1

CREATING YOUR FIRST DATABASE AND TABLE

SQL is more than just a means for extracting knowledge from data. It's also a language for *defining* the structures that hold data so we can organize *relationships* in the data. Chief among those structures is the table.

A table is a grid of rows and columns that store data. Each row holds a collection of columns, and each column contains data of a specified type: most commonly, numbers, characters, and dates. We use SQL to define the structure of a table and how each table might relate to other tables in the database. We also use SQL to extract, or *query*, data from tables.

Understanding tables is fundamental to understanding the data in your database. Whenever I start working with a fresh database, the first thing I do is look at the tables within. I look for clues in the table names and their column structure. Do the tables contain text, numbers, or both? How many rows are in each table?

Next, I look at how many tables are in the database. The simplest database might have a single table. A full-bore application that handles

customer data or tracks air travel might have dozens or hundreds. The number of tables tells me not only how much data I'll need to analyze, but also hints that I should explore relationships among the data in each table.

Before you dig into SQL, let's look at an example of what the contents of tables might look like. We'll use a hypothetical database for managing a school's class enrollment; within that database are several tables that track students and their classes. The first table, called `student_enrollment`, shows the students that are signed up for each class section:

```
student_id    class_id      class_section    semester
----------    ----------    -------------    ---------
CHRISPA004    COMPSCI101    3                Fall 2017
DAVISHE010    COMPSCI101    3                Fall 2017
ABRILDA002    ENG101        40               Fall 2017
DAVISHE010    ENG101        40               Fall 2017
RILEYPH002    ENG101        40               Fall 2017
```

This table shows that two students have signed up for `COMPSCI101`, and three have signed up for `ENG101`. But where are the details about each student and class? In this example, these details are stored in separate tables called `students` and `classes`, and each table relates to this one. This is where the power of a *relational database* begins to show itself.

The first several rows of the `students` table include the following:

```
student_id    first_name    last_name    dob
----------    ----------    ---------    ----------
ABRILDA002    Abril         Davis        1999-01-10
CHRISPA004    Chris         Park         1996-04-10
DAVISHE010    Davis         Hernandez    1987-09-14
RILEYPH002    Riley         Phelps       1996-06-15
```

The `students` table contains details on each student, using the value in the `student_id` column to identify each one. That value acts as a unique *key* that connects both tables, giving you the ability to create rows such as the following with the `class_id` column from `student_enrollment` and the `first_name` and `last_name` columns from `students`:

```
class_id      first_name    last_name
----------    ----------    ---------
COMPSCI101    Davis         Hernandez
COMPSCI101    Chris         Park
ENG101        Abril         Davis
ENG101        Davis         Hernandez
ENG101        Riley         Phelps
```

The `classes` table would work the same way, with a `class_id` column and several columns of detail about the class. Database builders prefer to organize data using separate tables for each main *entity* the database manages in order to reduce redundant data. In the example, we store each student's name and date of birth just once. Even if the student signs up for multiple

classes—as Davis Hernandez did—we don't waste database space entering his name next to each class in the `student_enrollment` table. We just include his student ID.

Given that tables are a core building block of every database, in this chapter you'll start your SQL coding adventure by creating a table inside a new database. Then you'll load data into the table and view the completed table.

Creating a Database

The PostgreSQL program you downloaded in the Introduction is a *database management system*, a software package that allows you to define, manage, and query databases. When you installed PostgreSQL, it created a *database server*—an instance of the application running on your computer—that includes a default database called `postgres`. The database is a collection of objects that includes tables, functions, user roles, and much more. According to the PostgreSQL documentation, the default database is "meant for use by users, utilities and third party applications" (see *https://www.postgresql.org/docs/current/static/app-initdb.html*). In the exercises in this chapter, we'll leave the default as is and instead create a new one. We'll do this to keep objects related to a particular topic or application organized together.

To create a database, you use just one line of SQL, shown in Listing 1-1. This code, along with all the examples in this book, is available for download via the resources at *https://www.nostarch.com/practicalSQL/*.

```
CREATE DATABASE analysis;
```

Listing 1-1: Creating a database named `analysis`

This statement creates a database on your server named `analysis` using default PostgreSQL settings. Note that the code consists of two keywords—`CREATE` and `DATABASE`—followed by the name of the new database. The statement ends with a semicolon, which signals the end of the command. The semicolon ends all PostgreSQL statements and is part of the ANSI SQL standard. Sometimes you can omit the semicolon, but not always, and particularly not when running multiple statements in the admin. So, using the semicolon is a good habit to form.

