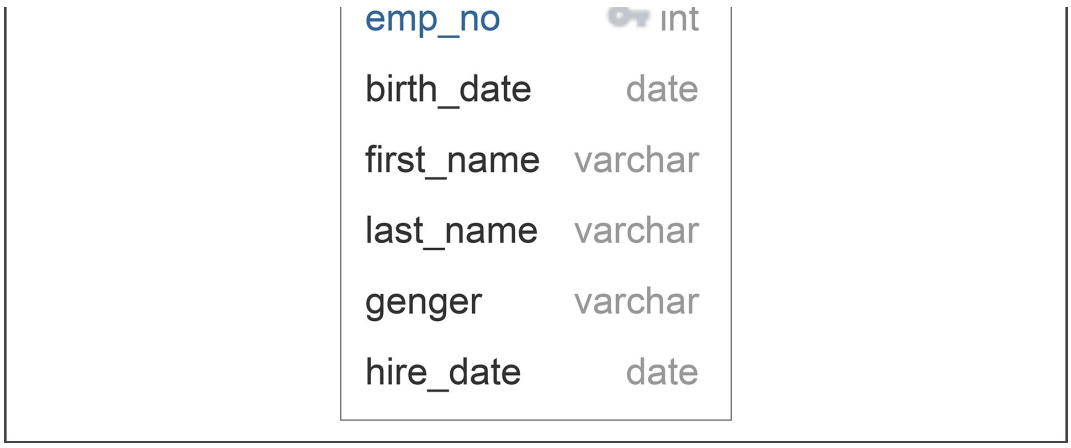
To avoid encountering this error, highlight the code block you want to run first, then execute it. This tells pgAdmin to run *only* that code.

Sometimes error messages are more confusing and require additional research to solve. There's a good chance that lots of people have run into the same issue. Googling the error message itself will likely bring you to a conversation between developers about why the problem occurred and suggestions to fix it. We'll have more practice with troubleshooting as we work through the module.

Create Additional Tables

Create another table for Employees.





Just like last time, we know what we want to name the new table, what the columns are named, and the data type for each.

Start the new table with the `CREATE TABLE` statement and the name of the table.

```
CREATE TABLE employees (
```

> **Rewind**
> Like all coding languages, the ability to add comments is beneficial for developers. Comments are a place to add thoughts when working through a problem and to add explanation to code blocks. Adding clean and clear comments provides a roadmap to the code.

Don't forget the parentheses; remember, all of the details of the table will be inside them. If we make a mistake and the table isn't laid out to match the data, an error will occur during import, which will require troubleshooting––figuring out where the error originated and how to fix it.

> **Note**
> Troubleshooting is part of any coding language and it's something you will encounter from time to time. Learning how to address

errors and implement fixes is a key part of being a developer, and we'll take time later to practice.

IMPORTANT

It's important to keep track of different data types in use. For example, if we use `int` instead of `varchar` for the first_name column, Postgres will search for integers instead of varying characters. Because the first name column will contain only letters and not the numbers pgAdmin is seeking, the import will generate an error and not be completed.

Continue with the table structure.

```
    emp_no INT NOT NULL,
    birth_date DATE NOT NULL,
    first_name VARCHAR NOT NULL,
    last_name VARCHAR NOT NULL,
    gender VARCHAR NOT NULL,
    hire_date DATE NOT NULL,
    PRIMARY KEY (emp_no)
);
```

Now we have everything needed to create a table: the SQL statement, the table components, and the closing parenthesis and semicolon.

Remember to execute the `CREATE TABLE` statement for the new table.

IMPORTANT

Did you run into an error when executing code the second time? That is because pgAdmin will run all of the code in the editor,

unless told otherwise.

We have already created the first table and saved it to the database, so now we only want to save the second table:

1. Highlight only the code you want to run.
2. Click the lightning bolt (execute) button.

Create one more together—this time with foreign keys included. Add the following code to the bottom of your query editor:

```
CREATE TABLE dept_manager (
dept_no VARCHAR(4) NOT NULL,
        emp_no INT NOT NULL,
        from_date DATE NOT NULL,
        to_date DATE NOT NULL,
FOREIGN KEY (emp_no) REFERENCES employees (emp_no),
FOREIGN KEY (dept_no) REFERENCES departments (dept_no),
        PRIMARY KEY (emp_no, dept_no)
);
```

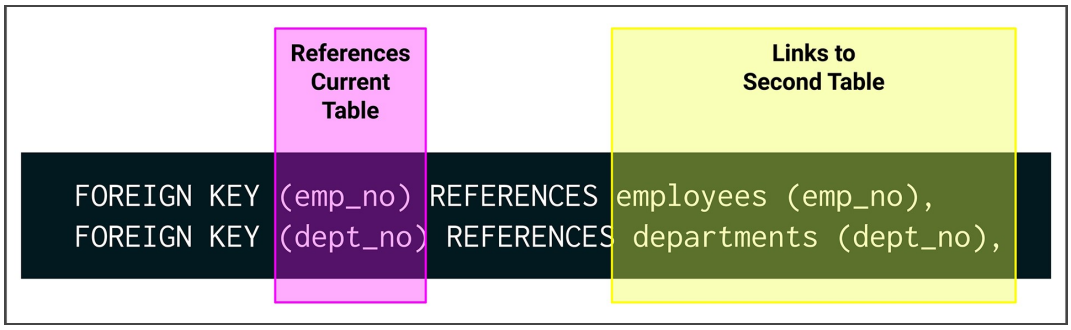
It looks similar to the last table we created, except for two lines:

```
FOREIGN KEY (emp_no) REFERENCES employees (emp_no),
FOREIGN KEY (dept_no) REFERENCES departments (dept_no),
```

Remember that foreign keys reference the primary key of other tables. In the two lines above we can see that:

1. The `FOREIGN KEY` constraint tells Postgres that there is a link between two tables
2. The parentheses following `FOREIGN KEY` specify which of the current table's columns is linked to another table

3. `REFERENCES table_name (column_name)` tells Postgres which other table uses that column as a primary key



The primary key is similar, but there are two keys listed this time instead of just one. Remember the analogy of phone numbers being primary and foreign keys? Think of the two primary keys listed as two unique phone numbers that belong to dept_manager. The Employees and Departments tables are other numbers that have been saved as contacts. One thing to keep in mind when working with foreign keys is that it's possible that data insertion will fail if the foreign key isn't present. This is a "foreign key constraint" and in this case, it means that the new data needs a reference point (such as dept_no or emp_no) to be successfully added to the table.

Let's create another table for the data in `salaries.csv`. At the bottom of the query editor, type the following:

```
CREATE TABLE salaries (
  emp_no INT NOT NULL,
  salary INT NOT NULL,
  from_date DATE NOT NULL,
  to_date DATE NOT NULL,
  FOREIGN KEY (emp_no) REFERENCES employees (emp_no),
  PRIMARY KEY (emp_no)
);
```

This code tells Postgres that our new table is named “salaries” and we’ll have columns for the emp_no, salary, from_date, and to_date. We also have specified that certain fields aren’t allowed any null space with the `NOT NULL` constraint, which is important because we want this data to be persistent for every employee. As a final step in table creation, we’ve also specified primary and foreign keys.

> **Skill Drill**
> Using the entity relationship diagram as a guide, create the remaining tables in pgAdmin. Use the ERD as a reference guide if you need to.
>
> Remember to use the REFERENCES keyword when creating the tables, such as we did when we created the dept_manager table. Otherwise, your code will generate errors when you import the data.

Query for Confirmation

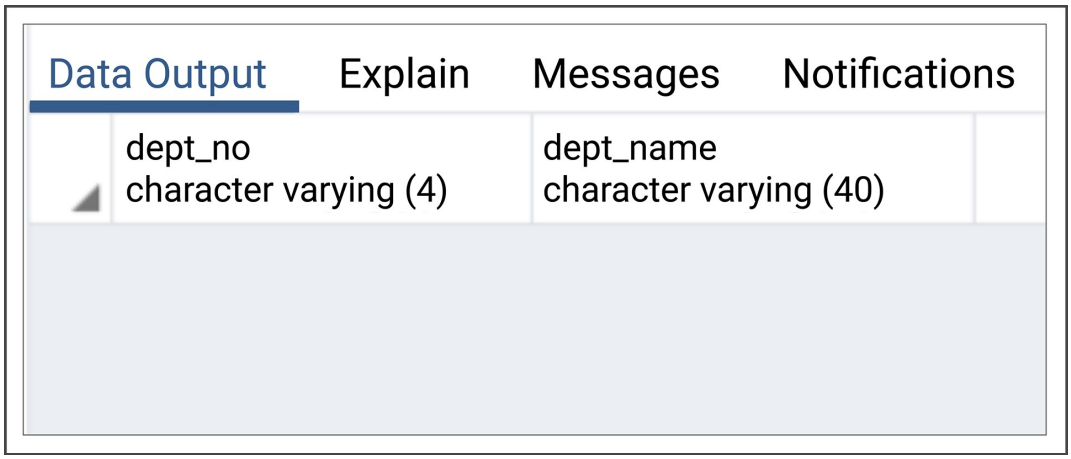
Confirm the tables were created successfully by running a `SELECT` statement, which performs a query instead of constructing anything. In Python, we would create a script to tell a computer how to act. With SQL, we create queries instead. These queries are interactive—we’re speaking directly to the data.

Think of it as asking the database a question. For example, say we want to know how many columns are in the departments table. How would we ask that particular question? We would create a SELECT statement, then run the code. This is called “querying the database.”

In the editor, after the table creation statements, type `SELECT * FROM departments;`.

- The `SELECT` statement tells Postgres that we're about to query the database.
- The asterisk tells Postgres that we're looking for every column in a table.
- `FROM departments` tells pgAdmin which table to search.
- The semicolon signifies the completion of the query.

After executing the `SELECT` statement, pgAdmin will automatically show the result in the Data Output tab at the bottom of the page.



Currently, we're able to see that the two columns exist and the type of data they contain. If we run the `SELECT` statement again after importing the CSV data, then the output would include the actual data as well. We'll start importing the data next.

> **Important**
> The information in the database is static, which means that it will always be in the database unless directly altered, but the query editor is not. It's similar to working in a Microsoft Word document: If something happens to your computer before you saved your

work, there's a good chance it'll be lost. If your computer crashes mid-query, the pgAdmin editor won't hold onto your code for you during a reboot.

Our queries are the meat and potatoes of SQL. We're finding connections between different tables and answering questions with the results. Even though losing a query isn't the end of the world (they can be rebuilt, after all), it does take time to replicate the work already completed.

To save your hard work in the query editor, it's actually a good idea to keep an external .sql file with your code saved in your class folder.

Copy and paste the code you used to create tables into a new VSCode window and save it as `schema.sql`. This way, you'll always have access to your code if you need to refer to or use it again.

Similarly, when we begin writing queries, create another `.sql` file named `queries.sql` and make sure to save successful queries in it as we go along.

Add, Commit, Push

After saving your `schema.sql` file, remember to push it to your repo!

7.2.3: Import Data

Alright, we've created a database. We've written our first SQL code and created tables modeled after our ERD. Bobby is almost ready to begin his analysis. The next step is to import the data from the CSV files. We'll make sure all of the tables we created in pgAdmin appear in the GUI first, because we'll be using the GUI to import data. The import itself should go smoothly, thanks to the thought and careful attention to detail we put into our diagram and table creation.

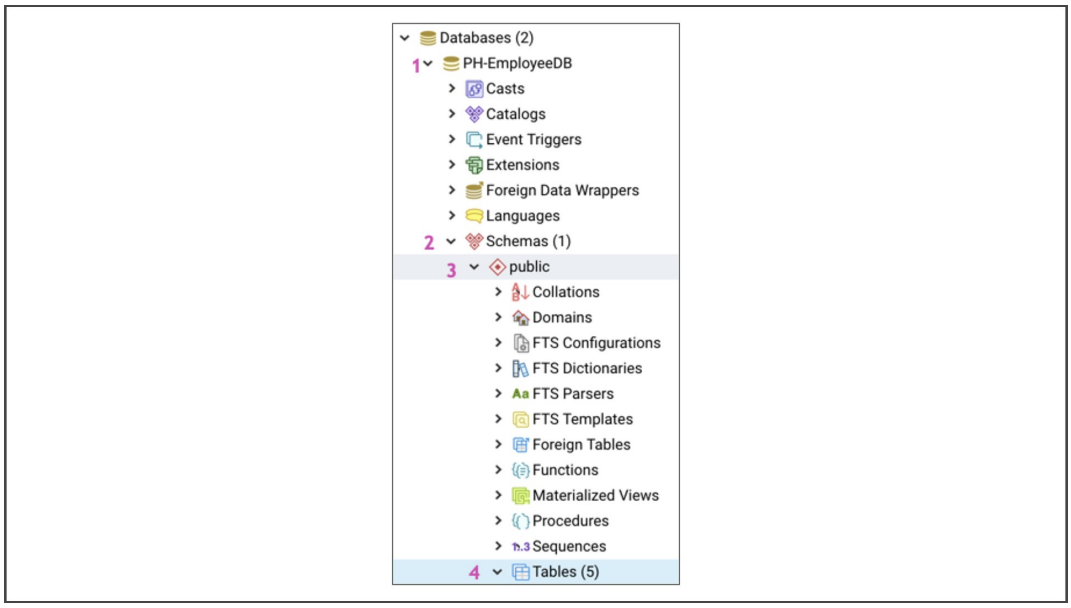
Now that Bobby's tables are ready, let's begin importing data.

> **Important**
> SQL is very interactive. Developers are not only importing data and asking it questions through the query language, but they can also update and edit the data stored in the tables as needed.
>
> For example, if a single row of data needs to be added to an existing table, a developer can manually add it by using the `INSERT` statement.

> If the data in a table is small enough in scale, it can be manually inserted this way completely, instead of importing a CSV file.
>
> Alternately, necessary edits and updates are completed manually as well. We won't be manually editing or uploading data to our tables in this lesson because our datasets are too large.

