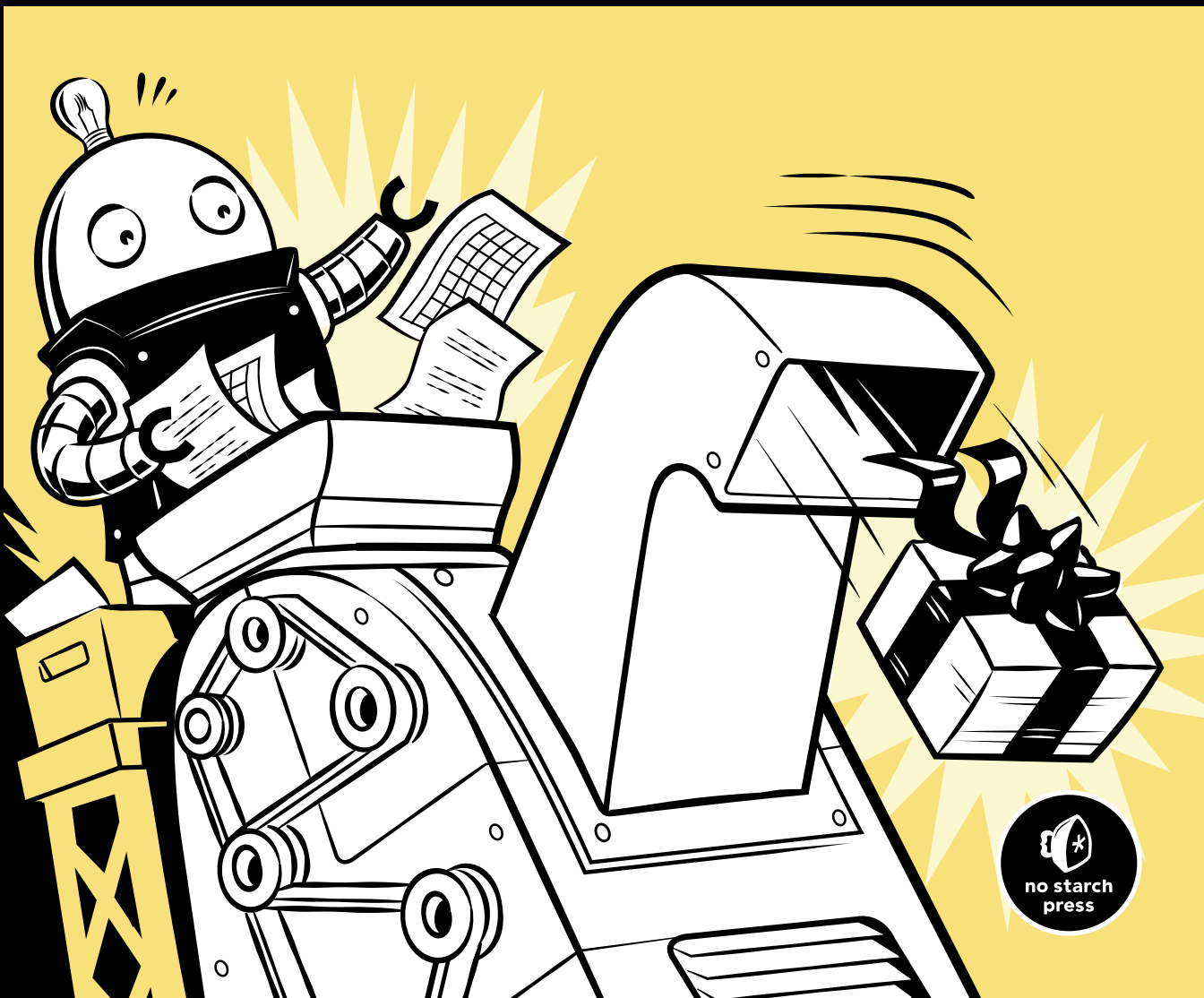
PRACTICAL SQL

A BEGINNER'S GUIDE TO *STORYTELLING* WITH *DATA*

ANTHONY DEBARROS



no starch press

PRACTICAL SQL

A Beginner's Guide to Storytelling with Data

by Anthony DeBarros

no starch
press

San Francisco

About the Author

Anthony DeBarros is an award-winning journalist who has combined avid interests in data analysis, coding, and storytelling for much of his career. He spent more than 25 years with the Gannett company, including the *Poughkeepsie Journal*, *USA TODAY*, and Gannett Digital. He is currently senior vice president for content and product development for a publishing and events firm and lives and works in the Washington, D.C., area.

About the Technical Reviewer

Josh Berkus is a "hacker emeritus" for the PostgreSQL Project, where he served on the Core Team for 13 years. He was also a database consultant for 15 years, working with PostgreSQL, MySQL, CitusDB, Redis, CouchDB, Hadoop, and Microsoft SQL Server. Josh currently works as a Kubernetes community manager at Red Hat, Inc.

BRIEF CONTENTS

CONTENTS IN DETAIL

FOREWORD

When people ask which programming language I learned first, I often absent-mindedly reply, "Python," forgetting that it was actually with SQL that I first learned to write code. This is probably because learning SQL felt so intuitive after spending years running formulas in Excel spreadsheets. I didn't have a technical background, but I found SQL's syntax, unlike that of many other programming languages, straightforward and easy to implement. For example, you run `SELECT *` on a SQL table to make every row and column appear. You simply use the `JOIN` keyword to return rows of data from different related tables, which you can then further group, sort, and analyze.

I'm a graphics editor, and I've worked as a developer and journalist at a number of publications, including *POLITICO*, *Vox*, and *USA TODAY*. My daily responsibilities involve analyzing data and creating visualizations from what I find. I first used SQL when I worked at *The Chronicle of Higher Education* and its sister publication, *The Chronicle of Philanthropy*. Our team

analyzed data ranging from nonprofit financials to faculty salaries at colleges and universities. Many of our projects included as much as 20 years' worth of data, and one of my main tasks was to import all that data into a SQL database and analyze it. I had to calculate the percent change in fundraising dollars at a nonprofit or find the median endowment size at a university to measure an institution's performance.

I discovered SQL to be a powerful language, one that fundamentally shaped my understanding of what you can—and can't—do with data. SQL excels at bringing order to messy, large data sets and helps you discover how different data sets are related. Plus, its queries and functions are easy to reuse within the same project or even in a different database.

This leads me to *Practical SQL*. Looking back, I wish I'd read Chapter 4 on "Importing and Exporting Data" so I could have understood the power of bulk imports instead of writing long, cumbersome `INSERT` statements when filling a table. The statistical capabilities of PostgreSQL, covered in Chapters 5 and 10 in this book, are also something I wish I had grasped earlier, as my data analysis often involves calculating the percent change or finding the average or median values. I'm embarrassed to say that I didn't know how `percentile_cont()`, covered in Chapter 5, could be used to easily calculate a median in PostgresSQL—with the added bonus that it also finds your data's natural breaks or quantiles.