Executing SQL in pgAdmin

As part of the Introduction to this book, you also installed the graphical administrative tool pgAdmin (if you didn't, go ahead and do that now). For much of our work, you'll use pgAdmin to run (or execute) the SQL statements we write. Later in the book in Chapter 16, I'll show you how to run SQL statements in a terminal window using the PostgreSQL command line program `psql`, but getting started is a bit easier with a graphical interface.

We'll use pgAdmin to run the SQL statement in Listing 1-1 that creates the database. Then, we'll connect to the new database and create a table. Follow these steps:

1. Run PostgreSQL. If you're using Windows, the installer set PostgreSQL to launch every time you boot up. On macOS, you must double-click *Postgres.app* in your Applications folder.
2. Launch pgAdmin. As you did in the Introduction, in the left vertical pane (the object browser) expand the plus sign to the left of the Servers node to show the default server. Depending on how you installed PostgreSQL, the default server may be named *localhost* or *PostgreSQL x*, where *x* is the version of the application.
3. Double-click the server name. If you supplied a password during installation, enter it at the prompt. You'll see a brief message that pgAdmin is establishing a connection.
4. In pgAdmin's object browser, expand **Databases** and click once on the `postgres` database to highlight it, as shown in Figure 1-1.
5. Open the Query Tool by choosing **Tools ▸ Query Tool**.
6. In the SQL Editor pane (the top horizontal pane), type or copy the code from Listing 1-1.
7. Click the lightning bolt icon to execute the statement. PostgreSQL creates the database, and in the Output pane in the Query Tool under Messages you'll see a notice indicating the query returned successfully, as shown in Figure 1-2.

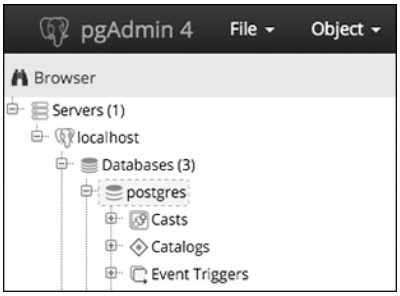

Figure 1-1: Connecting to the default `postgres` database

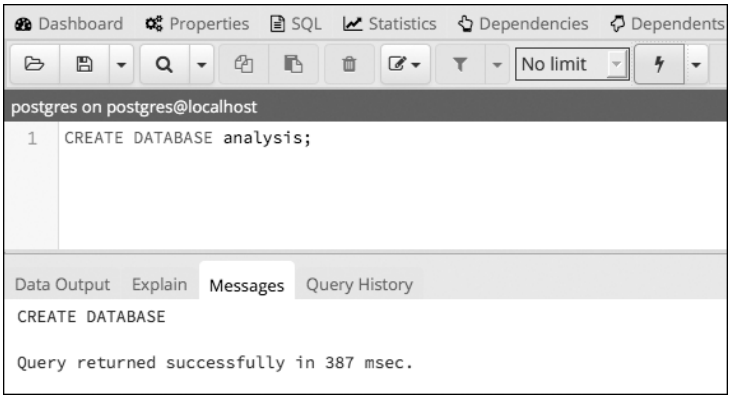

Figure 1-2: Creating the `analysis` database

8. To see your new database, right-click **Databases** in the object browser. From the pop-up menu, select **Refresh**, and the `analysis` database will appear in the list, as shown in Figure 1-3.

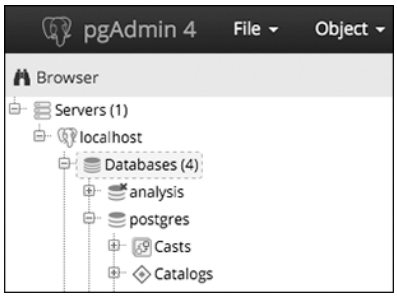

Figure 1-3: The `analysis` database displayed in the object browser

Good work! You now have a database called `analysis`, which you can use for the majority of the exercises in this book. In your own work, it's generally a best practice to create a new database for each project to keep tables with related data together.