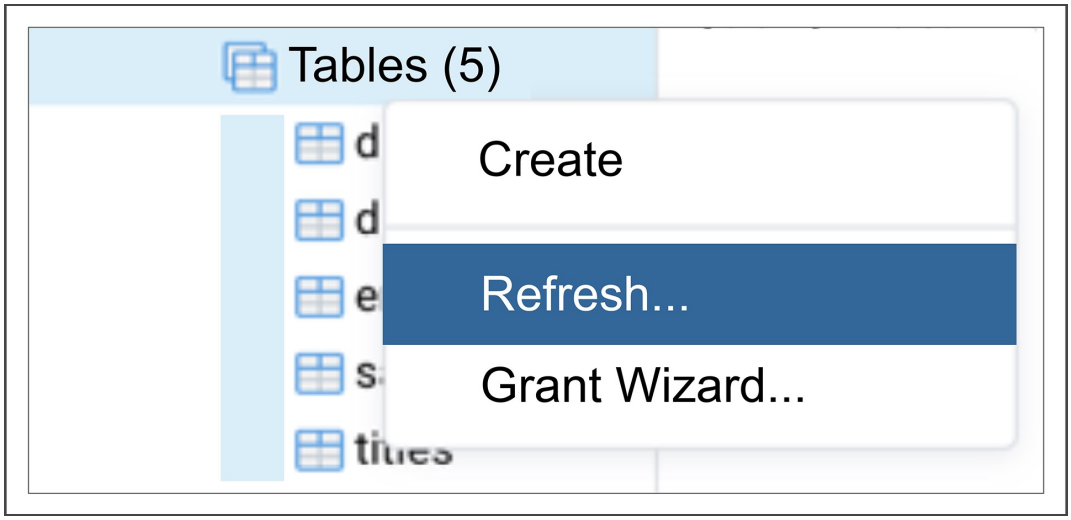
In the pgAdmin window, select the dropdown menu for our PH-EmployeeDB database. To import data into the tables, first confirm all of our tables are listed:

1. Find the PH-EmployeeDB collapsible menu and click it.
2. Scroll down and click "Schemas" to expand the menu.
3. Click "public."
4. Scroll down to "Tables" and note the number in parentheses.



We created six tables, one for each CSV file. If all six don't appear in parentheses, there are two things you can do:

1. Make sure you executed the `CREATE TABLE` statements for each by highlighting the specific code and clicking the lightning bolt on the pgAdmin toolbar.
2. Right-click the “Tables” menu and click “Refresh.”



Now that the database is updated, we can insert data. We need each table visible from the dropdown menu because we are importing data directly into each table without using the query editor.

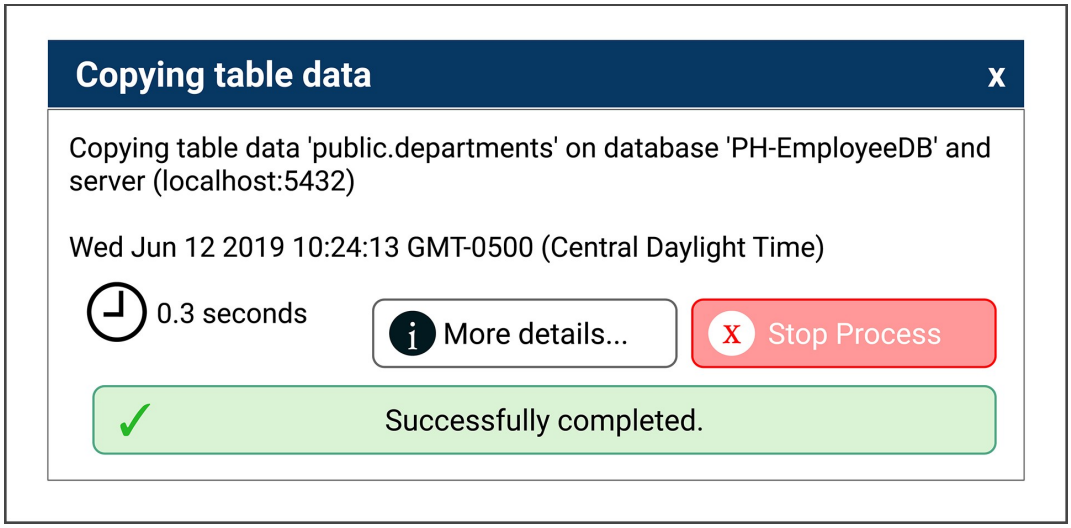
To import a CSV into Postgres with pgAdmin, follow these steps. We'll customize our options to fit our data import, and then check the table to make sure the data has been imported successfully.

1. Right-click the first table, departments.
2. From the menu that pops up, scroll to Import/Export.
3. Toggle the button to show "Import."
4. Click the ellipsis on the Filename field to search for your project folder.
5. Select departments.csv. Make sure Format is set to "csv" and Encoding is blank. **Note:** By default, the Encoding section is blank. If our files were encoded to provide an extra layer of security, we would need to select the type of encoding before importing them to

Postgres. We don't have to worry about this, though. Also, if "Encoding" is filled in with an encoding type such as BIG5 or LATIN1, cancel the import and start over.

6. Leave the OID field as is, but toggle the Header field to "Yes" and select the comma as the Delimiter. **Note:** If we don't specify that there is already a header included in the CSV data, then the header will be imported as data. This would result in errors because headers don't always match the data types in the columns.
7. Click OK to begin importing the data.

If the import is successful, a pop-up window will appear at the bottom of your pgAdmin page:



Check the import by typing `SELECT * FROM departments;` at the bottom of the query editor. The resulting table should mirror the CSV file:

5	d005	Development
6	d006	Quality Management
7	d007	Sales
8	d008	Research

If you see all nine departments, great job. If not, then there is probably an error somewhere in the table creation.

> **IMPORTANT**
> If you don't see the same nine tables as shown in the graphic above, then it's time to troubleshoot. Look for error messages in pgAdmin (this will be obvious and will pop up during the import). Also check the data itself. Is the file corrupted? Is anything missing?
>
> Also, search Google for answers. Have a specific, confusing error? Copy and paste the error message in the browser search field to find online forums discussing the same problems.
>
> Developing troubleshooting skills is critical to mastering any coding language, which is a lifelong process. The tools and libraries we use are often updated or changed in minor ways that can produce confounding bugs, so it's good to seek out help when encountering errors. Developers often turn to the Stack Overflow community for help resolving such dilemmas. This community is welcoming to users with all levels of experience—they were new once, too!
>
> We'll also be covering error handling in the next section, so don't worry if you've run into an error here.

Import data into the dept_manager table next. Follow the same steps as before to import data, only this time, right-click on the dept_manager table in the dropdown menu.

Remember to:

1. Find the correct CSV file to upload using the “filepath” field.
2. Toggle the “Header” field to “Yes.”
3. Select a comma as the delimiter.

Click OK to upload the CSV file.

> **Skill Drill**
> Import the data from each CSV into its corresponding table.

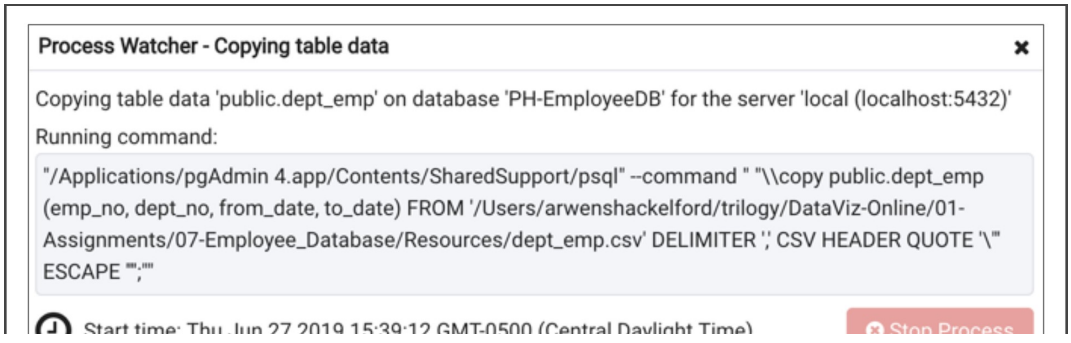
7.2.4: Troubleshoot Imports

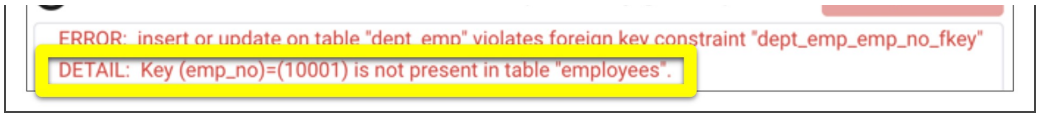
Every developer in every coding language will encounter an error. Sometimes they're due to typos, other times the code syntax is incorrect. In each case, there will be an error code provided. These codes help us (the developers) research the error and investigate solutions.

What if we run into an error during our table imports? What could be the possible cause? We'll help Bobby practice troubleshooting because this skill is one that is worth developing early.

Handle Common Errors

What if Bobby runs into an error while he's importing the data? Below is an example of a likely error:





Because the `FOREIGN KEY` constraint references other tables, we need to import the data in a specific order.

> **Hint**
> The error message above provides sufficient error text for a developer to Google. If searching for the detail in the data or your code doesn't help, Google the error itself for additional troubleshooting assistance.

For example, the dept_emp table references the Employees table through its foreign key. If there is no data in the Employees table, then there are no foreign keys to link to, and an error will occur.

Handle Mismatched Data Types

Another common scenario is when a data type in a table we've created doesn't match the CSV data. What should we do?

Because data within a Postgres database is static, we can't go back and fix a typo in our original table creation code. Remember the "table already exists" error from earlier? Postgres has this check in place so we don't accidentally overwrite good data.

If you need to update a table column to fix its data type, what is the best approach?

- ○ Update the code then rerun it because that will apply an update to the existing table.

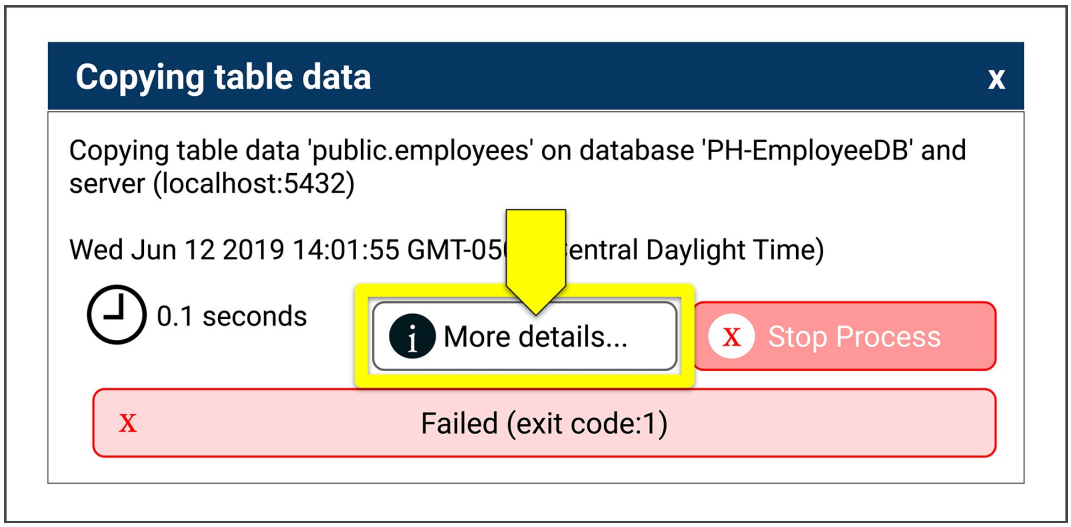
If you encounter an error during data import that corresponds to the incorrect data type in the table, you'll need to start fresh by deleting the table and creating a new one in its place.

Trying to update the `CREATE TABLE` statement and then re-executing it won't work because SQL doesn't allow duplicates or data overrides. Creating an entirely new table with a new name would work, but that adds two more complications:

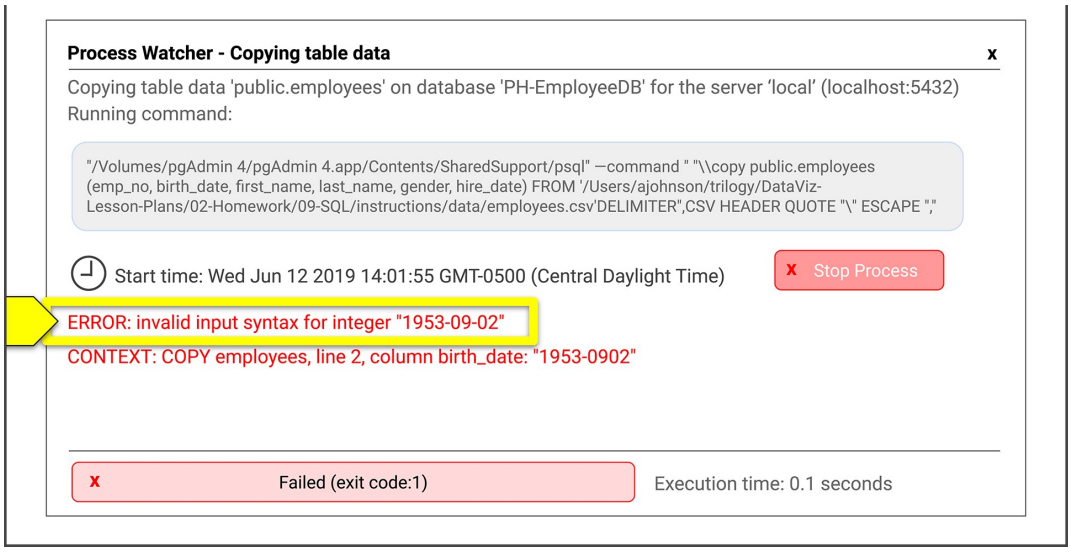
1. The old table will still exist and clutter up the fresh database.
2. The database doesn't follow the original layout in the entity relationship diagram.

The best solution is to delete the current, unusable table using the `DROP TABLE` statement, then create a new one with updated code.

Imagine that we created the Employees table with the wrong data type assigned to the birth_date column. When we tried to import the CSV data to the table, we encountered the following error:



That error doesn't tell us much more than that the import failed, but clicking on "More details" is enlightening.



Now the error tells us exactly where the import failed: The birth_date column should be a date instead of an integer. The correct resolution is to remove, or drop, the current table then recreate it using the proper data type.

Drop a Table

To drop the table, the following code is used:

```
DROP TABLE employees CASCADE;
```

- `DROP TABLE employees` tells Postgres that we want to remove the Employees table from the database completely.
- `CASCADE;` indicates that we also want to remove the connections to other tables in the database.

> **IMPORTANT**
> Even without data, by adding foreign keys that reference other tables, we've created a network of data connections. Not every

> table will need the `CASCADE` constraint, but it will come up when you need to drop a table that already has a defined relationship with another. Any table that does not reference a foreign key can be dropped without the `CASCADE` constraint.

After the table has been dropped, we can adjust the `CREATE TABLE employees` code block to match the CSV's data types. After re-executing the code to create that table, Postgres will add the new table to the database. Now we're ready to finish the data import.

7.3.1: Query Dates

Now Bobby is ready for analysis. He has created tables based off of his entity relationship diagram, and he's successfully imported the data to each table. That's a pretty big chunk of work he's completed. Not to mention the analysis—that's a big step, too.

The next part is to perform the analysis he was tasked with: future-proofing the company by determining how many people will be retiring and, of those employees, who is eligible for a retirement package.

Bobby will need to dive into the data and perform queries to learn when employees were hired as well as their age. To do this, he will need to create statements that use conditionals.

In Python, conditionals such as "if" and "else," and the logical operator "and," are similar to conditional expressions used in SQL. We're querying the database, and the returned results are based on the conditions we have set. For example, one of Bobby's tasks is to search for folks who are

retiring soon. The query he builds will include a condition involving employee birthdays. We need to know when they were born to determine when they'll retire, right?