But at that stage in my career, I was only scratching the surface of SQL's capabilities. It wasn't until 2014, when I became a data developer at Gannett Digital on a team led by Anthony DeBarros, that I learned to use PostgreSQL. I began to understand just how enormously powerful SQL was for creating a reproducible and sustainable workflow.

When I met Anthony, he had been working at *USA TODAY* and other Gannett properties for more than 20 years, where he had led teams that built databases and published award-winning investigations. Anthony was able to show me the ins and outs of our team's databases in addition to teaching me how to properly build and maintain my own. It was through working with Anthony that I truly learned how to code.

One of the first projects Anthony and I collaborated on was the 2014 U.S. midterm elections. We helped build an election forecast data visualization to show *USA TODAY* readers the latest polling averages, campaign finance data, and biographical information for more than 1,300 candidates in more than 500 congressional and gubernatorial races. Building our data infrastructure was a complex, multistep process powered by a PostgreSQL database at its heart.

Anthony taught me how to write code that funneled all the data from our sources into a half-dozen tables in PostgreSQL. From there, we could query the data into a format that would power the maps, charts, and front-end presentation of our election forecast.

Around this time, I also learned one of my favorite things about PostgreSQL—its powerful suite of geographic functions (Chapter 14 in this book). By adding the PostGIS extension to the database, you can create spatial data that you can then export as GeoJSON or as a shapefile, a format that is easy to map. You can also perform complex spatial

analysis, like calculating the distance between two points or finding the density of schools or, as Anthony shows in the chapter, all the farmers' markets in a given radius.

It's a skill I've used repeatedly in my career. For example, I used it to build a data set of lead exposure risk at the census-tract level while at *Vox*, which I consider one of my crowning PostGIS achievements. Using this database, I was able to create a data set of every U.S. Census tract and its corresponding lead exposure risk in a spatial format that could be easily mapped at the national level.

With so many different programming languages available—more than 200, if you can believe it—it's truly overwhelming to know where to begin. One of the best pieces of advice I received when first starting to code was to find an inefficiency in my workflow that could be improved by coding. In my case, it was building a database to easily query a project's data. Maybe you're in a similar boat or maybe you just want to know how to analyze large data sets.

Regardless, you're probably looking for a no-nonsense guide that skips the programming jargon and delves into SQL in an easy-to-understand manner that is both practical and, more importantly, applicable. And that's exactly what *Practical SQL* does. It gets away from programming theory and focuses on teaching SQL by example, using real data sets you'll likely encounter. It also doesn't shy away from showing you how to deal with annoying messy data pitfalls: misspelled names, missing values, and columns with unsuitable data types. This is important because, as you'll quickly learn, there's no such thing as clean data.

Over the years, my role as a data journalist has evolved. I build fewer databases now and build more maps. I also report more. But the core requirement of my job, and what I learned when first learning SQL, remains the same: know thy data and to thine own data be true. In other words, the most important aspect of working with data is being able to understand what's in it.

You can't expect to ask the right questions of your data or tell a compelling story if you don't understand how to best analyze it. Fortunately, that's where *Practical SQL* comes in. It'll teach you the fundamentals of working with data so that you can discover your own stories and insights.

Sarah Frostenson
Graphics Editor at *POLITICO*

ACKNOWLEDGMENTS

Practical SQL is the work of many hands. My thanks, first, go to the team at No Starch Press. Thanks to Bill Pollock and Tyler Ortman for capturing the vision and sharpening the initial concept; to developmental editors Annie Choi and Liz Chadwick for refining each chapter; to copyeditor Anne Marie Walker for polishing the final drafts with an eagle eye; and to production editor Janelle Ludowise for laying out the book and keeping the process well organized.

Josh Berkus, Kubernetes community manager for Red Hat, Inc., served as our technical reviewer. To work with Josh was to receive a master class in SQL and PostgreSQL. Thank you, Josh, for your patience and high standards.

Thank you to Investigative Reporters and Editors (IRE) and its members and staff past and present for training journalists to find great stories in data. IRE is where I got my start with SQL and data journalism.

During my years at *USA TODAY*, many colleagues either taught me SQL or imparted memorable lessons on data analysis. Special thanks to

Paul Overberg for sharing his vast knowledge of demographics and the U.S. Census, to Lou Schilling for many technical lessons, to Christopher Schnaars for his SQL expertise, and to Sarah Frostenson for graciously agreeing to write the book's foreword.

My deepest appreciation goes to my dear wife, Elizabeth, and our sons. Thank you for making every day brighter and warmer, for your love, and for bearing with me as I completed this book.

INTRODUCTION

Shortly after joining the staff of *USA TODAY* I received a data set I would analyze almost every week for the next decade. It was the weekly Best-Selling Books list, which ranked the nation's top-selling books based on confidential sales data. The list not only produced an endless stream of story ideas to pitch, but it also captured the zeitgeist of America in a singular way.

For example, did you know that cookbooks sell a bit more during the week of Mother's Day, or that Oprah Winfrey turned many obscure writers into number one best-selling authors just by having them on her show? Week after week, the book list editor and I pored over the sales figures and book genres, ranking the data in search of the next headline. Rarely did we come up empty: we chronicled everything from the rocket-rise of the blockbuster *Harry Potter* series to the fact that *Oh, the Places You'll Go!* by Dr. Seuss has become a perennial gift for new graduates.

My technical companion during this time was the database programming language *SQL* (for *Structured Query Language*). Early on, I convinced *USA TODAY*'s IT department to grant me access to the SQL-based database system that powered our book list application. Using SQL, I was able to unlock the stories hidden in the database, which contained titles, authors, genres, and various codes that defined the publishing world. Analyzing data with SQL to discover interesting stories is exactly what you'll learn to do using this book.

What Is SQL?

SQL is a widely used programming language that allows you to define and query databases. Whether you're a marketing analyst, a journalist, or a researcher mapping neurons in the brain of a fruit fly, you'll benefit from using SQL to manage database objects as well as create, modify, explore, and summarize data.

Because SQL is a mature language that has been around for decades, it's deeply ingrained in many modern systems. A pair of IBM researchers first outlined the syntax for SQL (then called SEQUEL) in a 1974 paper, building on the theoretical work of the British computer scientist Edgar F. Codd. In 1979, a precursor to the database company Oracle (then called Relational Software) became the first to use the language in a commercial product. Today, it continues to rank as one of the most-used computer languages in the world, and that's unlikely to change soon.

SQL comes in several variants, which are generally tied to specific database systems. The American National Standards Institute (ANSI) and International Organization for Standardization (ISO), which set standards for products and technologies, provide standards for the language and shepherd revisions to it. The good news is that the variants don't stray far from the standard, so once you learn the SQL conventions for one database, you can transfer that knowledge to other systems.

Why Use SQL?