Connecting to the Analysis Database

Before you create a table, you must ensure that pgAdmin is connected to the `analysis` database rather than to the default `postgres` database.

To do that, follow these steps:

1. Close the Query Tool by clicking the **X** at the top right of the tool. You don't need to save the file when prompted.
2. In the object browser, click once on the `analysis` database.
3. Reopen the Query Tool by choosing **Tools ▸ Query Tool**.
4. You should now see the label `analysis on postgres@localhost` at the top of the Query Tool window. (Again, instead of `localhost`, your version may show `PostgreSQL`.)

Now, any code you execute will apply to the `analysis` database.

Creating a Table

As I mentioned earlier, tables are where data lives and its relationships are defined. When you create a table, you assign a name to each *column* (sometimes referred to as a *field* or *attribute*) and assign it a *data type*. These are the values the column will accept—such as text, integers, decimals, and dates—and the definition of the data type is one way SQL enforces the integrity of data. For example, a column defined as `date` will take data in one of several standard formats, such as *`YYYY-MM-DD`*. If you try to enter characters not in a date format, for instance, the word `peach`, you'll receive an error.

Data stored in a table can be accessed and analyzed, or queried, with SQL statements. You can sort, edit, and view the data, and easily alter the table later if your needs change.

Let's make a table in the `analysis` database.

The CREATE TABLE Statement

For this exercise, we'll use an often-discussed piece of data: teacher salaries. Listing 1-2 shows the SQL statement to create a table called `teachers`:

```
❶ CREATE TABLE teachers (
   ❷ id bigserial,
   ❸ first_name varchar(25),
     last_name varchar(50),
     school varchar(50),
   ❹ hire_date date,
   ❺ salary numeric
❻ );
```

Listing 1-2: Creating a table named teachers with six columns

This table definition is far from comprehensive. For example, it's missing several *constraints* that would ensure that columns that must be filled do indeed have data or that we're not inadvertently entering duplicate values. I cover constraints in detail in Chapter 7, but in these early chapters I'm omitting them to focus on getting you started on exploring data.

The code begins with the two SQL keywords ❶ `CREATE` and `TABLE` that, together with the name `teachers`, signal PostgreSQL that the next bit of code describes a table to add to the database. Following an opening parenthesis, the statement includes a comma-separated list of column names along with their data types. For style purposes, each new line of code is on its own line and indented four spaces, which isn't required, but it makes the code more readable.

Each column name represents one discrete data element defined by a data type. The `id` column ❷ is of data type `bigserial`, a special integer type that auto-increments every time you add a row to the table. The first row receives the value of `1` in the `id` column, the second row `2`, and so on. The `bigserial` data type and other serial types are PostgreSQL-specific implementations, but most database systems have a similar feature.

Next, we create columns for the teacher's first and last name, and the school where they teach ❸. Each is of the data type `varchar`, a text column with a maximum length specified by the number in parentheses. We're assuming that no one in the database will have a last name of more than 50 characters. Although this is a safe assumption, you'll discover over time that exceptions will always surprise you.

The teacher's `hire_date` ❹ is set to the data type `date`, and the `salary` column ❺ is a `numeric`. I'll cover data types more thoroughly in Chapter 3, but this table shows some common examples of data types. The code block wraps up ❻ with a closing parenthesis and a semicolon.

Now that you have a sense of how SQL looks, let's run this code in pgAdmin.

Making the teachers Table

You have your code and you're connected to the database, so you can make the table using the same steps we did when we created the database:

1. Open the pgAdmin Query Tool (if it's not open, click once on the `analysis` database in pgAdmin's object browser, and then choose **Tools ▸ Query Tool**).
2. Copy the `CREATE TABLE` script from Listing 1-2 into the SQL Editor.
3. Execute the script by clicking the lightning bolt icon.

If all goes well, you'll see a message in the pgAdmin Query Tool's bottom output pane that reads, `Query returned successfully with no result in 84 msec`. Of course, the number of milliseconds will vary depending on your system.