Determine Retirement Eligibility

Bobby's boss has determined that anyone born between 1952 and 1955 will begin to retire. The first query we need to help Bobby write will return a list of those employees. At the bottom of your query editor, type the following code:

```
SELECT first_name, last_name
FROM employees
WHERE birth_date BETWEEN '1952-01-01' AND '1955-12-31';
```

At a glance, we can tell we're searching for the first and last names of employees born between January 1, 1952, and December 31, 1955. Let's break the code down into its components, though, so we can really understand what it's doing:

- The `SELECT` statement is more specific this time. Instead of an asterisk to indicate that we want all of the records, we're requesting only the first and last names of the employees.
- `FROM employees` tells SQL in which of the six tables to look.
- The `WHERE` clause brings up even more specifics. We want SQL to look in the birth_date column for anyone born between January 1, 1952, and December 31, 1955.

Notice how `BETWEEN` and `AND` are both capitalized in our statement? This is part of the SQL syntax. It not only signals the conditions present, but also makes the code easier to read.

How many employees are ready for retirement?

- ○ 1,000 to 3,000
- ○ 4,000 to 6,000
- ○ 7,000 to 10,000
- ○ 10,000 or more

Check Answer

Finish ▸

Pewlett Hackard has a lot of employees getting ready to age out of the program. This is going to create a considerable amount of openings. Refine this list further by looking only at how many employees were born in 1952. Create another query that will search for only 1952 birth dates.

```
SELECT first_name, last_name
FROM employees
WHERE birth_date BETWEEN '1952-01-01' AND '1952-12-31';
```

This query is almost the same as the last. We've only changed a single digit: the year was switched from 1955 to 1952 after the `AND` clause.

> **Skill Drill**

> Create three more queries to search for employees who were born in 1953, 1954, and 1955.

Narrow the Search for Retirement Eligibility

There are quite a few folks getting ready to retire. Each of those new queries has a lengthy list of people. Let's see if we can narrow it down a bit more by adding another condition to the query. We'll start with our original query, shown below.

```
-- Retirement eligibility
SELECT first_name, last_name
FROM employees
WHERE birth_date BETWEEN '1952-01-01' AND '1955-12-31';
```

We'll modify this query to include a specific hiring range. This time, we're looking for employees born between 1952 and 1955, who were also hired between 1985 and 1988. The modification is subtle, too. We're going to adjust one line of the code block and add another to the end.

The first piece, an adjustment, is to place parentheses around the `WHERE` clause (without including the keyword itself). We'll also remove the semicolon since the code block isn't complete yet. The updated code should look as follows:

```
-- Retirement eligibility
SELECT first_name, last_name
FROM employees
WHERE (birth_date BETWEEN '1952-01-01' AND '1955-12-31')
```

Next, we'll add the final line of code. Our current code has a single condition in place that tells Postgres to search only for people born between 1952 and 1955. The next line of code is our second condition that they were also hired between 1985 and 1988.

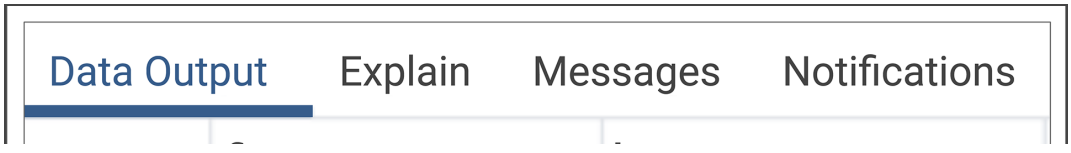
```
AND (hire_date BETWEEN '1985-01-01' AND '1988-12-31');
```

Notice how the second condition is inside parentheses? This is a tuple; in Python, data can be stored inside a tuple and accessed in the same way as a list. In SQL, the tuples in this block of code are part of the syntax. They basically place each condition in a group, and Postgres looks for the first group first, then looks inside the second group to deliver the second condition. Altogether, the completed code block consists of the following:

1. The `SELECT` statement, pulling data from the first and last name columns
2. The `FROM` statement, telling Postgres from which table we're getting the data
3. And two conditional statements: the dates of birth and the dates of hire

```
-- Retirement eligibility
SELECT first_name, last_name
FROM employees
WHERE (birth_date BETWEEN '1952-01-01' AND '1955-12-31')
AND (hire_date BETWEEN '1985-01-01' AND '1988-12-31');
```

Run the completed query to view the result.



	first_name character varying	last_name character varying
1	Georgi	Facello
2	Chirstian	Koblick
3	Sumant	Peac
4	Kazuhide	Peha

A list of first and last names should appear in your Data Output window. It's a long list, too—too long to scroll to the end in any reasonable amount of time.

> **Rewind**
> Much like the `count()` method in Pandas, PostgreSQL has a `COUNT` function that will count the number of rows in a table. Like Python, when we use this function in SQL, we can find the length of a table, or how many viable rows of data exist in it.

Count the Queries

In Postgres, the `COUNT` function works in a similar fashion as the Python `count` method: It counts the rows of a specified column. We can insert the `COUNT` function into our original query, with some adjustments, but because we want a different result, it's better to create a new query instead.

The new query is extremely similar to the original. We only need to use a single column to get the count, though. Copy the original query and paste it at the bottom of the query editor so we can update it.

```
-- Number of employees retiring
SELECT COUNT(first_name)
FROM employees
WHERE (birth_date BETWEEN '1952-01-01' AND '1955-12-31')
AND (hire_date BETWEEN '1985-01-01' AND '1988-12-31');
```

Our new query is almost exactly the same as the original with one exception: the `COUNT()` function has been inserted after the `SELECT` statement, and only one column `(first_name)` is included.

With this update, we are telling Postgres to count the number of rows in the first_name column. We could change the column to last_name and get the same results. This is because the length of our query is the same regardless of which of the two columns we use in the `COUNT` function.

What is the `COUNT` of the current query?

- ○ 41,380
- ○ 38,410
- ○ 40,138
- ○ 13,840

Check Answer

Finish ▸

Create New Tables

Bobby has a list of retirement-eligible employees to submit to his boss, but it's still on the database. If data was imported to the database using CSV files, why not export it the same way?

We'll need to take a couple of extra steps to help Bobby prepare his data. In Postgres, only data that is saved as a table can be easily exported. We already have a query that returns the results we need, so let's use that to create our table. Copy the query that returns employees born and hired within the correct window, then paste it to the bottom of the query editor again.

For reference, the original query we'll be modifying is as follows:

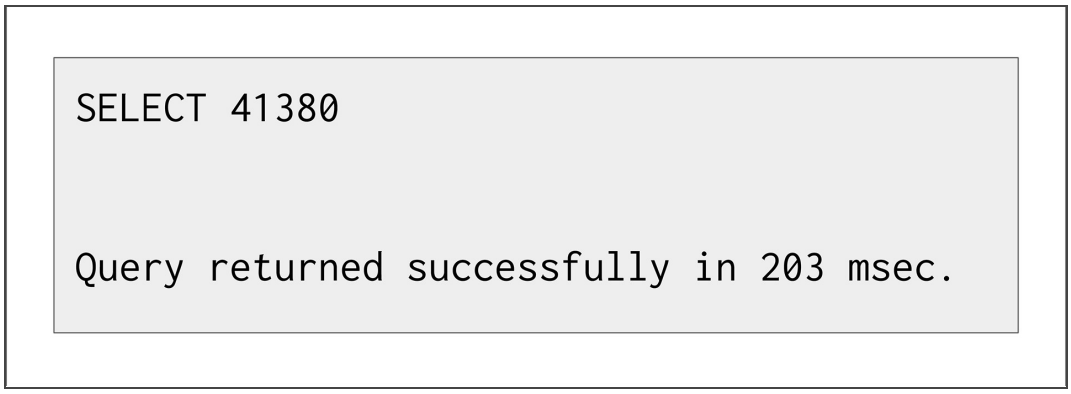
```
SELECT first_name, last_name
FROM employees
WHERE (birth_date BETWEEN '1952-01-01' AND '1955-12-31')
AND (hire_date BETWEEN '1985-01-01' AND '1988-12-31');
```

This time, the change we'll make to the code is also small—we're modifying the `SELECT` statement into a `SELECT INTO` statement. This statement tells Postgres that instead of generating a list of results, the data is saved as a new table completely. Update our code to include the `INTO` portion of the `SELECT` statement.

Insert a new, blank line between the `SELECT` and `FROM` sections of the code. In this vacant space, type in `INTO retirement_info`. With the addition of this line, we're telling Postgres to save the data into a table named "retirement_info." The complete code should look as follows:

```
SELECT 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');
```

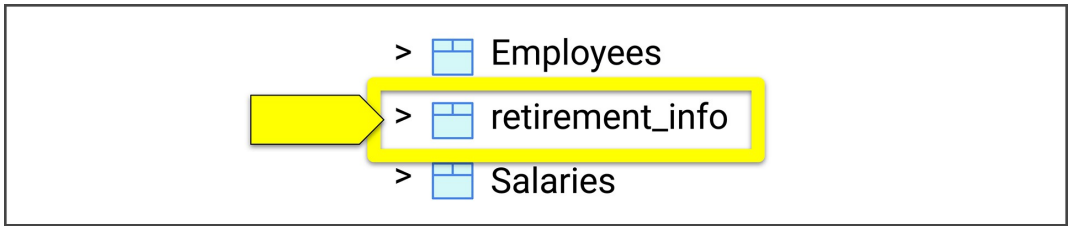
When executing this code, instead of receiving a list of results, you should see a message confirming the query has been successful.



So how can we see what the table looks like? We query it with a `SELECT` statement.

```
SELECT * FROM retirement_info;
```

Our list of data from earlier is now an actual table that we can use with statements and functions to perform analysis. Additionally, if you refresh the list of tables from the dropdown menu on the left, it will now appear in the list.

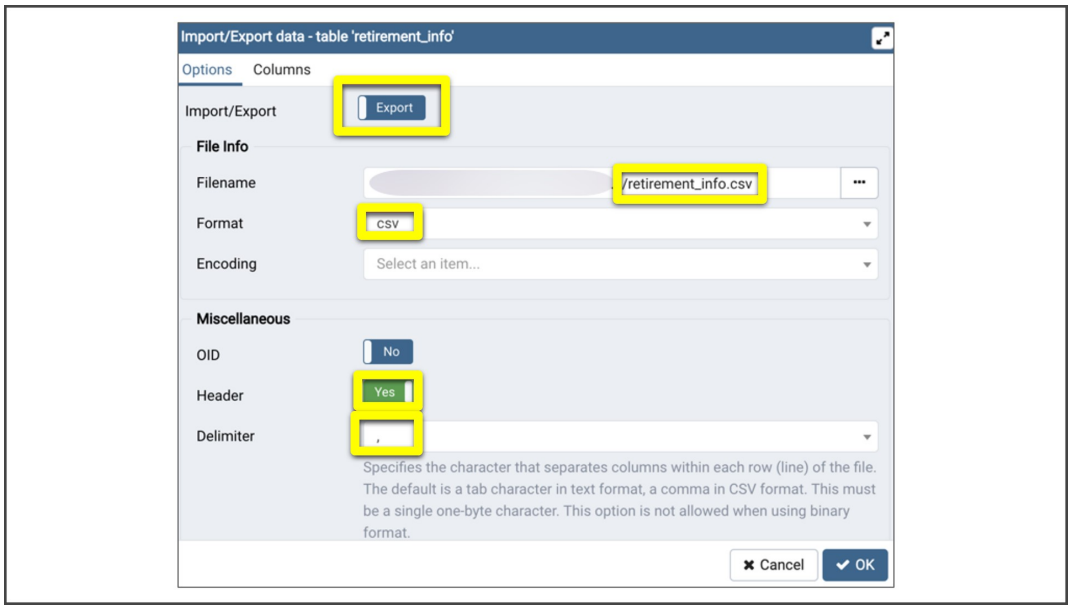


This list of tables is where we'll export the data.

Export Data

The export setup is very similar to the import. Let's walk through exporting the "retirement_info" table together.

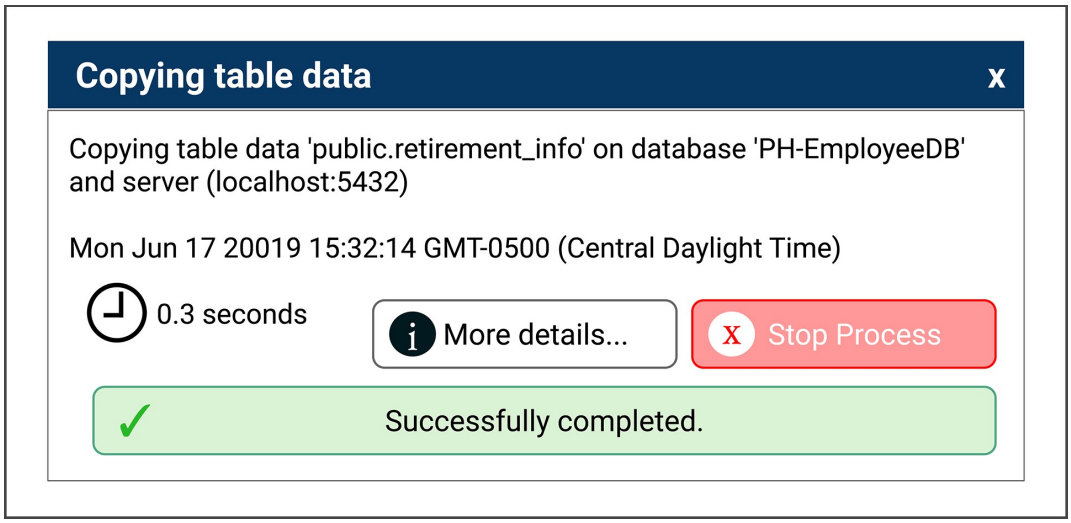
Right-click on your new table and select "Import/Export." Instead of importing anything, this time we'll be exporting.



Then, follow these steps:

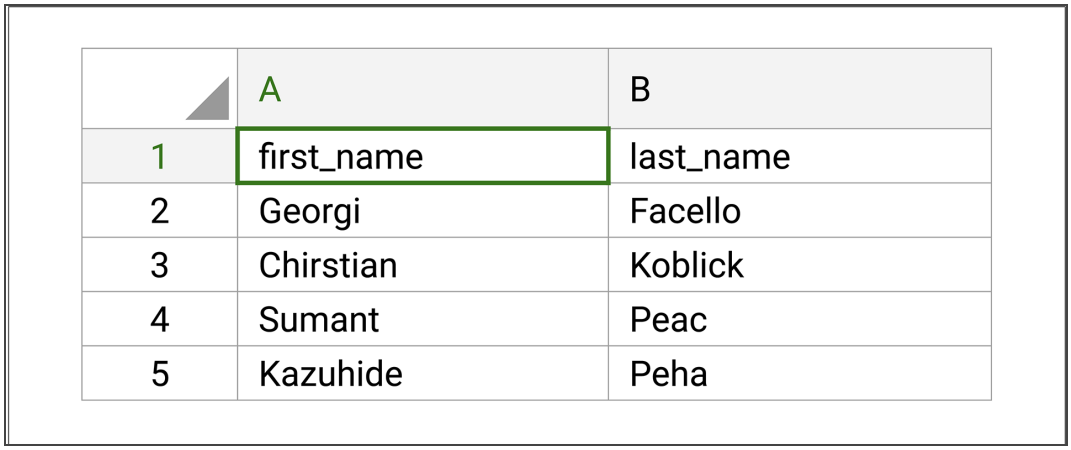
1. Keep the Import/Export button toggled to "Export."
2. Click on the `...` in the Filename field to automatically select the same directory from which you imported the other CSVs. Select a directory, but be sure to rename it to `retirement_info.csv`.
3. Be sure the format is still CSV.
4. Toggle the Header section to "Yes" to include column names in the new CSV files.
5. Select the comma as the delimiter to maintain the same format with all CSV files.