So why should you use SQL? After all, SQL is not usually the first tool people choose when they're learning to analyze data. In fact, many people start with Microsoft Excel spreadsheets and their assortment of analytic functions. After working with Excel, they might graduate to Access, the database system built into Microsoft Office, which has a graphical query interface that makes it easy to get work done, making SQL skills optional.

But as you might know, Excel and Access have their limits. Excel currently allows 1,048,576 rows maximum per worksheet, and Access limits database size to two gigabytes and limits columns to 255 per table. It's not uncommon for data sets to surpass those limits, particularly when you're working with data dumped from government systems. The last obstacle you want to discover while facing a deadline is that your database system doesn't have the capacity to get the job done.

Using a robust SQL database system allows you to work with terabytes of data, multiple related tables, and thousands of columns. It gives you improved programmatic control over the structure of your data, leading to efficiency, speed, and—most important—accuracy.

SQL is also an excellent adjunct to programming languages used in the data sciences, such as R and Python. If you use either language, you can connect to SQL databases and, in some cases, even incorporate SQL syntax directly into the language. For people with no background in programming languages, SQL often serves as an easy-to-understand introduction into concepts related to data structures and programming logic.

Additionally, knowing SQL can help you beyond data analysis. If you delve into building online applications, you'll find that databases provide the backend power for many common web frameworks, interactive maps, and content management systems. When you need to dig beneath the surface of these applications, SQL's capability to manipulate data and databases will come in very handy.

About This Book

Practical SQL is for people who encounter data in their everyday lives and want to learn how to analyze and transform it. To this end, I discuss real-world data and scenarios, such as U.S. Census demographics, crime statistics, and data about taxi rides in New York City. Along with information about databases and code, you'll also learn tips on how to analyze and acquire data as well as other valuable insights I've accumulated throughout my career. I won't focus on setting up servers or other tasks typically handled by a database administrator, but the SQL and PostgreSQL fundamentals you learn in this book will serve you well if you intend to go that route.

I've designed the exercises for beginner SQL coders but will assume that you know your way around your computer, including how to install programs, navigate your hard drive, and download files from the internet. Although many chapters in this book can stand alone, you should work through the book sequentially to build on the fundamentals. Some data sets used in early chapters reappear later in the book, so following the book in order will help you stay on track.

Practical SQL starts with the basics of databases, queries, tables, and data that are common to SQL across many database systems. Chapters 13 to 17 cover topics more specific to PostgreSQL, such as full text search and GIS. The following table of contents provides more detail about the topics discussed in each chapter:

Chapter 1: Creating Your First Database and Table introduces PostgreSQL, the pgAdmin user interface, and the code for loading a simple data set about teachers into a new database.

Chapter 2: Beginning Data Exploration with `SELECT` explores basic SQL query syntax, including how to sort and filter data.

Chapter 3: Understanding Data Types explains the definitions for setting columns in a table to hold specific types of data, from text to dates to various forms of numbers.

Chapter 4: Importing and Exporting Data explains how to use SQL commands to load data from external files and then export it. You'll load a table of U.S. Census population data that you'll use throughout the book.

Chapter 5: Basic Math and Stats with SQL covers arithmetic operations and introduces aggregate functions for finding sums, averages, and medians.

Chapter 6: Joining Tables in a Relational Database explains how to query multiple, related tables by joining them on key columns. You'll learn how and when to use different types of joins.

Chapter 7: Table Design that Works for You covers how to set up tables to improve the organization and integrity of your data as well as how to speed up queries using indexes.

Chapter 8: Extracting Information by Grouping and Summarizing explains how to use aggregate functions to find trends in U.S. library use based on annual surveys.

Chapter 9: Inspecting and Modifying Data explores how to find and fix incomplete or inaccurate data using a collection of records about meat, egg, and poultry producers as an example.

Chapter 10: Statistical Functions in SQL introduces correlation, regression, and ranking functions in SQL to help you derive more meaning from data sets.

Chapter 11: Working with Dates and Times explains how to create, manipulate, and query dates and times in your database, including working with time zones, using data on New York City taxi trips and Amtrak train schedules.

Chapter 12: Advanced Query Techniques explains how to use more complex SQL operations, such as subqueries and cross tabulations, and the `CASE` statement to reclassify values in a data set on temperature readings.

Chapter 13: Mining Text to Find Meaningful Data covers how to use PostgreSQL's full text search engine and regular expressions to extract data from unstructured text, using a collection of speeches by U.S. presidents as an example.

Chapter 14: Analyzing Spatial Data with PostGIS introduces data types and queries related to spatial objects, which will let you analyze geographical features like states, roads, and rivers.

Chapter 15: Saving Time with Views, Functions, and Triggers explains how to automate database tasks so you can avoid repeating routine work.

Chapter 16: Using PostgreSQL from the Command Line covers how to use text commands at your computer's command prompt to connect to your database and run queries.

Chapter 17: Maintaining Your Database provides tips and procedures for tracking the size of your database, customizing settings, and backing up data.

Chapter 18: Identifying and Telling the Story Behind Your Data provides guidelines for generating ideas for analysis, vetting data, drawing sound conclusions, and presenting your findings clearly.

Appendix: Additional PostgreSQL Resources lists software and documentation to help you grow your skills.

Each chapter ends with a "Try It Yourself" section that contains exercises to help you reinforce the topics you learned.

Using the Book's Code Examples

Each chapter includes code examples, and most use data sets I've already compiled. All the code and sample data in the book is available to download at *https://www.nostarch.com/practicalSQL/*. Click the **Download the code from GitHub** link to go to the GitHub repository that holds this material. At GitHub, you should see a "Clone or Download" button that gives you the option to download a ZIP file with all the materials. Save the file to your computer in a location where you can easily find it, such as your desktop.

Inside the ZIP file is a folder for each chapter. Each folder contains a file named *Chapter_XX* (*XX* is the chapter number) that ends with a *.sql* extension. You can open those files with a text editor or with the PostgreSQL administrative tool you'll install. You can copy and paste code when the book instructs you to run it. Note that in the book, several code examples are truncated to save space, but you'll need the full listing from the *.sql* file to complete the exercise. You'll know an example is truncated when you see `--snip--` inside the listing.

Also in the *.sql* files, you'll see lines that begin with two hyphens (--) and a space. These are comments that provide the code's listing number and additional context, but they're not part of the code. These comments also note when the file has additional examples that aren't in the book.

NOTE *After downloading data, Windows users might need to provide permission for the database to read files. To do so, right-click the folder containing the code and data, select Properties, and click the Security tab. Click* ***Edit****, then* ***Add****. Type the name* ***Everyone*** *into the object names box and click* ***OK****. Highlight Everyone in the user list, select all boxes under Allow, and then click* ***Apply*** *and* ***OK****.*

Using PostgreSQL

In this book, I'll teach you SQL using the open source PostgreSQL database system. PostgreSQL, or simply Postgres, is a robust database system that can handle very large amounts of data. Here are some reasons PostgreSQL is a great choice to use with this book:

- It's free.
- It's available for Windows, macOS, and Linux operating systems.
- Its SQL implementation closely follows ANSI standards.
- It's widely used for analytics and data mining, so finding help online from peers is easy.
- Its geospatial extension, PostGIS, lets you analyze geometric data and perform mapping functions.
- It's available in several variants, such as Amazon Redshift and Greenplum, which focus on processing huge data sets.
- It's a common choice for web applications, including those powered by the popular web frameworks Django and Ruby on Rails.