Now, find the table you created. Go back to the main pgAdmin window and, in the object browser, right-click the `analysis` database and choose **Refresh**. Choose **Schemas ▸ public ▸ Tables** to see your new table, as shown in Figure 1-4.

Expand the `teachers` table node by clicking the plus sign to the left of its name. This reveals more details about the table, including the column names, as shown in Figure 1-5. Other information appears as well, such as indexes, triggers, and constraints, but I'll cover those in later chapters. Clicking on the table name and then selecting the **SQL** menu in the pgAdmin workspace will display the SQL statement used to make the `teachers` table.

Congratulations! So far, you've built a database and added a table to it. The next step is to add data to the table so you can write your first query.

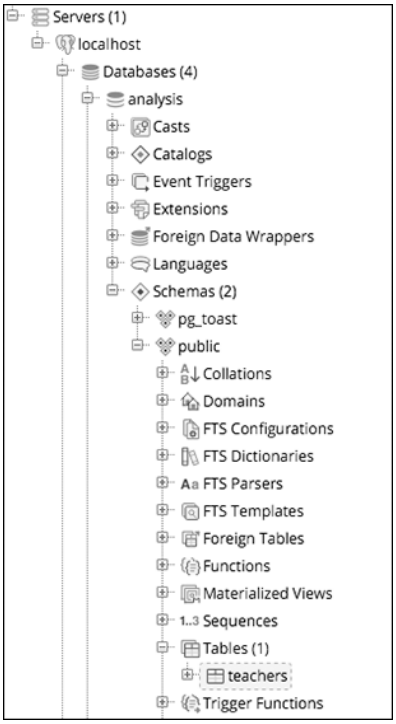

Figure 1-4: The teachers table in the object browser

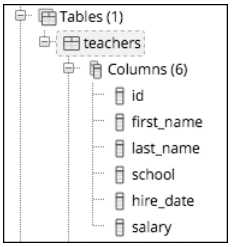

Figure 1-5: Table details for teachers

Inserting Rows into a Table

You can add data to a PostgreSQL table in several ways. Often, you'll work with a large number of rows, so the easiest method is to import data from a text file or another database directly into a table. But just to get started, we'll add a few rows using an `INSERT INTO ... VALUES` statement that specifies the target columns and the data values. Then we'll view the data in its new home.

The INSERT Statement

To insert some data into the table, you first need to erase the `CREATE TABLE` statement you just ran. Then, following the same steps as you did to create the database and table, copy the code in Listing 1-3 into your pgAdmin Query Tool:

```
❶ INSERT INTO teachers (first_name, last_name, school, hire_date, salary)
❷ VALUES ('Janet', 'Smith', 'F.D. Roosevelt HS', '2011-10-30', 36200),
       ('Lee', 'Reynolds', 'F.D. Roosevelt HS', '1993-05-22', 65000),
       ('Samuel', 'Cole', 'Myers Middle School', '2005-08-01', 43500),
       ('Samantha', 'Bush', 'Myers Middle School', '2011-10-30', 36200),
       ('Betty', 'Diaz', 'Myers Middle School', '2005-08-30', 43500),
       ('Kathleen', 'Roush', 'F.D. Roosevelt HS', '2010-10-22', 38500);❸
```

Listing 1-3: Inserting data into the teachers table

This code block inserts names and data for six teachers. Here, the PostgreSQL syntax follows the ANSI SQL standard: after the `INSERT INTO` keywords is the name of the table, and in parentheses are the columns to be filled ❶. In the next row is the `VALUES` keyword and the data to insert into each column in each row ❷. You need to enclose the data for each row in a set of parentheses, and inside each set of parentheses, use a comma to separate each column value. The order of the values must also match the order of the columns specified after the table name. Each row of data ends with a comma, and the last row ends the entire statement with a semicolon ❸.

Notice that certain values that we're inserting are enclosed in single quotes, but some are not. This is a standard SQL requirement. Text and dates require quotes; numbers, including integers and decimals, don't require quotes. I'll highlight this requirement as it comes up in examples. Also, note the date format we're using: a four-digit year is followed by the month and date, and each part is joined by a hyphen. This is the international standard for date formats; using it will help you avoid confusion. (Why is it best to use the format *YYYY-MM-DD*? Check out *https://xkcd.com/1179/* to see a great comic about it.) PostgreSQL supports many additional date formats, and I'll use several in examples.