Click OK to start the export. After the file has been created, pgAdmin will confirm our file is ready to be viewed.



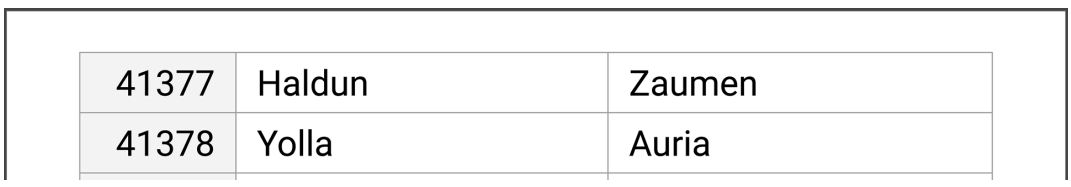
Open your project folder and navigate to the resource file holding your CSVs, then open the new file to verify the data was successfully exported.

The first five names look the same as our earlier query results.



	A	B
1	first_name	last_name
2	Georgi	Facello
3	Chirstian	Koblick
4	Sumant	Peac
5	Kazuhide	Peha

Did the entire list get exported? In Excel, use the hotkey to move to the bottom of the spreadsheet (Command + down arrow for a Mac, or CTRL + down arrow for Windows).



41377	Haldun	Zaumen
41378	Yolla	Auria

41379	Mihalis	Crabtree
41380	Mohammed	Pleszkun
41381	Nathan	Ranta

The Excel index shows there are 41,381 rows, including the header row. That's everything. This file holds the information requested: the names of all employees who were born between 1952 and 1955 and who were hired by Pewlett Hackard between 1985 and 1988.

This hefty list of employees is now ready for Bobby to present to his manager. Let's keep going.

> **IMPORTANT**
> Much like we created a `.txt` file to save our ERD schema creation text, we should save our queries as we go as well. This is for a couple of reasons.
>
> First, if our computer crashes, pgAdmin may not save the text in our query editor. While the database is static and new tables will remain in place, it's a good idea to keep the queries on hand to reference later.
>
> Second, referencing the queries is helpful for us because we may be able to adjust and alter existing code to reuse it later. It also is useful for other people to understand how we created our queries and new tables.
>
> Copy the SQL queries we've created so far in our pgAdmin query editor into a new VSCode window. Save this file as `queries.sql` in our repo folder. Make sure to add the new file and commit it to our repo. This way, we won't need to worry about our work getting lost

or deleted.

You'll want to save new code we create to this file and commit it as we continue to progress through the module.

Add, Commit, Push

Remember to add, commit, and push your work!

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.

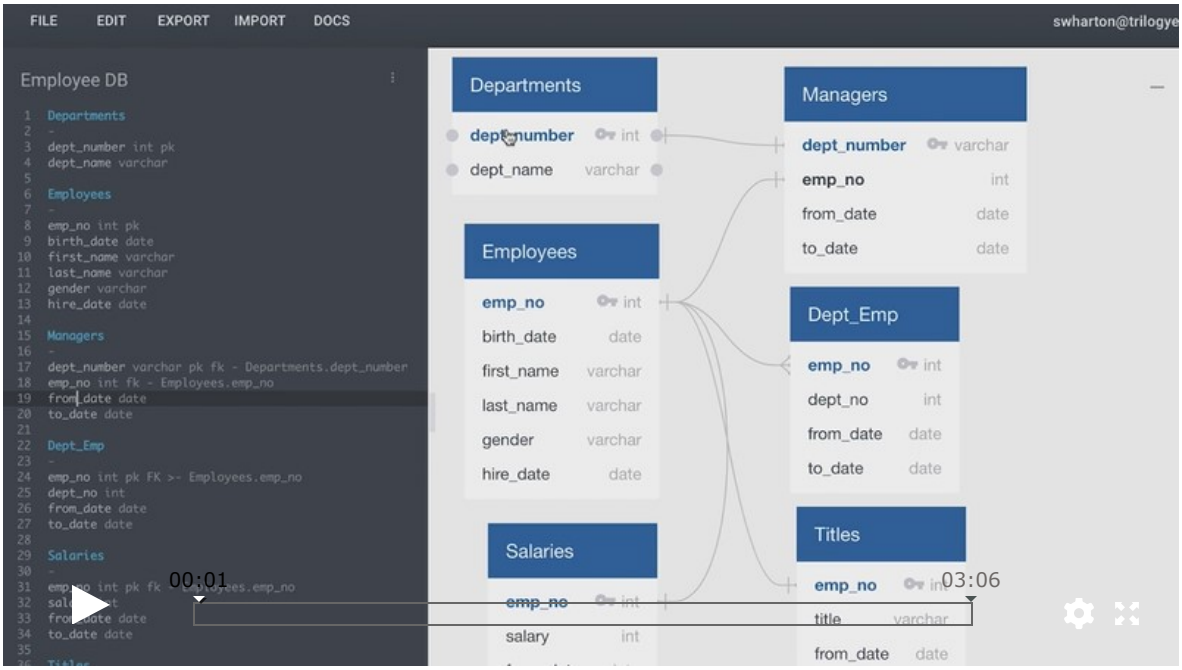
Merging DataFrames in Pandas is very similar to joining tables in SQL. There are several different types of joins and each one will yield a different result. We'll cover each type of join available in SQL as well as where and when to use it.

> 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.

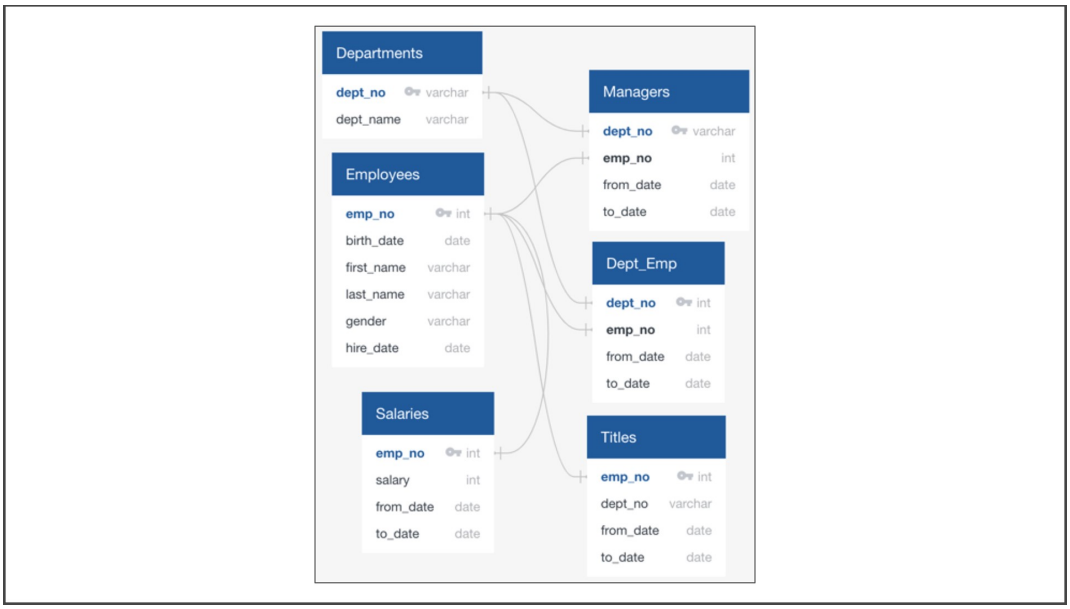
Bobby will need to break down that large list into smaller pieces, then present those to his boss instead. We'll have a little bit of tweaking to do and a few more queries to build.

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.

Check Answer

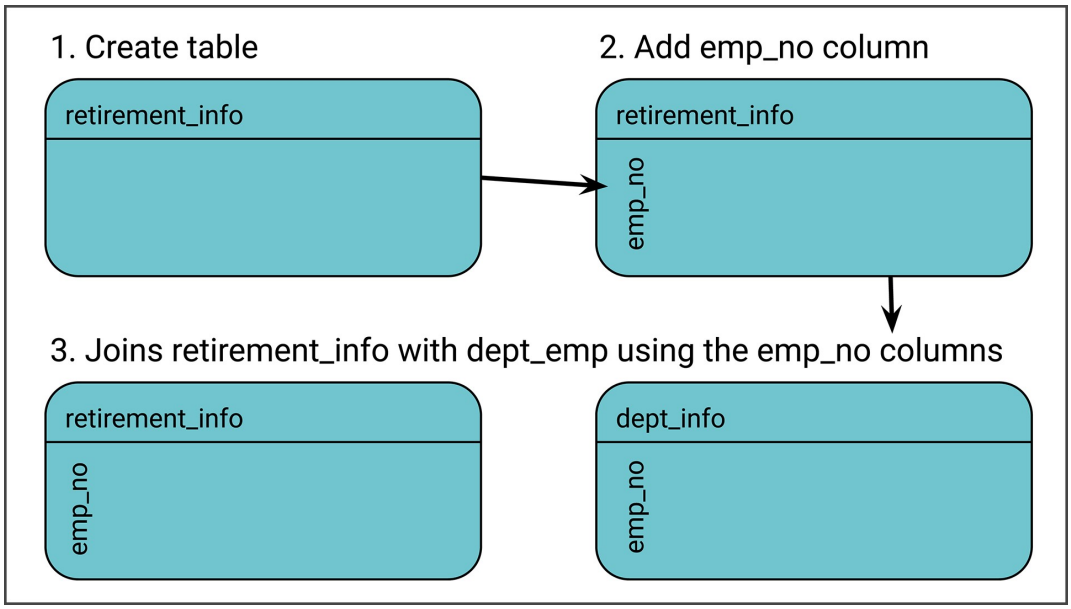
The Dept_Emp table contains both an emp_no and a dept_no. Between the Employees table and the Dept_Emp*p* table, we would have:

- 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`

Check Answer

Finish ▸

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;
```

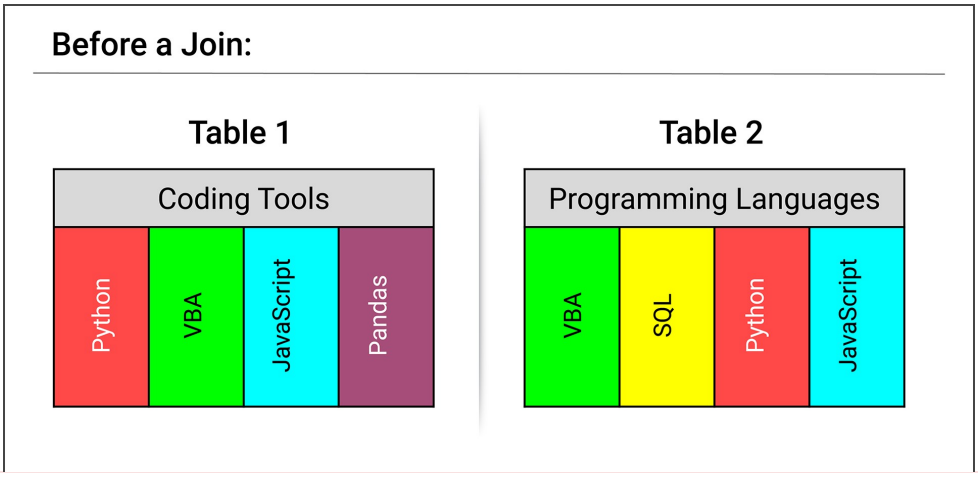
After executing this code, the retirement_info table that's generated now includes the emp_no column. We are ready to begin combining different tables using joins to help Bobby create the new list his boss has requested.

Use Different Types of Joins

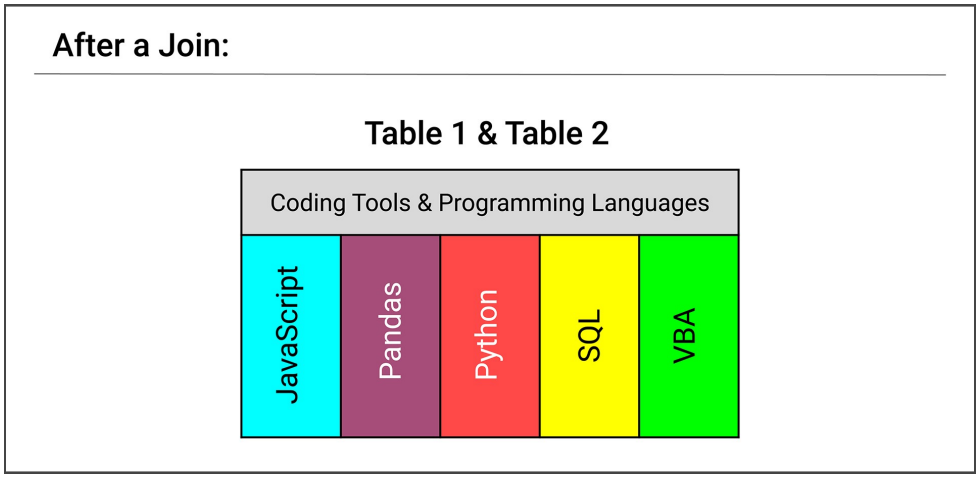
Joins are common in every coding language that deals with data. There are several types of joins available for use, depending on what data we want displayed. For example, in the retirement_info table, we want all of the information from the table. We'll be joining it with the dept_emp table, but what 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.



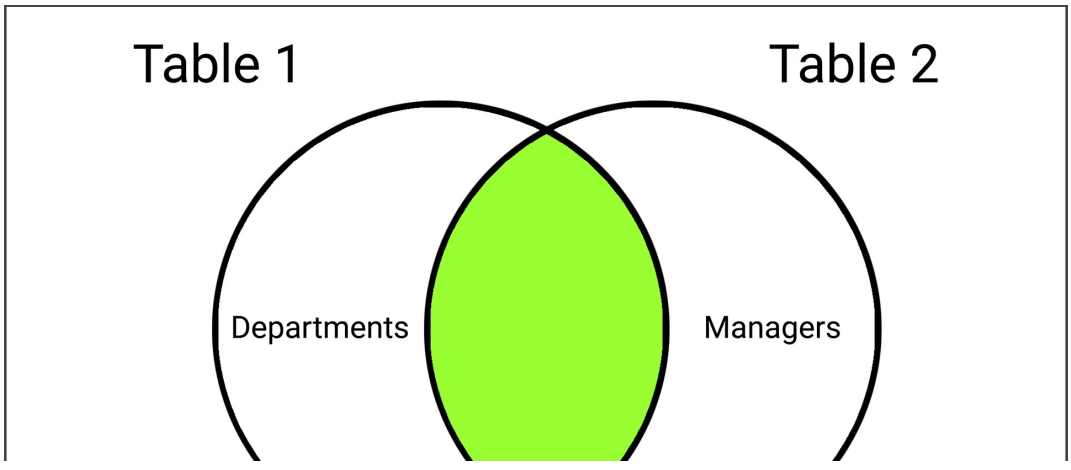
Joins treat the tables in this manner, as if they're side-by-side, and it makes it a bit easier to visualize how the data will be joined.

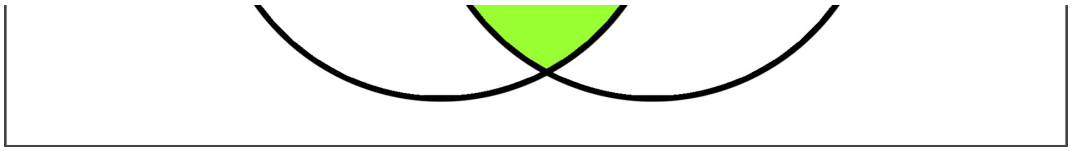


Which type of join would be best suited for this task? Let's take a look at what is available with Postgres.

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.

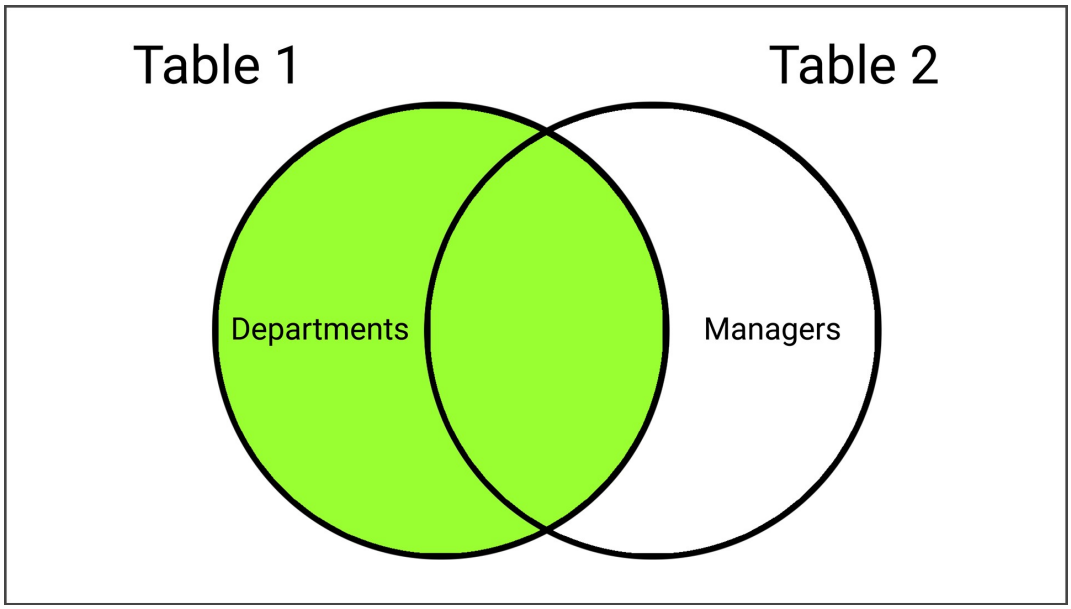




Say we want to view the department number and name as well as the managers' employee numbers. We only want the matching data from both tables, though. This means that if there are any NaNs from either table, they won't be included in the data that gets returned. So if a department doesn't have a manager,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

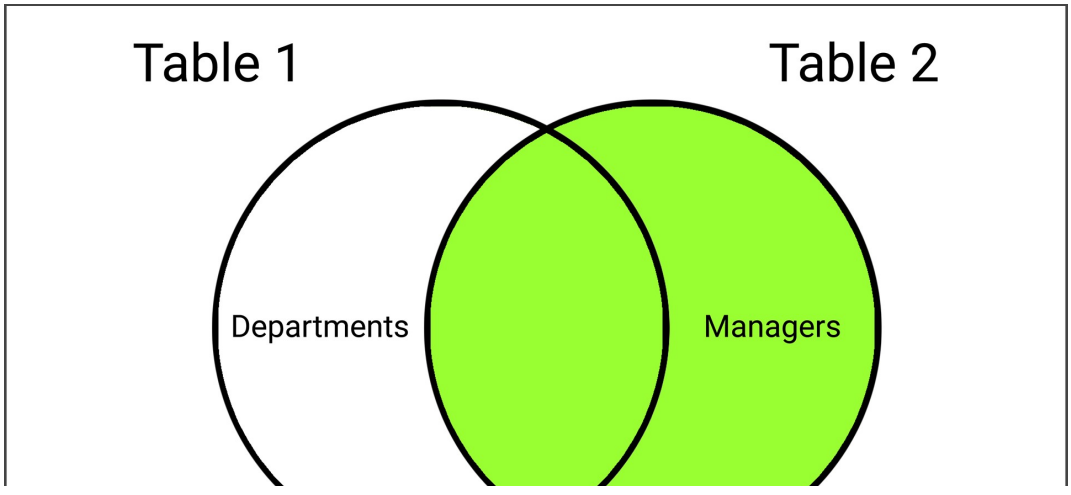
This time, when rows from the Managers table do not have matching data for every row in the Departments table, a NaN is inserted into that column and row intersection instead. This way, all of the data we want included from Table 1 is still present, and data from Table 2 isn't rearranged and mismatched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

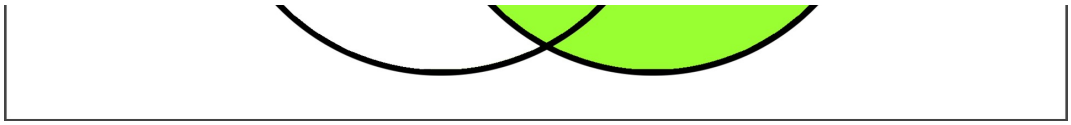
If we wanted to add information from the Managers table to the Departments table, we'd use a left join. This way, every row of the Departments table would be returned with or without manager information. If a department doesn't have a manager, a NaN would appear in place of actual data, and it would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



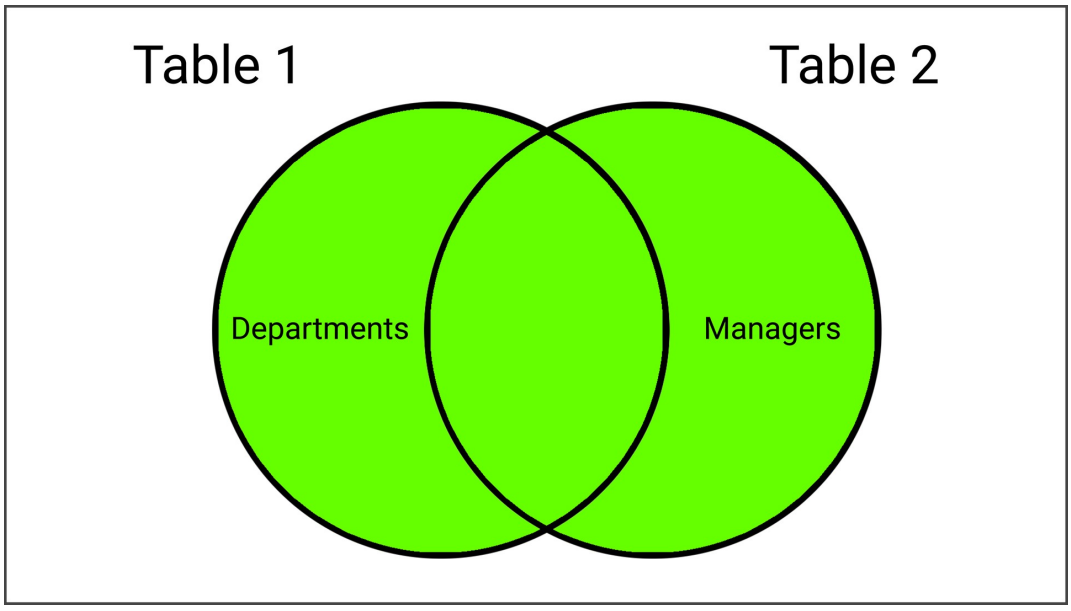


If a manager wasn't assigned to a department, the result would look as follows:

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

7.3.3: Joins in Action

The first round of employee lists Bobby generated were extensive, but not specific enough for our needs. Now that we've researched the different types available to us as well as when to use them, we'll be able to help Bobby create the lists he needs to present to his supervisor.

Joins can be very confusing, so we'll be practicing using them not only to make new tables fitting the bill for Bobby's task, but also so we can get more comfortable with them and even level up our skills a bit.

We aren't limited to only two tables when using joins, we can combine three or even four tables if needed. As you can imagine, the code can get a bit messy with that many moving parts. We'll practice not only joining multiple tables, but also a way to clear up code by using aliases, where we assign a nickname to a table.

Use Inner Join for Departments and dept-manager Tables

Let's create a query that will return each department name from the Departments table as well as the employee numbers and the from- and to- dates from the dept_manager table. We'll use an inner join because we want all of the matching rows from both tables.

The code for our inner join would look like the following:

```
-- Joining departments and dept_manager tables
SELECT departments.dept_name,
     dept_manager.emp_no,
     dept_manager.from_date,
     dept_manager.to_date
FROM departments
INNER JOIN dept_manager
ON departments.dept_no = dept_manager.dept_no;
```

This code tells Postgres the following:

- The `SELECT` statement selects only the columns we want to view from each table.
- The `FROM` statement points to the first table to be joined, Departments (Table 1).
- `INNER JOIN` points to the second table to be joined, dept_manager (Table 2).
- `ON departments.dept_no = managers.dept_no;` indicates where Postgres should look for matches.

As Bobby is working through this first join, he realizes that he overlooked something important: start and end dates. Some of the folks from our

original list may not even work with the company anymore. Our original retirement_info table only included individuals with certain birth- and hire-dates—how many of these people have already left the company?

Use Left Join to Capture retirement-info Table

We'll need to help Bobby recreate this list. Think about what we need to have a fully accurate retirement_info table:

- Employee number
- Employee name (first and last)
- If the person is presently employed with PH

Which tables have this information? Our current retirement_info is already filtered to list only the employees born and hired within the correct time frame. The dept_emp table has the last bit we need. We'll need to perform a join to get this information into one spot. Let's get started.

First, we start with the `SELECT` statement.

```
-- Joining retirement_info and dept_emp tables
SELECT retirement_info.emp_no,
        retirement_info.first_name,
retirement_info.last_name,
        dept_emp.to_date
```

Next, we assign the left table with `FROM`.

```
FROM retirement_info
```

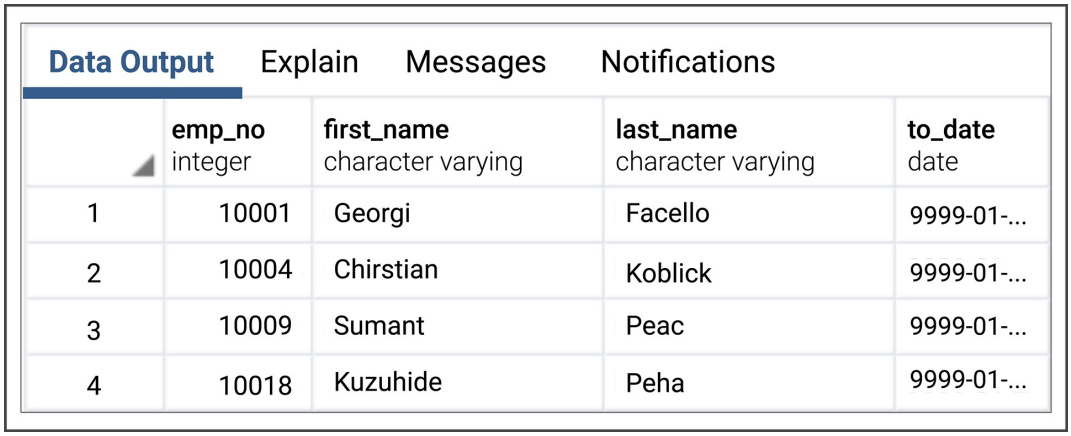
Then we specify the join we'll use. This time, use a `LEFT JOIN` to include every row of the first table (retirement_info). This also tells Postgres which table is second, or on the right side `(dept_emp)`.

```
LEFT JOIN dept_emp
```

Now we need to tell Postgres where the two tables are linked with the `ON` clause.

```
ON retirement_info.emp_no = dept_emp.emp_no;
```

Run the code to see what the data looks like. The "Data Output" tab should contain each of the four columns specified earlier: emp_no (employee number), first_name, last_name, and the to_date. Data from two different tables have been successfully merged into one!



Data Output | Explain | Messages | Notifications

	emp_no integer	first_name character varying	last_name character varying	to_date date
1	10001	Georgi	Facello	9999-01-...
2	10004	Chirstian	Koblick	9999-01-...
3	10009	Sumant	Peac	9999-01-...
4	10018	Kuzuhide	Peha	9999-01-...

Use Aliases for Code Readability

Joining tables can get messy. There are several different table and column name combinations to keep track of, and they can get lengthy as the query is created.

SQL has a method to shorten the code and provide greater readability by using an alias instead of a full tablename.

> **Important**
> An alias in SQL allows developers to give nicknames to tables. This helps improve code readability by shortening longer names into one-, two-, or three-letter temporary names. This is commonly used in joins because multiple tables and columns are often listed.

Practice on the join we performed earlier:

```
-- Joining retirement_info and dept_emp tables
SELECT retirement_info.emp_no,
        retirement_info.first_name,
retirement_info.last_name,
        dept_emp.to_date
FROM retirement_info
LEFT JOIN dept_emp
ON retirement_info.emp_no = dept_emp.emp_no;
```

Each table name can be shortened to a nickname (e.g., retirement_info becomes “ri”). Let’s start with updating the `SELECT` statement.

```
SELECT ri.emp_no,
        ri.first_name,
ri.last_name,
        de.to_date
```

This is already less cluttered. There are considerably fewer characters to type (and read) and less space is taken up in the editor. But how does

SQL know what the nickname refers to? We still need to define the new aliases, right? That takes place in the next two lines:

```
FROM retirement_info as ri
LEFT JOIN dept_emp as de
```

And we can continue using the aliases to finish the code.