Of course, you can also use another database system, such as Microsoft SQL Server or MySQL; many code examples in this book translate easily to either SQL implementation. However, some examples, especially later in the book, do not, and you'll need to search online for equivalent solutions. Where appropriate, I'll note whether an example code follows the ANSI SQL standard and may be portable to other systems or whether it's specific to PostgreSQL.

Installing PostgreSQL

You'll start by installing the PostgreSQL database and the graphical administrative tool pgAdmin, which is software that makes it easy to manage your database, import and export data, and write queries.

One great benefit of working with PostgreSQL is that regardless of whether you work on Windows, macOS, or Linux, the open source community has made it easy to get PostgreSQL up and running. The following sections outline installation for all three operating systems as of this writing, but options might change as new versions are released. Check the documentation noted in each section as well as the GitHub repository with the book's resources; I'll maintain the files with updates and answers to frequently asked questions.

NOTE *Always install the latest available version of PostgreSQL for your operating system to ensure that it's up to date on security patches and new features. For this book, I'll assume you're using version 10.0 or later.*

Windows Installation

For Windows, I recommend using the installer provided by the company EnterpriseDB, which offers support and services for PostgreSQL users. EnterpriseDB's package bundles PostgreSQL with pgAdmin and the company's own Stack Builder, which also installs the spatial database extension PostGIS and programming language support, among other tools. To get the software, visit *https://www.enterprisedb.com/* and create a free account. Then go to the downloads page at *https://www.enterprisedb.com/software-downloads-postgres/*.

Select the latest available 64-bit Windows version of EDB Postgres Standard unless you're using an older PC with 32-bit Windows. After you download the installer, follow these steps:

1. Right-click the installer and select **Run as administrator**. Answer **Yes** to the question about allowing the program to make changes to your computer. The program will perform a setup task and then present an initial welcome screen. Click through it.
2. Choose your installation directory, accepting the default.
3. On the Select Components screen, select the boxes to install PostgreSQL Server, the pgAdmin tool, Stack Builder, and Command Line Tools.
4. Choose the location to store data. You can choose the default, which is in a "data" subdirectory in the PostgreSQL directory.
5. Choose a password. PostgreSQL is robust with security and permissions. This password is for the initial database superuser account, which is called `postgres`.
6. Select a port number where the server will listen. Unless you have another database or application using it, the default of `5432` should be fine. If you have another version of PostgreSQL already installed or some other application is using that default, the value might be `5433` or another number, which is also okay.
7. Select your locale. Using the default is fine. Then click through the summary screen to begin the installation, which will take several minutes.
8. When the installation is done, you'll be asked whether you want to launch EnterpriseDB's Stack Builder to obtain additional packages. Select the box and click **Finish**.
9. When Stack Builder launches, choose the PostgreSQL installation on the drop-down menu and click **Next**. A list of additional applications should download.
10. Expand the **Spatial Extensions** menu and select either the 32-bit or 64-bit version of PostGIS Bundle for the version of Postgres you installed. Also, expand the **Add-ons, tools and utilities** menu and select EDB Language Pack, which installs support for programming languages including Python. Click through several times; you'll need to wait while the installer downloads the additional components.

11. When installation files have been downloaded, click **Next** to install both components. For PostGIS, you'll need to agree to the license terms; click through until you're asked to Choose Components. Make sure PostGIS and Create spatial database are selected. Click **Next**, accept the default database location, and click **Next** again.
12. Enter your database password when prompted and continue through the prompts to finish installing PostGIS.
13. Answer **Yes** when asked to register GDAL. Also, answer **Yes** to the questions about setting POSTGIS_ENABLED_DRIVERS and enabling the POSTGIS_ENABLE_OUTDB_RASTERS environment variable.

When finished, a PostgreSQL folder that contains shortcuts and links to documentation should be on your Windows Start menu.

If you experience any hiccups installing PostgreSQL, refer to the "Troubleshooting" section of the EDB guide at *https://www.enterprisedb.com/resources/product-documentation/*. If you're unable to install PostGIS via Stack Builder, try downloading a separate installer from the PostGIS site at *http://postgis.net/windows_downloads/* and consult the guides at *http://postgis.net/documentation/*.

macOS Installation

For macOS users, I recommend obtaining Postgres.app, an open source macOS application that includes PostgreSQL as well as the PostGIS extension and a few other goodies:

1. Visit *http://postgresapp.com/* and download the app's Disk Image file that ends in *.dmg*.
2. Double-click the *.dmg* file to open it, and then drag and drop the app icon into your *Applications* folder.
3. Double-click the app icon. When Postgres.app opens, click **Initialize** to create and start a PostgreSQL database.

A small elephant icon in your menu bar indicates that you now have a database running. To use included PostgreSQL command line tools, you'll need to open your Terminal application and run the following code at the prompt (you can copy the code as a single line from the Postgres.app site at *https://postgresapp.com/documentation/install.html*):

```
sudo mkdir -p /etc/paths.d &&
echo /Applications/Postgres.app/Contents/Versions/latest/bin | sudo tee /etc/paths.d/
postgresapp
```

Next, because Postgres.app doesn't include pgAdmin, you'll need to follow these steps to download and run pgAdmin:

1. Visit the pgAdmin site's page for macOS downloads at *https://www.pgadmin.org/download/pgadmin-4-macos/*.

2. Select the latest version and download the installer (look for a Disk Image file that ends in *.dmg*).
3. Double-click the *.dmg* file, click through the prompt to accept the terms, and then drag pgAdmin's elephant app icon into your *Applications* folder.
4. Double-click the app icon to launch pgAdmin.

NOTE *On macOS, when you launch pgAdmin the first time, a dialog might appear that displays "pgAdmin4.app can't be opened because it is from an unidentified developer." Right-click the icon and select* ***Open****. The next dialog should give you the option to open the app; going forward, your Mac will remember you've granted this permission.*

Installation on macOS is relatively simple, but if you encounter any issues, review the documentation for Postgres.app at *https://postgresapp.com/documentation/* and for pgAdmin at *https://www.pgadmin.org/docs/*.

Linux Installation

If you're a Linux user, installing PostgreSQL becomes simultaneously easy and difficult, which in my experience is very much the way it is in the Linux universe. Most popular Linux distributions—including Ubuntu, Debian, and CentOS—bundle PostgreSQL in their standard package. However, some distributions stay on top of updates more than others. The best path is to consult your distribution's documentation for the best way to install PostgreSQL if it's not already included or if you want to upgrade to a more recent version.