You might be wondering about the `id` column, which is the first column in the table. When you created the table, your script specified that column to be the `bigserial` data type. So as PostgreSQL inserts each row, it automatically fills the `id` column with an auto-incrementing integer. I'll cover that in detail in Chapter 3 when I discuss data types.

Now, run the code. This time the message in the Query Tool should include the words `Query returned successfully: 6 rows affected`.

Viewing the Data

You can take a quick look at the data you just loaded into the `teachers` table using pgAdmin. In the object browser, locate the table and right-click. In the pop-up menu, choose **View/Edit Data ▸ All Rows**. As Figure 1-6 shows, you'll see the six rows of data in the table with each column filled by the values in the SQL statement.

Data Output | Explain | Messages | Query History

	id bigint	first_name character vary	last_name character vary	school character varying	hire_date date	salary numeric
1	1	Janet	Smith	F.D. Roosevelt ...	2011-10-30	36200
2	2	Lee	Reynolds	F.D. Roosevelt ...	1993-05-22	65000
3	3	Samuel	Cole	Myers Middle S...	2005-08-01	43500
4	4	Samantha	Bush	Myers Middle S...	2011-10-30	36200
5	5	Betty	Diaz	Myers Middle S...	2005-08-30	43500
6	6	Kathleen	Roush	F.D. Roosevelt ...	2010-10-22	38500

Figure 1-6: Viewing table data directly in pgAdmin

Notice that even though you didn't insert a value for the `id` column, each teacher has an ID number assigned.

You can view data using the pgAdmin interface in a few ways, but we'll focus on writing SQL to handle those tasks.

When Code Goes Bad

There may be a universe where code always works, but unfortunately, we haven't invented a machine capable of transporting us there. Errors happen. Whether you make a typo or mix up the order of operations, computer languages are unforgiving about syntax. For example, if you forget a comma in the code in Listing 1-3, PostgreSQL squawks back an error:

```
ERROR:  syntax error at or near "("
LINE 5:     ('Samuel', 'Cole', 'Myers Middle School', '2005-08-01', 43...
            ^
********** Error **********
```

Fortunately, the error message hints at what's wrong and where: a syntax error is near an open parenthesis on line 5. But sometimes error messages can be more obscure. In that case, you do what the best coders do: a quick internet search for the error message. Most likely, someone else has experienced the same issue and might know the answer.

Formatting SQL for Readability

SQL requires no special formatting to run, so you're free to use your own psychedelic style of uppercase, lowercase, and random indentations. But that won't win you any friends when others need to work with your code (and sooner or later someone will). For the sake of readability and being a good coder, it's best to follow these conventions:

- Uppercase SQL keywords, such as `SELECT`. Some SQL coders also uppercase the names of data types, such as `TEXT` and `INTEGER`. I use lowercase characters for data types in this book to separate them in your mind from keywords, but you can uppercase them if desired.
- Avoid camel case and instead use `lowercase_and_underscores` for object names, such as tables and column names (see more details about case in Chapter 7).
- Indent clauses and code blocks for readability using either two or four spaces. Some coders prefer tabs to spaces; use whichever works best for you or your organization.

We'll explore other SQL coding conventions as we go through the book, but these are the basics.

Wrapping Up

You accomplished quite a bit in this first chapter: you created a database and a table, and then loaded data into it. You're on your way to adding SQL to your data analysis toolkit! In the next chapter, you'll use this set of teacher data to learn the basics of querying a table using `SELECT`.

TRY IT YOURSELF

Here are two exercises to help you explore concepts related to databases, tables, and data relationships:

1. Imagine you're building a database to catalog all the animals at your local zoo. You want one table to track the kinds of animals in the collection and another table to track the specifics on each animal. Write `CREATE TABLE` statements for each table that include some of the columns you need. Why did you include the columns you chose?
2. Now create `INSERT` statements to load sample data into the tables. How can you view the data via the pgAdmin tool? Create an additional `INSERT` statement for one of your tables. Purposely omit one of the required commas separating the entries in the `VALUES` clause of the query. What is the error message? Would it help you find the error in the code?