```
ON ri.emp_no = de.emp_no;
```

These aliases only exist within this query and aren't committed to that database.

Update our other join. Go back to your query editor and locate this block of code:

```
-- Joining departments and dept_manager tables
SELECT departments.dept_name,
     dept_manager.emp_no,
     dept_manager.from_date,
     dept_manager.to_date
FROM departments
INNER JOIN dept_manager
ON departments.dept_no = dept_manager.dept_no;
```

Using the same alias method and syntax as before, rename departments to "d" and dept_manager to "dm."

Starting with the `SELECT` statement, update the table names.

```
SELECT d.dept_name,
     dm.emp_no,
```

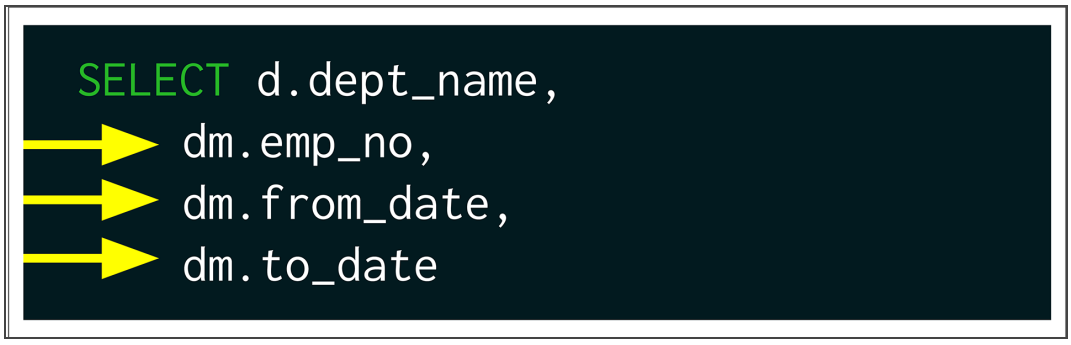
```
    dm.from_date,
    dm.to_date
```

That's much cleaner already. Continue with the last few lines.

```
FROM departments as d
INNER JOIN dept_manager as dm
ON d.dept_no = dm.dept_no;
```

Did you notice a slight difference in the spacing in our new tables? Instead of each table's name and column being stretched into a single line, they are now on their own lines. The spacing is similar to when we created tables, too.

> **Note**
> Joins aren't always completed instantly; the performance can vary based on different factors. For example, joining more than three tables will be slower than joining two tables. The size of the datasets being joined is another factor to consider. Even the physical design of the table can play a part!



This is all to help with code readability. Not only is the code neat and tidy, but it's best practice to maintain the syntax like this when there is more than one column listed in the `SELECT` statement.

Use Left Join for retirement_info and dept_emp tables

Now that we have a list of all retirement-eligible employees, it's important to make sure that they are actually still employed with PH. To do so, we're going to perform another join, this time between the retirement_info and dept_emp tables. The basic information to include in the new list is:

- Employee number
- First name
- Last name
- To-date

In the pgAdmin query editor, let's begin by specifying these columns and tables.

```
SELECT ri.emp_no,
        ri.first_name,
        ri.last_name,
de.to_date
```

Next, we need to create a new table to hold the information. Let's name it "current_emp."

```
INTO current_emp
```

The next step is to add the code that will join these two tables.

```
FROM retirement_info as ri
LEFT JOIN dept_emp as de
ON ri.emp_no = de.emp_no
```

Finally, because this is a table of current employees, we need to add a filter, using the WHERE keyword and the date 9999-01-01.

```
WHERE de.to_date = ('9999-01-01');
```

When this block of code is executed, a new table containing only the current employees who are eligible for retirement will be returned.

7.3.4: Use Count, Group By, and Order By

There is a lot that goes into using joins, but practice certainly helps. Learning aliases has made our code cleaner, too. Thanks to the extra lists we've been working on, Bobby understands more about how joins work in SQL. He can put that knowledge into action by joining the correct tables to create new lists.

But before he's able to hand that list over, he'll need to use `COUNT` and `GROUP BY` with joins to separate the employees into their departments. These two components are very similar to functions used in Pandas. `COUNT` will count the rows of data in a dataset, and we can use `GROUP BY` to group our data by type. If the boss were to ask Bobby how many employees are retiring from the Sales department, we would use both of these functions together with joins to generate an answer.

We can also arrange the data so it presents itself in descending or ascending order by using another function: `ORDER BY`. This is another function that appears in Pandas and works the same way.

Let's continue to work on these lists with Bobby, this time using `COUNT`, `GROUP BY`, and `ORDER BY` with the joins.

We know that we'll need to use some joins to organize the data we need in one table. We also know that Bobby needs a count of employees for each department.

To organize the counts, we'll need to add a `GROUP BY` clause to our select statement. In Postgres, `GROUP BY` is used when we want to group rows of identical data together in a table. It is precisely the clause we want to group separate employees into their departments.

In your query editor, join the current_emp and dept_emp tables. The new code should be added at the bottom, below the existing code.

```
-- Employee count by department number
SELECT COUNT(ce.emp_no), de.dept_no
FROM current_emp as ce
LEFT JOIN dept_emp as de
ON ce.emp_no = de.emp_no
GROUP BY de.dept_no;
```

A few things to note:

- The `COUNT` function was used on the employee numbers.
- Aliases were assigned to both tables.
- `GROUP BY` was added to the `SELECT` statement.

We added `COUNT()` to the `SELECT` statement because we wanted a total number of employees. We couldn't actually use the `SUM()` function because the employee numbers would simply be added, which would leave us with one really large and useless number.

Bobby's boss asked for a list of how many employees per department were leaving, so the only columns we really needed for this list were the employee number and the department number.

We used a `LEFT JOIN` in this query because we wanted all employee numbers from Table 1 to be included in the returned data. Also, if any employee numbers weren't assigned a department number, that would be made apparent.

The `ON` portion of the query tells Postgres which columns we're using to match the data. Both tables have an emp_no column, so we're using that to match the records from both tables.

`GROUP BY` is the magic clause that gives us the number of employees retiring from each department.

How many employees will retire from Department No. 006 (d006)?

- ○ 2,432
- ○ 2,342
- ○ 2,234
- ○ 2,818

Check Answer

Finish ▸

Did you notice that the data output isn't in any particular order? In fact, if you executed the code again, the same numbers would be returned in another order altogether. Thankfully, there is one additional clause we can add to the query to keep everything in order: `ORDER BY`.

`ORDER BY` does exactly as it reads: It puts the data output in order for us. Let's update our code. In the query editor, add the `ORDER BY` line to the end of the code block, so your code looks like the following:

```
-- Employee count by department number
SELECT COUNT(ce.emp_no), de.dept_no
FROM current_emp as ce
LEFT JOIN dept_emp as de
ON ce.emp_no = de.emp_no
GROUP BY de.dept_no
ORDER BY de.dept_no;
```

Now, instead of a randomly ordered output each time, it will be organized by the department number.

	count bigint	**dept_no** character varying (4)
1	2199	d001
2	1908	d002
3	1953	d003
4	8174	d004
5	9281	d005
6	2234	d006
7	5860	d007
8	2413	d008
9	2597	d009

Now test your skills with the following Skill Drill.

Skill Drill

Bobby can present this table to his boss since it provides a breakdown for each department. Update the code block to create a new table, then export it as a CSV.

7.3.5: Create Additional Lists

So far, we've completed several joins, become far more familiar with aliases, and we've also started incorporating other functions into our queries. This is awesome because we're able to generate summary data for Bobby's supervisor. All of those rows of data compacted into a few lines, and presented in descending order? That's really great work and we've accomplished a lot.

Bobby's supervisor has been really impressed with the work completed so far. He's even thought of a few more lists he'd like to see. These lists will require queries that filter and order the data and even join more than one table together.

With these lists, we'll continue to work with all of the SQL skills we've been developing and, like a puzzle, put them all together to build the exact queries we need to create the requested lists.

Because of the number of people leaving each department, the boss has requested three lists that are more specific:

1. **Employee Information:** A list of employees containing their unique employee number, their last name, first name, gender, and salary
2. **Management:** A list of managers for each department, including the department number, name, and the manager's employee number, last name, first name, and the starting and ending employment dates
3. **Department Retirees:** An updated current_emp list that includes everything it currently has, but also the employee's departments

Get started with the first list.

List 1: Employee Information

The first requested list is general employee information, but with their current salaries included. Let's look at our ERD again.

The Employees table has all of the information we need and uses the emp_no column as the primary key. The Salaries table has the additional information we need as well as the same primary key. The only problem is that the Employees table holds data for all employees, even the ones who

are not retiring. If we use this table, we will have a far bigger list to present than expected.

To include all of the information Bobby's manager wants, we'll need to create an entirely new table from the beginning. Here's everything we need:

1. Employee number
2. First name
3. Last name
4. Gender
5. to_date
6. Salary

According to our ERD, the Salaries table has a to_date column in it. Let's make sure it aligns with the employment date or something else. Run a `SELECT` statement in the query editor to take a look.

```
SELECT * FROM salaries;
```

These dates are all over the place. We want to know what the most recent date on this list is, so let's sort that column in descending order. Back in the query editor, modify our select statement as follows:

```
SELECT * FROM salaries
ORDER BY to_date DESC;
```

	emp_no integer	**salary** integer	**from_date** date	**to_date** date
1	57279	84427	2000-02-01	2000-01-...
2	54420	66835	2000-02-01	2000-01-...

3	27628	40000	2000-02-01	2000-01-...
4	40179	56806	2000-02-01	2000-01-...
5	100364	40000	2000-02-01	2000-01-...

That looks a little better, but what's wrong with the date? It's certainly not the most recent date of employment, so it must have something to do with salaries. Looks like we'll need to pull employment dates from the dept_emp table again.

Now that we know what data we need from which tables, we can get started. It doesn't have to be from scratch, though. We've already created code to filter the Employees table to show only employes born and hired at the correct time, so let's look at our query editor to see if we can reuse that code.

```
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');
```

This looks promising because the age and hiring filters are already in place, the only thing we're missing is the gender. Add that to our `SELECT` statement and also format it to fit within the best practices guidelines.

```
SELECT emp_no,
        first_name,
last_name,
        gender
```

We won't want to save this query into the same table we used before. Not only would it be confusing, but Postgres wouldn't allow it anyway. We'll

want to update the `INTO` portion. The rest of the code looks good, as we want the same filters to be in place, so leave it as-is.

```
INTO emp_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');
```

Now that our employees table has been filtered again and is being saved into a new temporary table (emp_info), we need to join it to the salaries table to add the to_date and Salary columns to our query. This will require a join, so let's get started. First, update the `SELECT` statement by adding the two columns we need from the Salaries table. Remember to use aliases to make it easier to read.

```
SELECT e.emp_no,
        e.first_name,
e.last_name,
        e.gender,
        s.salary,
        de.to_date
```

All columns are accounted for. We already know we're naming our new table emp_info, so we can leave the `INTO` statement as-is. Let's move on to the joins. In this case, we'll use inner joins in our query. This is because we want only the matching records. Back in the query editor, update the `INTO` and `FROM` lines, then add the first join directly below.

```
INTO emp_info
FROM employees as e
INNER JOIN salaries as s
ON (e.emp_no = s.emp_no)
```

Up to this point, we have updated and added code to:

- Select columns from three tables
- Create a new temp table
- Add aliases
- Join two of the three tables

Adding a third join seems tricky, but thankfully, the syntax is exactly the same. All we need to do is add the next join right under the first. Back in the query editor, add the following:

```
INNER JOIN dept_emp as de
ON (e.emp_no = de.emp_no)
```

Almost there! We have all of the joins, but we still need to make sure the filters are in place correctly. The birth and hire dates are still resting right under our joins, so update that with the proper aliases. For more on how to join two or more tables, refer to **Joining More than Two Tables (http://www.postgresqltutorial.com/postgresql-inner-join/)** .

```
WHERE (e.birth_date BETWEEN '1952-01-01' AND '1955-12-31')
     AND (e.hire_date BETWEEN '1985-01-01' AND '1988-12-31')
```

Okay, now we have joined all three tables and have updated the birth and hire date filters to reference the correct table using an alias. We have one more filter to add, then we're ready to check and export the data.

The last filter we need is the to_date of 999-01-01 from the dept_emp table. To add another filter to our current `WHERE` clause, we will use `AND` again. In the query editor, add this last line:

```
    AND (de.to_date = '9999-01-01');
```

Before we create a temporary table using this code, comment out the `INTO` line so that we don't run it with the rest. This way, we'll be able to see the results of our code immediately. This is useful because if there is a mistake, we won't need to backtrack and delete the table.

Highlight the `INTO` emp_info line and press Command + forward slash, / (for Mac), or CTRL + forward slash, / (for Windows). This will automatically add the double hyphen to indicate a comment. Now highlight the entire block and run the code.

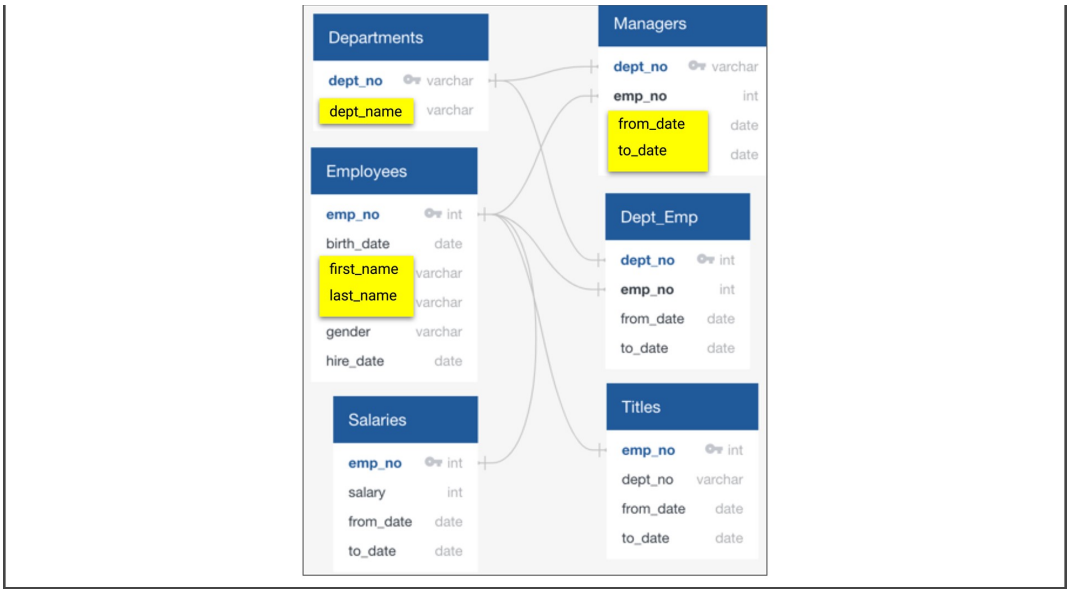
The results are looking good and the list contains everything Bobby's boss requested. Let's uncomment `INTO` and run the code again to save the temporary table. Remember to export this list as a CSV into your current project folder.

Those salaries still look a little strange, though. Bobby will need to ask his manager about the lack of employee raises.

List 2: Management

The next list to work on involves the management side of the business. Many employees retiring are part of the management team, and these positions require training, so Bobby is creating this list to reflect the upcoming departures.

This list includes the manager's employee number, first name, last name, and their starting and ending employment dates. Look at the ERD again and see where the data we need resides.



We can see that the information we need is in three tables: Departments, Managers, and Employees. Remember, we're still using our filtered Employees table, current_emp, for this query.

Let's do this one together. At the bottom of the query editor, type the following:

```
-- List of managers per department
SELECT  dm.dept_no,
        d.dept_name,
        dm.emp_no,
        ce.last_name,
        ce.first_name,
        dm.from_date,
        dm.to_date
INTO manager_info
FROM dept_manager AS dm
    INNER JOIN departments AS d
        ON (dm.dept_no = d.dept_no)
    INNER JOIN current_emp AS ce
        ON (dm.emp_no = ce.emp_no);
```

This is pretty similar to the last set of joins we completed. Like last time, comment out the `INTO` line before we run the code for the first time.

	dept_no character varying (4)	dept_name character varying (40)	emp_no integer	last_name character varying	first_name character varying	from_date date	to_date date
1	d003	Human Resources	110183	Ossenbruggen	Shirish	1985-01-01	1992-03-...
2	d004	Production	110386	Kieras	Shem	1992--08-02	1996-08-...
3	d007	Sales	111133	Zhang	Hauke	1991-03-07	9999-01-...
4	d008	Research	111534	Kambil	Hilary	1991-04-08	9999-01-...
5	d009	Customer Service	111692	Butterworth	Tonny	1985-01-01	1988-10-...

The result of this query looks even more strange than the salaries. How can only five departments have active managers? This is another question Bobby will need to ask his manager.

List 3: Department Retirees

The final list needs only to have the departments added to the current_emp table. We've already consolidated most of the information into one table, but let's look at the department names and numbers we'll need.

The Dept_Emp and Departments tables each have a portion of the data we'll need, so we'll need to perform two more joins in the next query.

We'll use inner joins on the current_emp, departments, and dept_emp to include the list of columns we'll need to present to Bobby's manager:

1. emp_no
2. first_name
3. last_name
4. dept_name

In the query editor, begin with the `SELECT` statement. Type the following:

```
SELECT ce.emp_no,
ce.first_name,
ce.last_name,
d.dept_name
```

Notice we have only selected four columns from two tables, yet there are three tables in the ERD that we need. That's because we don't need to see a column from each table in a join, but we do need the foreign and primary keys to link them together. Continue with the `INTO` statement, this time naming the temporary table dept_info.

```
-- INTO dept_info
```

We should go ahead and comment it out now because we know we'll want to test the code before saving it as a table.

Next, start defining the aliases with `FROM` and the joins. In the query editor, type the following:

```
FROM current_emp as ce
INNER JOIN dept_emp AS de
ON (ce.emp_no = de.emp_no)
INNER JOIN departments AS d
ON (de.dept_no = d.dept_no);
```

After executing the code and checking the results, a few folks are are appearing twice. Maybe they moved departments? It's interesting how each list has given Bobby a question to ask his manager. So far, Bobby would like to know the following:

1. What's going on with the salaries?
2. Why are there only five active managers for nine departments?
3. Why are some employees appearing twice?

To help Bobby find these answers, we're going to create tailored lists.

7.3.6: Create a Tailored List

Even more joins have been completed and the lists delivered to management. Everyone is impressed with what we've been able to create. Think of it: We started with only six CSV files and no real database or data management system in place. Now, we have created a model of the database with an ERD, imported data, and to tie it all together, we have performed many queries to help PH future-proof the company. That is quite a lot of work we've gotten done.

After Bobby's lists have been passed to the department supervisors to begin their future-proofing preparation, one manager has asked for an additional list. This list will be created using the same tools we've been working with so far: queries using filters, joins, and functions.

> The department head for Sales was a little surprised at how many folks will be leaving, so has asked for an additional list, containing only employees in their department. The new list Bobby will need to make will

contain everything in the retirement_info table, only tailored for the Sales team.

> **Skill Drill**
> Create a query that will return only the information relevant to the Sales team. The requested list includes:
>
> - Employee numbers
> - Employee first name
> - Employee last name
> - Employee department name

The same manager asking for a list of retiring employees has asked for a list of employees in both the Sales and Development departments because, together, both managers want to try a new mentoring program for employees getting ready to retire. Instead of having a large chunk of their workforce retiring, they want to introduce a mentoring program: experienced and successful employees stepping back into a part-time role instead of retiring completely. Their new role in the company would be as a mentor to the newly hired folks. Before they can present their idea to the CEO, they'd like to have an idea of how many people between the departments they would need to pitch their idea to.

The new query should return the same information as the last, only with the combined departments.

> **Skill Drill**
> Create another query that will return the following information for the Sales and Development teams:
>
> - Employee numbers

- Employee first name
- Employee last name
- Employee department name

Hint: You'll need to use the `IN` condition with the `WHERE` clause. See the **PostgreSQL documentation (https://www.techonthenet.com/postgresql/in.php)** for additional information.

The `IN` condition is necessary because you're creating two items in the same column.

Module 7 Challenge

Submit Assignment

Due Mar 1 by 11:59pm **Points** 100
Submitting a text entry box or a website url

> All of Bobby's hard work has paid off. The database modeling is clean and clear. The relationships that Bobby mapped out with the ERD helped with creating all of the different queries he worked through. Speaking of queries, the lists Bobby produced generated much enthusiasm in the office!
>
> In fact, everyone is so impressed that they requested a few more lists. Also, the HR Director got wind of the great work that has been completed and would like an additional list of employees who would be good candidates for a supervisory role. Ideally, these candidates would be born in 1965.

In this challenge, you will use advanced queries and joins to create a list of candidates for the mentorship program. To complete this task, you'll use your knowledge of aliasing, filtering, and creating new tables.

Background

To create the new list of potential mentors, you will need to create a query that returns a list of current employees eligible for retirement, as well as their most recent titles. To get the final list with the recent titles, you'll also need

to partition the data so that each employee is only included on the list once. In addition, you'll need to perform a query that shows how many current employees of each title are presently eligible for retirement. The final query should return the potential mentor's employee number, first and last name, their title, birth date and employment dates.

Objectives

The goals of this challenge are for you to:

- Use an ERD to understand relationships between SQL tables.
- Create new tables in pgAdmin by using different joins.
- Write basic- to intermediate-level SQL statements.
- Export new tables to a CSV file.

Part 1 Instructions

Generate the following tables:

Number of [titles] Retiring

1. Create a new table using an `INNER JOIN` that contains the following information:
    - Employee number
    - First and last name
    - Title
    - from_date
    - Salary

2. Export the data as a CSV.

Reference your ERD to help determine which tables you'll use to complete this join.

Only the Most Recent Titles

Exclude the rows of data containing duplicate names. (This is tricky.) **Hint:** Refer to [Partitioning Your Data (https://blog.theodo.fr/2018/01/search-destroy-duplicate-rows-postgresql/)](https://blog.theodo.fr/2018/01/search-destroy-duplicate-rows-postgresql/) for help.

1. In descending order (by date), list the frequency count of employee titles (i.e., how many employees share the same title?).
2. Export the data as a CSV.

Who's Ready for a Mentor?

1. Create a new table that contains the following information:
    - Employee number
    - First and last name
    - Title
    - from_date and to_date

    Note: The birth date needs to be between January 1, 1965 and December 31, 1965. Also, make sure only current employees are included in this list.

2. Export the data as a CSV.

Part 2 Instructions

Create a technical report in markdown format that contains the following:

- Brief project summary
- PNG of your ERD
- Code for the requested queries, with examples of each output

When writing your project summary, use your own words to describe the purpose of the queries used (for example, determining how many employees are retiring) and what you observed through your results.

Submission

For this set of queries, paste the code into a `challenge.sql` file in your repo folder. Add, commit, and push these new files to your repository.

Remember to include the following:

1. `README.md` file containing your technical report.This file needs to be in the main folder of your repository; this way, it will serve as a homepage to your repo when you share the link.
2. `challenge.sql` file containing your queries
3. CSVs created from your exported data

Submit the link to your repository through Canvas.

Module 7 Detailed Rubric

Note: You are allowed to miss up to two Challenge assignments and still earn your certificate. If you complete all Challenge assignments, your lowest two grades will be dropped. If you wish to skip this assignment, click Submit then indicate you are skipping by typing “I choose to skip this assignment” in the text box.

Criteria	Ratings					Pts
Written Report Please see detailed description in the rubric pdf hyperlinked in the assignment description.	**30.0 pts** **Mastery** Presents a cohesive written report that includes all of the elements described in the detailed rubric.	**25.0 pts** **Approaching Mastery** Presents a cohesive written report that includes 3 of the items listed in the detailed rubric.	**20.0 pts** **Progressing** Presents a developing written report that includes 3 of the elements described in the detailed rubric.	**10.0 pts** **Emerging** Presents a limited written report that includes the elements listed in the detailed rubric.	**0.0 pts** **Incomplete** No submission was received -OR- Submission was empty or blank - OR- Submission contains evidence of academic dishonesty	30.0 pts
Use an ERD to understand relationships between SQL tables Please see detailed description in the rubric pdf hyperlinked in the assignment description.	**22.0 pts** **Mastery** The Entity Relationship Diagram includes all correct tables and all of the items listed in the detailed rubric.	**17.0 pts** **Approaching Mastery** The Entity Relationship Diagram includes all correct tables and at least three of the items listed in the detailed rubric.	**12.0 pts** **Progressing** The Entity Relationship Diagram may omit tables, but includes at least 2 of the items listed in the detailed rubric.	**5.0 pts** **Emerging** The Entity Relationship Diagram may omit tables, but includes at least 1 of the items listed in the detailed rubric.	**0.0 pts** **Incomplete** No submission was received - OR- Submission was empty or blank - OR- Submission contains evidence of academic dishonesty	22.0 pts
Create new tables in pgAdmin by using different joins Please see detailed description in the rubric pdf hyperlinked in the	**22.0 pts** **Mastery** All table schemas run without error and include all of the items listed in the	**17.0 pts** **Approaching Mastery** 4-5 table schemas run without error and include at least 4 the items listed in the detailed rubric.	**12.0 pts** **Progressing** 2-3 table schemas run without error and include at least 2 of the items listed in the detailed rubric.	**5.0 pts** **Emerging** 1 table schema runs without error and include at least 1 of the items listed in the detailed rubric.	**0.0 pts** **Incomplete** No submission was received -OR- Submission was empty or blank - OR- Submission contains evidence of	22.0 pts

Criteria	Ratings					Pts
assignment description.	detailed rubric.				academic dishonesty	
Write basic-to intermediate-level SQL statements Please see detailed description in the rubric pdf hyperlinked in the assignment description.	**22.0 pts** **Mastery** Queries provided for all questions; all queries provide the expected results; all queries run without error	**17.0 pts** **Approaching Mastery** Queries provided for all questions, but queries provide the expected results, with non-efficient code, or they do not provide the expected result	**12.0 pts** **Progressing** Queries not provided for all of the questions; 3 queries do not provide the expected result (including errors)	**5.0 pts** **Emerging** Queries not provided for all of the questions and 4 or more do not provide the expected result (including errors) -OR- No queries provided.	**0.0 pts** **Incomplete** No submission was received -OR- Submission was empty or blank -OR- Submission contains evidence of academic dishonesty	22.0 pts
Export new tables to a CSV file Please see detailed description in the rubric pdf hyperlinked in the assignment description.	**4.0 pts** **Mastery** New tables are exported to a CSV file, with no errors.	**3.0 pts** **Approaching Mastery** New tables are exported to a CSV file, with one or two minor errors.	**2.0 pts** **Progressing** New tables are exported to a CSV file, with significant errors.	**1.0 pts** **Emerging** New tables are not exported to a CSV file.	**0.0 pts** **Incomplete** No submission was received -OR- Submission was empty or blank -OR- Submission contains evidence of academic dishonesty	4.0 pts
						Total Points: 100.0

	Mastery	Approaching Mastery	Progressing	Emerging	Incomplete
Written Report (30 points)	Presents a cohesive written report that includes the following: Summary of the results ✓ number of individuals retiring ✓ number of individuals being hired ✓ number of individuals available for mentorship role Additionally, the summary should include ✓ one recommendation for further analysis on this data set	Presents a cohesive written report that includes three of the four following items: Summary of the results ✓ number of individuals retiring ✓ number of individuals being hired ✓ number of individuals available for mentorship role Additionally, the summary should include ✓ one recommendation for further analysis on this data set	Presents a developing written report that includes three of the four following items: Summary of the results ✓ number of individuals retiring ✓ number of individuals being hired ✓ number of individuals available for mentorship role Additionally, the summary should include ✓ one recommendation for further analysis on this data set	Presents a limited written report that includes the following items: Summary of the results ✓ number of individuals retiring ✓ number of individuals being hired ✓ number of individuals available for mentorship role	No submission was received -OR- Submission was empty or blank -OR- Submission contains evidence of academic dishonesty
Use an ERD to understand relationships between SQL tables (22 points)	The Entity Relationship Diagram includes all correct tables and all of the following: ✓ Each table has the correct column names ✓ Each table has the correct corresponding data types ✓ Primary Keys set for each table ✓ Tables are correctly related using Foreign Keys	The Entity Relationship Diagram includes all correct tables and at least 3 of the following: ✓ Each table has the correct column names ✓ Each table has the correct corresponding data types ✓ Primary Keys set for each table ✓ Tables are correctly related using Foreign Keys	The Entity Relationship Diagram may omit tables, but includes at least 2 of the following: ✓ Each table has the correct column names ✓ Each table has the correct corresponding data types ✓ Primary Keys set for each table ✓ Tables are correctly related using Foreign Keys	The Entity Relationship Diagram may omit tables, but includes at least 1 of the following: ✓ Each table has the correct column names ✓ Each table has the correct corresponding data types ✓ Primary Keys set for each table ✓ Tables are correctly related using Foreign Keys -OR- ✓ No Entity Relationship Diagram is included.	
Create new tables in pgAdmin by using different joins (22 points)	All table schemas run without error and include all of the following: ✓ All required columns ✓ Columns are set to the correct data type ✓ Primary key for each table ✓ Correctly references related tables using Foreign Keys ✓ Correctly uses `NOT NULL` condition on necessary columns ✓ Accurately defines value length for columns	4-5 table schemas run without error and include at least 4 of the following: ✓ All required columns ✓ Columns are set to the correct data type ✓ Primary key for each table ✓ Correctly references related tables using Foreign Keys ✓ Correctly uses `NOT NULL` condition on necessary columns ✓ Accurately defines value length for columns	2-3 table schemas run without error and include at least 2 of the following: ✓ All required columns ✓ Columns are set to the correct data type ✓ Primary key for each table ✓ Correctly references related tables using Foreign Keys ✓ Correctly uses `NOT NULL` condition on necessary columns ✓ Accurately defines value length for columns	1 table schema runs without error and include at least 1 the following: ✓ All required columns ✓ Columns are set to the correct data type ✓ Primary key for each table ✓ Correctly references related tables using Foreign Keys ✓ Correctly uses `NOT NULL` condition on necessary columns ✓ Accurately defines value length for columns -OR-	

				✓ No table schemas created or the provided schemas do not run	
Write basic- to intermediate-level SQL statements (22 points)	Queries provided for all questions and: ✓ All queries provide the expected results ✓ All queries run without error	Queries provided for all questions but: ✓ Queries provide the expected results, with non-efficient code. -OR- ✓ 1-2 queries do not provide the expected result (including errors)	✓ Queries not provided for all of the questions -AND- ✓ 3 queries do not provide the expected result (including errors)	✓ Queries not provided for all of the questions -AND- ✓ 4 or more do not provide the expected result (including errors) -OR- ✓ No queries provided	
Export new tables to a CSV file (4 points)	New tables are exported to a CSV file, with no errors.	New tables are exported to a CSV file, with one or two minor errors.	New tables are exported to a CSV file, with significant errors.	New tables are not exported to a CSV file.	

Module 7 Career Connection

Increase Your Visibility in the Industry

Introduction:

Employer Competitive candidates stand out in their search by making themselves visible and selling their strengths. They're clear on their search goals, the value they're able to add to their target jobs, and the importance of increasing their visibility to help them reach those goals.

Now that you're clear on your professional story, you can put it to use towards building your visibility. View the guide at Milestone 3 for best practices on in-person and online networking.

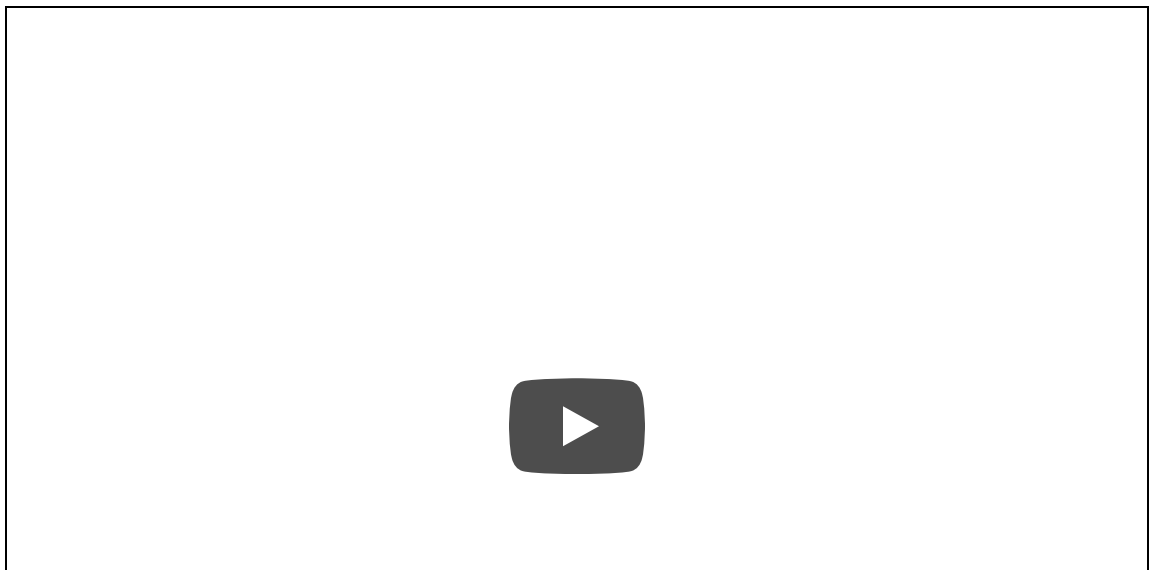
Career Services Next Step: [Link Milestone 3 (https://courses.bootcampspot.com/courses/138/pages/milestone-3-build-your-visibility)](https://courses.bootcampspot.com/courses/138/pages/milestone-3-build-your-visibility)

Milestone 3: Build Your Visibility

"80% of jobs never get posted and are only found through networking."
– The Muse (https://www.themuse.com/advice/6-insider-job-search-facts-thatll-make-you-rethink-how-youre-applying)

Key Takeaways

- By the end of this Milestone, you will be able to develop a plan for networking and outreach.
- Review this guide, and mark it complete.
- For more insight on networking, view the "Expand Your Network" workshop below.



Employer Competitive candidates stand out in their search by making themselves visible and selling their strengths. They're clear on their search **goals**, the **value** they're able to add to their target jobs, and the importance of **increasing their visibility** to help them reach those goals.

For a guided experience on networking, see the video above.

Getting Started

1. Review the guide below for best practices on in-person and online networking.
2. If you'd like additional support with networking, see the resources we've included at the end of this guide.
3. Once you're done, mark this milestone complete. Reminder: This Milestone **does not** require a submission to a Profile Coach.

About Networking

Successful networking involves building relationships, and it's important to note, that building a strong network doesn't happen overnight. It takes time and consistent effort, but it will be worth it!

Networking IS....

- A chance to learn more about an industry and what employers are looking for
- A chance to gain visibility and make new connections

- A chance to gain confidence in your ability to describe your interests, skills, values (which will help you in a real interview)

Networking IS NOT...

- Asking for a job
- An interview for employment
- A guarantee of employment or employability
- Just a business card swap at a meeting or conference
- Lots of connections on LinkedIn

In-Person Networking

(1) Identify Your Current Network
You may have more people in your network than you realize. Use the categories below to help you think about who's in your network and how you might reach out to them.

    - Family
    - Friends
    - Former Colleagues/ Supervisors
    - Professors, Trainers, Etc.
    - Other Connections

(2) Expand your Network
Consider the following approaches to expand your network:

    - **Reach out to every person on your networking list above**, and send them your materials with a specific ask. “Asks” can include a quick chat on the phone for advice or a lunch date to talk

about your target industry as well as recommendations for who you should connect with next.

- If you're currently employed, **ask your boss for projects that require you to interact with new departments or individuals**. For example, you can propose that you help the company enhance its website, and in doing so, you'll interact with other developers and/or the marketing department.
- **Find volunteer opportunities**. Get involved in an organization or group that interests you, and offer to contribute some of your new tech skills. You may meet people who can be helpful.
- **Create business cards** that include your target role, links to Github, LinkedIn, and a QB code to scan for your resume.
- Continue to use **LinkedIn weekly to connect with employees and decision makers**. Look for people who might have secondary connections to you. Send personal messages about your passions and common interests, and request informational interviews.
- **Here is a great reminder of all the many places where you can network. (http://www.jobmonkey.com/best-places-to-network/)**

(3) Attend Networking Events

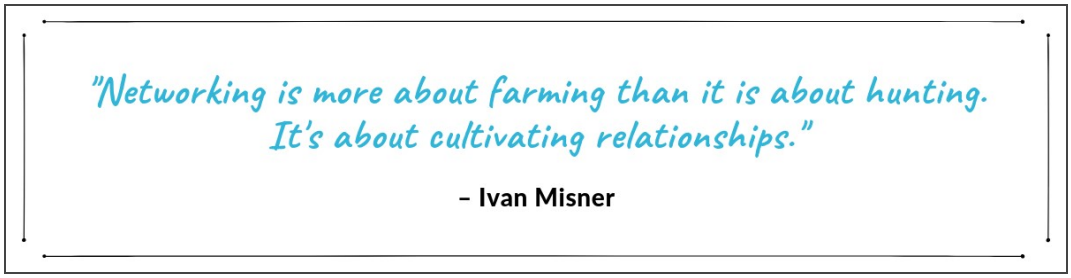
It's always helpful to set a goal when attending networking events (e.g. "I will have 3 meaningful conversations that may lead to potential follow-up," or "I will not leave until I have entered into at least 5 conversations."). Establishing a goal allows you to set a measurable standard of success for the event, which can help change your experience of networking into a positive one.

TIP: Always bring business cards with you to events. On the back of the cards you receive, take notes about the person you're speaking with so that you can follow up in a personal way.

Here are some additional tips that will help you differentiate yourself at an event:

1. **Master the use of tech language** – The better your vocabulary (especially as it relates to your industry), the more impressed people will be. Being confident, articulate, and knowledgeable will help you create a strong first impression.
2. **Eye Contact** – Always maintain eye contact when you're speaking with someone. Looking away can make you appear less confident. Also, remember to smile.
3. **Leave personal space** – Don’t stand too close to anyone. Keep a reasonable distance.
4. **Acknowledge your understanding** – When someone else is talking, acknowledge that you heard them with non-verbal body language such as nodding.
5. **Wait your turn** – Successful professionals are also good listeners. Allow your new connection the opportunity to complete their thoughts before offering a response.
6. **Watch body language** – Mirror the body language of the person with whom you are interacting. If they sit down, you should sit down too—they may be ready for a longer conversation. Try not to cross or fold your arms, as that may create the appearance that you are guarded. Overall, be mindful of both you and your new connection's body language.
7. **Be curious** – Open the conversation with questions. Focus on the other person’s interests first, and show genuine interest.

Here are some conversation starters that might help you as well. (https://www.themuse.com/advice/30-brilliant-networking-conversation-starters)



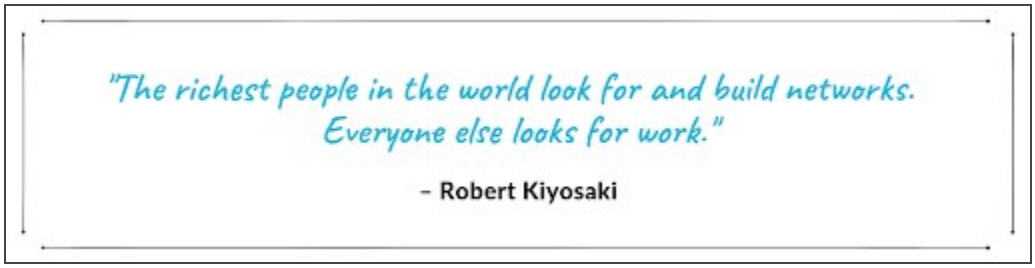
(4) Follow-Up

Networking only works if you follow up! After meeting new contacts, follow up with a personal message soon after you've met.

When reaching out, whether via email or phone, here are a few tips:

1. **Remind them how they know you**. Always begin by referencing a common person, event, educational experience, work experience, organization, or award that creates a common bond.
2. **Be clear on what you bring to the table**. Express interest in the person's work, and add value instead of asking for something. Sharing interesting articles, making introductions to helpful contacts, supporting the contact's endeavors, and engaging with their LinkedIn posts are great ways to add value.
3. **Be flexible with scheduling**. Make it easy and convenient for the contact to say yes to connecting again!
4. **Do your homework!** Research your new connections to help you better foster a relationship with them. LinkedIn and general internet searches provide instant access to information on your targeted connections.

5. **Don’t give up, and don’t take it personally**. Some people hesitate to reach out again for fear of being ignored, rejected, or of being a pest. It’s okay if someone doesn’t take you up on your offer. If you are reaching out to people regularly, you'll get more accepted invitations than passes.
6. **Breathe, and stay calm**. It’s perfectly normal to be nervous about calling people. Networking is a skill that requires practice. It may help to practice your calls with friends or family. It may also help to remember that you're not calling to ask for favors — you are asking to learn from someone. Most people love sharing their expertise!



(5) Request Informational Interviews & Seek Mentors

Informational interviews are your opportunity to explore whether a your goals or current opportunities really are the right match for you. They're also great ways to expand your network through introductions from the connection you're interviewing.

Before the Interview

DO

- Research the individual — use LinkedIn, Google them, or personal connections to prepare.
- Prepare a list of questions (at least 4).
- Review a list of conversation starters for informational interviews, and have a few ready to go.

- Be ready to deliver your elevator pitch.

DO NOT

- Plan to “wing it” — while these are not job interviews, preparation is needed.
- Script every second of the interview — you need to build a relationship as well.
- Assume this person is going to lead the conversation or listen to you talk the entire time.

During the Interview

DO

- Smile, be aware of appropriate eye contact, and lean forward.
- Ask questions to demonstrate interest and active listening.
- Find a personal connection through interests, passions, or hobbies.
- Listen for ways you may be able to help or volunteer for them.
- Use varied tones and volumes to demonstrate your passion and enthusiasm.
- Describe work you have done that might be interesting.

DO NOT

- Complain about previous employers or peers.
- Dominate conversation — be sure to let them talk.
- Answer questions with one word answers — be concise, but be thorough too.
- Look at your phone during the conversation.

After the Interview

DO

- Jot down notes to remember the conversation.
- Write a thank you email.
- Follow up about once a month with updates and check ins.

DO NOT

- Follow up too frequently (more than about once per month).
- Text a thank you — this should be a more formal thank you.

NOTE: Informational interviews should lead to more interviews, volunteer or open-source projects, or ideas about new directions to take. For more on informational interviews, check out this **article called 5 Tips for Non-Awkward Informational Interviews (https://www.themuse.com/advice/5-tips-for-nonawkward-informational-interviews)** .

Online Networking

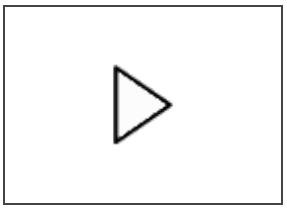
SOCIAL MEDIA

- Make sure your profiles look polished on platforms like LinkedIn, Angel.co, and any others. Examples of excellent Data profiles can be found here:
  - **https://www.linkedin.com/in/robinhchoi/ (https://www.linkedin.com/in/robinhchoi/)**
  - **https://www.linkedin.com/in/carlosmarin2/ (https://www.linkedin.com/in/carlosmarin2/)**
  - **https://www.linkedin.com/in/dylansather/ (https://www.linkedin.com/in/dylansather/)**

    - https://www.linkedin.com/in/leonardo-apolonio/ (https://www.linkedin.com/in/leonardo-apolonio/)
- On LinkedIn, Facebook, and other platforms, follow companies, thought leaders, and professionals in the industry. Learn how to do this here: http://bit.ly/2Ec5ikA (http://bit.ly/2Ec5ikA). Engage with these companies and individuals through likes and comments on their posts.
- Look for alumni groups for your current or past organizations.
- Use these templates to help you draft your outreach messages. (http://bit.ly/2vRZj24)

Additional Resources

- 10 Simple Ways to Improve Your Networking Skills (https://www.youtube.com/watch?v=E5xTbn6OnAA)



(https://www.youtube.com/watch?v=E5xTbn6OnAA)

- How to Find Your Next Job Over Coffee (https://blog.udacity.com/2014/11/informational-interviews-how-to-find.html)
- Visit the 'Build Your Visibility' section of your Career Services Resource Library (https://legacy.gitbook.com/read/book/cstrilogy/career-resources-data-analytics-library?key=datastudents) – for outreach templates, potential jobs, and additional networking resources.