Alternatively, the PostgreSQL project maintains complete up-to-date package repositories for Red Hat variants, Debian, and Ubuntu. Visit *https://yum.postgresql.org/* and *https://wiki.postgresql.org/wiki/Apt* for details. The packages you'll want to install include the client and server for PostgreSQL, pgAdmin (if available), PostGIS, and PL/Python. The exact names of these packages will vary according to your Linux distribution. You might also need to manually start the PostgreSQL database server.

pgAdmin is rarely part of Linux distributions. To install it, refer to the pgAdmin site at *https://www.pgadmin.org/download/* for the latest instructions and to see whether your platform is supported. If you're feeling adventurous, you can find instructions on building the app from source code at *https://www.pgadmin.org/download/pgadmin-4-source-code/*.

Working with pgAdmin

Before you can start writing code, you'll need to become familiar with pgAdmin, which is the administration and management tool for PostgreSQL. It's free, but don't underestimate its performance. In fact, pgAdmin is a full-featured tool similar to tools for purchase, such as Microsoft's SQL Server Management Studio, in its capability to let you control multiple aspects of server operations. It includes a graphical interface for configuring and administrating your PostgreSQL server and databases, and—most appropriately for this book—offers a SQL query tool for writing, testing, and saving queries.

If you're using Windows, pgAdmin should come with the PostgreSQL package you downloaded from EnterpriseDB. On the Start menu, select

PostgreSQL ▸ pgAdmin 4 (the version number of Postgres should also appear in the menu). If you're using macOS and have installed pgAdmin separately, click the pgAdmin icon in your *Applications* folder, making sure you've also launched Postgres.app.

When you open pgAdmin, it should look similar to Figure 1.

Figure 1: The macOS version of the pgAdmin opening screen

The left vertical pane displays an object browser where you can view available servers, databases, users, and other objects. Across the top of the screen is a collection of menu items, and below those are tabs to display various aspects of database objects and performance.

Next, use the following steps to connect to the default database:

1. In the object browser, expand the plus sign (+) to the left of the Servers node to show the default server. Depending on your operating system, the default server name could be *localhost* or *PostgreSQL x*, where *x* is the Postgres version number.
2. Double-click the server name. Enter the password you chose during installation if prompted. A brief message appears while pgAdmin is establishing a connection. When you're connected, several new object items should display under the server name.
3. Expand *Databases* and then expand the default database `postgres`.
4. Under `postgres`, expand the *Schemas* object, and then expand *public*.

Your object browser pane should look similar to Figure 2.

NOTE *If pgAdmin doesn't show a default under Servers, you'll need to add it. Right-click Servers, and choose the Create Server option. In the dialog, type a name for your server in the General tab. On the Connection tab, in the Host name/address box, type* `localhost`*. Click* ***Save****, and you should see your server listed.*

This collection of objects defines every feature of your database server. There's a lot here, but for now we'll focus on the location of tables. To view a table's structure or perform actions on it with pgAdmin, this is where you can access the table. In Chapter 1, you'll use this browser to create a new database and leave the default `postgres` as is.

In addition, pgAdmin includes a *Query Tool*, which is where you write and execute code. To open the Query Tool, in pgAdmin's object browser, click once on any database to highlight it. For example, click the `postgres` database and then select **Tools ▸ Query Tool**. The Query Tool has two panes: one for writing queries and one for output.

It's possible to open multiple tabs to connect to and write queries for different databases or just to organize your code the way you would like. To open another tab, click another database in the object browser and open the Query Tool again via the menu.

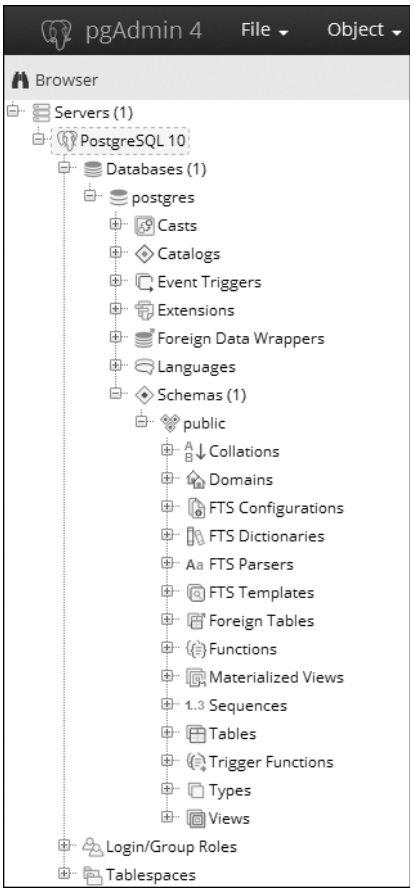


Figure 2: The pgAdmin object browser

Alternatives to pgAdmin

Although pgAdmin is great for beginners, you're not required to use it. If you prefer another administrative tool that works with PostgreSQL, feel free to use it. If you want to use your system's command line for all the exercises in this book, Chapter 16 provides instructions on using the PostgreSQL command line tool `psql`. (The Appendix lists PostgreSQL resources you can explore to find additional administrative tools.)

Wrapping Up

Now that you've installed PostgreSQL and pgAdmin, you're ready to start learning SQL and use it to discover valuable insights into your data!

In Chapter 1, you'll learn how to create a database and a table, and then you'll load some data to explore its contents. Let's get started!

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.

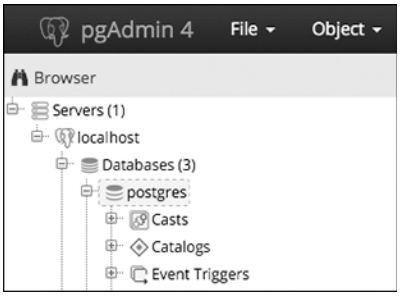


Figure 1-1: Connecting to the default `postgres` database

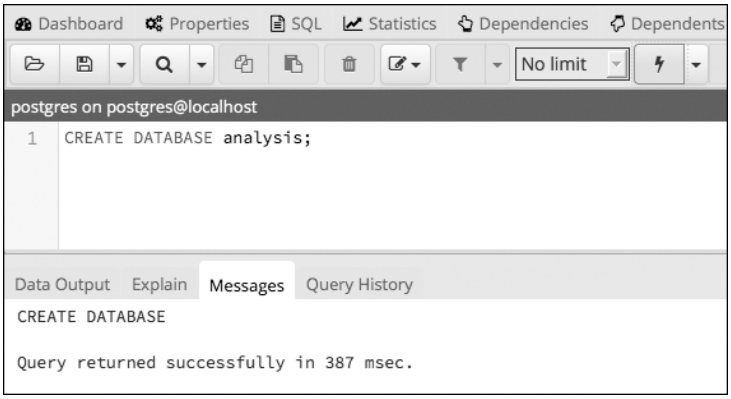


Figure 1-2: Creating the `analysis` database

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.

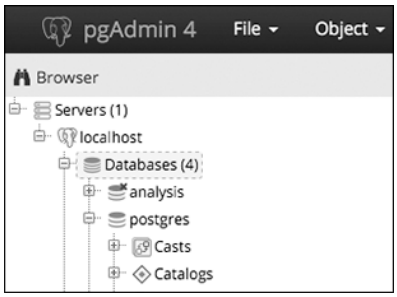


Figure 1-3: The `analysis` database displayed in the object browser

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.

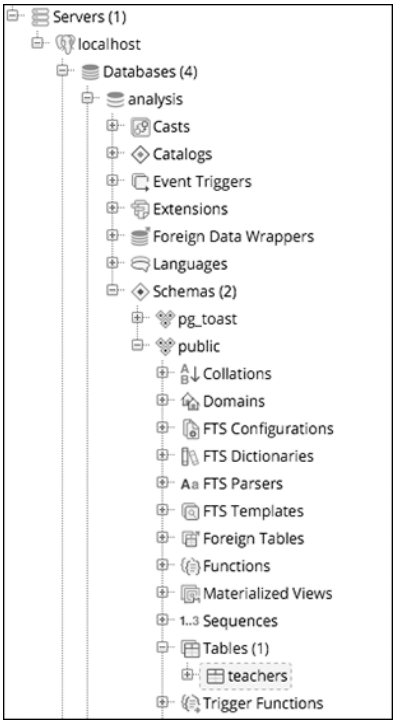


Figure 1-4: The teachers table in the object browser

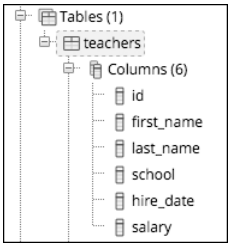


Figure 1-5: Table details for teachers

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?

2

BEGINNING DATA EXPLORATION WITH SELECT

For me, the best part of digging into data isn't the prerequisites of gathering, loading, or cleaning the data, but when I actually get to *interview* the data. Those are the moments when I discover whether the data is clean or dirty, whether it's complete, and most of all, what story the data can tell. Think of interviewing data as a process akin to interviewing a person applying for a job. You want to ask questions that reveal whether the reality of their expertise matches their resume.

Interviewing is exciting because you discover truths. For example, you might find that half the respondents forgot to fill out the email field in the questionnaire, or the mayor hasn't paid property taxes for the past five years. Or you might learn that your data is dirty: names are spelled inconsistently, dates are incorrect, or numbers don't jibe with your expectations. Your findings become part of the data's story.

In SQL, interviewing data starts with the `SELECT` keyword, which retrieves rows and columns from one or more of the tables in a database.

A SELECT statement can be simple, retrieving everything in a single table, or it can be complex enough to link dozens of tables while handling multiple calculations and filtering by exact criteria.

We'll start with simple SELECT statements.

Basic SELECT Syntax

Here's a SELECT statement that fetches every row and column in a table called my_table:

```
SELECT * FROM my_table;
```

This single line of code shows the most basic form of a SQL query. The asterisk following the SELECT keyword is a *wildcard*. A wildcard is like a stand-in for a value: it doesn't represent anything in particular and instead represents everything that value could possibly be. Here, it's shorthand for "select all columns." If you had given a column name instead of the wildcard, this command would select the values in that column. The FROM keyword indicates you want the query to return data from a particular table. The semicolon after the table name tells PostgreSQL it's the end of the query statement.

Let's use this SELECT statement with the asterisk wildcard on the teachers table you created in Chapter 1. Once again, open pgAdmin, select the analysis database, and open the Query Tool. Then execute the statement shown in Listing 2-1:

```
SELECT * FROM teachers;
```

Listing 2-1: Querying all rows and columns from the teachers table

The result set in the Query Tool's output pane contains all the rows and columns you inserted into the teachers table in Chapter 1. The rows may not always appear in this order, but that's okay.

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

Note that the id column (of type bigserial) automatically fills with sequential integers, even though you didn't explicitly insert them. Very handy. This auto-incrementing integer acts as a unique identifier, or key, that not only ensures each row in the table is unique, but also will later give us a way to connect this table to other tables in the database.

Let's move on to refining this query.

Querying a Subset of Columns

Using the asterisk wildcard is helpful for discovering the entire contents of a table. But often it's more practical to limit the columns the query retrieves, especially with large databases. You can do this by naming columns, separated by commas, right after the `SELECT` keyword. For example:

```
SELECT some_column, another_column, amazing_column FROM table_name;
```

With that syntax, the query will retrieve all rows from just those three columns.

Let's apply this to the `teachers` table. Perhaps in your analysis you want to focus on teachers' names and salaries, not the school where they work or when they were hired. In that case, you might select only a few columns from the table instead of using the asterisk wildcard. Enter the statement shown in Listing 2-2. Notice that the order of the columns in the query is different than the order in the table: you're able to retrieve columns in any order you'd like.

```
SELECT last_name, first_name, salary FROM teachers;
```

Listing 2-2: Querying a subset of columns

Now, in the result set, you've limited the columns to three:

```
last_name    first_name    salary
---------    ----------    ------
Smith        Janet         36200
Reynolds     Lee           65000
Cole         Samuel        43500
Bush         Samantha      36200
Diaz         Betty         43500
Roush        Kathleen      38500
```

Although these examples are basic, they illustrate a good strategy for beginning your interview of a data set. Generally, it's wise to start your analysis by checking whether your data is present and in the format you expect. Are dates in a complete month-date-year format, or are they entered (as I once ruefully observed) as text with the month and year only? Does every row have a value? Are there mysteriously no last names starting with letters beyond "M"? All these issues indicate potential hazards ranging from missing data to shoddy recordkeeping somewhere in the workflow.

We're only working with a table of six rows, but when you're facing a table of thousands or even millions of rows, it's essential to get a quick read on your data quality and the range of values it contains. To do this, let's dig deeper and add several SQL keywords.

Using DISTINCT to Find Unique Values

In a table, it's not unusual for a column to contain rows with duplicate values. In the `teachers` table, for example, the `school` column lists the same school names multiple times because each school employs many teachers.

To understand the range of values in a column, we can use the `DISTINCT` keyword as part of a query that eliminates duplicates and shows only unique values. Use the `DISTINCT` keyword immediately after `SELECT`, as shown in Listing 2-3:

```
SELECT DISTINCT school
FROM teachers;
```

Listing 2-3: Querying distinct values in the `school` column

The result is as follows:

```
school
------------------
F.D. Roosevelt HS
Myers Middle School
```

Even though six rows are in the table, the output shows just the two unique school names in the `school` column. This is a helpful first step toward assessing data quality. For example, if a school name is spelled more than one way, those spelling variations will be easy to spot and correct. When you're working with dates or numbers, `DISTINCT` will help highlight inconsistent or broken formatting. For example, you might inherit a data set in which dates were entered in a column formatted with a `text` data type. That practice (which you should avoid) allows malformed dates to exist:

```
date
---------
5/30/2019
6//2019
6/1/2019
6/2/2019
```

The `DISTINCT` keyword also works on more than one column at a time. If we add a column, the query returns each unique pair of values. Run the code in Listing 2-4:

```
SELECT DISTINCT school, salary
FROM teachers;
```

Listing 2-4: Querying distinct pairs of values in the `school` and `salary` columns

Now the query returns each unique (or distinct) salary earned at each school. Because two teachers at Myers Middle School earn $43,500, that pair is listed in just one row, and the query returns five rows rather than all six in the table:

```
school               salary
-------------------  ------
Myers Middle School  43500
Myers Middle School  36200
F.D. Roosevelt HS    65000
F.D. Roosevelt HS    38500
F.D. Roosevelt HS    36200
```

This technique gives us the ability to ask, "For each *x* in the table, what are all the *y* values?" For each factory, what are all the chemicals it produces? For each election district, who are all the candidates running for office? For each concert hall, who are the artists playing this month?

SQL offers more sophisticated techniques with aggregate functions that let us count, sum, and find minimum and maximum values. I'll cover those in detail in Chapter 5 and Chapter 8.

Sorting Data with ORDER BY

Data can make more sense, and may reveal patterns more readily, when it's arranged in order rather than jumbled randomly.

In SQL, we order the results of a query using a clause containing the keywords `ORDER BY` followed by the name of the column or columns to sort. Applying this clause doesn't change the original table, only the result of the query. Listing 2-5 shows an example using the `teachers` table:

```
SELECT first_name, last_name, salary
FROM teachers
ORDER BY salary DESC;
```

Listing 2-5: Sorting a column with `ORDER BY`

By default, `ORDER BY` sorts values in ascending order, but here I sort in descending order by adding the `DESC` keyword. (The optional `ASC` keyword specifies sorting in ascending order.) Now, by ordering the `salary` column from highest to lowest, I can determine which teachers earn the most:

```
first_name  last_name  salary
----------  ---------  ------
Lee         Reynolds   65000
Samuel      Cole       43500
Betty       Diaz       43500
Kathleen    Roush      38500
Janet       Smith      36200
Samantha    Bush       36200
```

SORTING TEXT MAY SURPRISE YOU

Sorting a column of numbers in PostgreSQL yields what you might expect: the data ranked from largest value to smallest or vice versa depending on whether or not you use the `DESC` keyword. But sorting a column with letters or other characters may return surprising results, especially if it has a mix of uppercase and lowercase characters, punctuation, or numbers that are treated as text.

During PostgreSQL installation, the server is assigned a particular *locale* for *collation*, or ordering of text, as well as a *character set*. Both are based either on settings in the computer's operating system or custom options supplied during installation. (You can read more about collation in the official PostgreSQL documentation at *https://www.postgresql.org/docs/current/static/collation.html*.) For example, on my Mac, my PostgreSQL install is set to the locale en_US, or U.S. English, and the character set UTF-8. You can view your server's collation setting by executing the statement `SHOW ALL;` and viewing the value of the parameter `lc_collate`.

In a character set, each character gets a numerical value, and the sorting order depends on the order of those values. Based on UTF-8, PostgreSQL sorts characters in this order:

1. Punctuation marks, including quotes, parentheses, and math operators
2. Numbers 0 to 9
3. Additional punctuation, including the question mark
4. Capital letters from A to Z
5. More punctuation, including brackets and underscore
6. Lowercase letters a to z
7. Additional punctuation, special characters, and the extended alphabet

Normally, the sorting order won't be an issue because character columns usually just contain names, places, descriptions, and other straightforward text. But if you're wondering why the word *Ladybug* appears before *ladybug* in your sort, you now have an explanation.

The ability to sort in our queries gives us great flexibility in how we view and present data. For example, we're not limited to sorting on just one column. Enter the statement in Listing 2-6:

```
SELECT last_name, school, hire_date
FROM teachers
❶ ORDER BY school ASC, hire_date DESC;
```

Listing 2-6: Sorting multiple columns with `ORDER BY`

In this case, we're retrieving the last names of teachers, their school, and the date they were hired. By sorting the `school` column in ascending order

and hire_date in descending order ❶, we create a listing of teachers grouped by school with the most recently hired teachers listed first. This shows us who the newest teachers are at each school. The result set should look like this:

```
last_name    school               hire_date
---------    -------------------  ----------
Smith        F.D. Roosevelt HS    2011-10-30
Roush        F.D. Roosevelt HS    2010-10-22
Reynolds     F.D. Roosevelt HS    1993-05-22
Bush         Myers Middle School  2011-10-30
Diaz         Myers Middle School  2005-08-30
Cole         Myers Middle School  2005-08-01
```

You can use ORDER BY on more than two columns, but you'll soon reach a point of diminishing returns where the effect will be hardly noticeable. Imagine if you added columns about teachers' highest college degree attained, the grade level taught, and birthdate to the ORDER BY clause. It would be difficult to understand the various sort directions in the output all at once, much less communicate that to others. Digesting data happens most easily when the result focuses on answering a specific question; therefore, a better strategy is to limit the number of columns in your query to only the most important, and then run several queries to answer each question you have.

Filtering Rows with WHERE

Sometimes, you'll want to limit the rows a query returns to only those in which one or more columns meet certain criteria. Using teachers as an example, you might want to find all teachers hired before a particular year or all teachers making more than $75,000 at elementary schools. For these tasks, we use the WHERE clause.

The WHERE keyword allows you to find rows that match a specific value, a range of values, or multiple values based on criteria supplied via an *operator*. You also can exclude rows based on criteria.

Listing 2-7 shows a basic example. Note that in standard SQL syntax, the WHERE clause follows the FROM keyword and the name of the table or tables being queried:

```
SELECT last_name, school, hire_date
FROM teachers
WHERE school = 'Myers Middle School';
```

Listing 2-7: Filtering rows using WHERE

The result set shows just the teachers assigned to Myers Middle School:

```
last_name    school               hire_date
---------    -------------------  ----------
Cole         Myers Middle School  2005-08-01
Bush         Myers Middle School  2011-10-30
Diaz         Myers Middle School  2005-08-30
```

Here, I'm using the equals comparison operator to find rows that exactly match a value, but of course you can use other operators with `WHERE` to customize your filter criteria. Table 2-1 provides a summary of the most commonly used comparison operators. Depending on your database system, many more might be available.

Table 2-1: Comparison and Matching Operators in PostgreSQL

Operator	Function	Example
`=`	Equal to	`WHERE school = 'Baker Middle'`
`<>` or `!=`	Not equal to*	`WHERE school <> 'Baker Middle'`
`>`	Greater than	`WHERE salary > 20000`
`<`	Less than	`WHERE salary < 60500`
`>=`	Greater than or equal to	`WHERE salary >= 20000`
`<=`	Less than or equal to	`WHERE salary <= 60500`
`BETWEEN`	Within a range	`WHERE salary BETWEEN 20000 AND 40000`
`IN`	Match one of a set of values	`WHERE last_name IN ('Bush', 'Roush')`
`LIKE`	Match a pattern (case sensitive)	`WHERE first_name LIKE 'Sam%'`
`ILIKE`	Match a pattern (case insensitive)	`WHERE first_name ILIKE 'sam%'`
`NOT`	Negates a condition	`WHERE first_name NOT ILIKE 'sam%'`

* The != operator is not part of standard ANSI SQL but is available in PostgreSQL and several other database systems.

The following examples show comparison operators in action. First, we use the equals operator to find teachers whose first name is Janet:

```
SELECT first_name, last_name, school
FROM teachers
WHERE first_name = 'Janet';
```

Next, we list all school names in the table but exclude F.D. Roosevelt HS using the not equal operator:

```
SELECT school
FROM teachers
WHERE school != 'F.D. Roosevelt HS';
```

Here we use the less than operator to list teachers hired before January 1, 2000 (using the date format *YYYY-MM-DD*):

```
SELECT first_name, last_name, hire_date
FROM teachers
WHERE hire_date < '2000-01-01';
```

Then we find teachers who earn $43,500 or more using the >= operator:

```
SELECT first_name, last_name, salary
FROM teachers
WHERE salary >= 43500;
```

The next query uses the `BETWEEN` operator to find teachers who earn between $40,000 and $65,000. Note that `BETWEEN` is *inclusive*, meaning the result will include values matching the start and end ranges specified.

```
SELECT first_name, last_name, school, salary
FROM teachers
WHERE salary BETWEEN 40000 AND 65000;
```

We'll return to these operators throughout the book, because they'll play a key role in helping us ferret out the data and answers we want to find.

Using LIKE and ILIKE with WHERE

Comparison operators are fairly straightforward, but `LIKE` and `ILIKE` deserve additional explanation. First, both let you search for patterns in strings by using two special characters:

Percent sign (%) A wildcard matching one or more characters

Underscore (_) A wildcard matching just one character

For example, if you're trying to find the word `baker`, the following `LIKE` patterns will match it:

```
LIKE 'b%'
LIKE '%ak%'
LIKE '_aker'
LIKE 'ba_er'
```

The difference? The `LIKE` operator, which is part of the ANSI SQL standard, is case sensitive. The `ILIKE` operator, which is a PostgreSQL-only implementation, is case insensitive. Listing 2-8 shows how the two keywords give you different results. The first `WHERE` clause uses `LIKE` ❶ to find names that start with the characters `sam`, and because it's case sensitive, it will return zero results. The second, using the case-insensitive `ILIKE` ❷, will return `Samuel` and `Samantha` from the table:

```
  SELECT first_name
  FROM teachers
❶ WHERE first_name LIKE 'sam%';

  SELECT first_name
  FROM teachers
❷ WHERE first_name ILIKE 'sam%';
```

Listing 2-8: Filtering with `LIKE` and `ILIKE`

Over the years, I've gravitated toward using `ILIKE` and wildcard operators in searches to make sure I'm not inadvertently excluding results from searches. I don't assume that whoever typed the names of people, places, products, or other proper nouns always remembered to capitalize them. And if one of the goals of interviewing data is to understand its quality, using a case-insensitive search will help you find variations.

Because `LIKE` and `ILIKE` search for patterns, performance on large databases can be slow. We can improve performance using indexes, which I'll cover in "Speeding Up Queries with Indexes" on page 108.

Combining Operators with AND and OR

Comparison operators become even more useful when we combine them. To do this, we connect them using keywords `AND` and `OR` along with, if needed, parentheses.

The statements in Listing 2-9 show three examples that combine operators this way:

```
SELECT *
FROM teachers
❶ WHERE school = 'Myers Middle School'
      AND salary < 40000;

SELECT *
FROM teachers
❷ WHERE last_name = 'Cole'
      OR last_name = 'Bush';

SELECT *
FROM teachers
❸ WHERE school = 'F.D. Roosevelt HS'
      AND (salary < 38000 OR salary > 40000);
```

Listing 2-9: Combining operators using `AND` and `OR`

The first query uses `AND` in the `WHERE` clause ❶ to find teachers who work at Myers Middle School and have a salary less than $40,000. Because we connect the two conditions using `AND`, both must be true for a row to meet the criteria in the `WHERE` clause and be returned in the query results.

The second example uses `OR` ❷ to search for any teacher whose last name matches Cole or Bush. When we connect conditions using `OR`, only one of the conditions must be true for a row to meet the criteria of the `WHERE` clause.

The final example looks for teachers at Roosevelt whose salaries are either less than $38,000 or greater than $40,000 ❸. When we place statements inside parentheses, those are evaluated as a group before being combined with other criteria. In this case, the school name must be exactly `F.D. Roosevelt HS` and the salary must be either less or higher than specified for a row to meet the criteria of the `WHERE` clause.

Putting It All Together

You can begin to see how even the previous simple queries allow us to delve into our data with flexibility and precision to find what we're looking for. You can combine comparison operator statements using the `AND` and `OR` keywords to provide multiple criteria for filtering, and you can include an `ORDER BY` clause to rank the results.

With the preceding information in mind, let's combine the concepts in this chapter into one statement to show how they fit together. SQL is particular about the order of keywords, so follow this convention:

```
SELECT column_names
FROM table_name
WHERE criteria
ORDER BY column_names;
```

Listing 2-10 shows a query against the `teachers` table that includes all the aforementioned pieces:

```
SELECT first_name, last_name, school, hire_date, salary
FROM teachers
WHERE school LIKE '%Roos%'
ORDER BY hire_date DESC;
```

Listing 2-10: A `SELECT` statement including `WHERE` and `ORDER BY`

This listing returns teachers at Roosevelt High School, ordered from newest hire to earliest. We can see a clear correlation between a teacher's hire date at the school and his or her current salary level:

```
first_name  last_name  school              hire_date   salary
----------  ---------  ------------------  ----------  ------
Janet       Smith      F.D. Roosevelt HS   2011-10-30  36200
Kathleen    Roush      F.D. Roosevelt HS   2010-10-22  38500
Lee         Reynolds   F.D. Roosevelt HS   1993-05-22  65000
```

Wrapping Up

Now that you've learned the basic structure of a few different SQL queries, you've acquired the foundation for many of the additional skills I'll cover in later chapters. Sorting, filtering, and choosing only the most important columns from a table can yield a surprising amount of information from your data and help you find the story it tells.

In the next chapter, you'll learn about another foundational aspect of SQL: data types.

TRY IT YOURSELF

Explore basic queries with these exercises:

1. The school district superintendent asks for a list of teachers in each school. Write a query that lists the schools in alphabetical order along with teachers ordered by last name A–Z.
2. Write a query that finds the one teacher whose first name starts with the letter S and who earns more than $40,000.
3. Rank teachers hired since January 1, 2010, ordered by highest paid to lowest.